# **Transcript: Speculative Execution of Untrusted JavaScript Code**

Mohan Dhawan Rutgers University Chung-chieh Shan Rutgers University Vinod Ganapathy Rutgers University

## Abstract

Transcript is a system that enhances JavaScript with support for speculative execution. It introduces a new *transaction* construct, which hosting Web applications can use to demarcate regions that contain untrusted guest code. Actions performed within a transaction are logged and considered speculative until they are examined by the host and committed. Uncommitted actions simply do not take and cannot affect the host in any way. Transcript therefore provides hosting Web applications with powerful mechanisms to mediate the actions of untrusted guests and also cleanly recover from the effects of security-violating guest code.

This paper describes the design of Transcript and its implementation in Firefox. Our exposition focuses on the novel features introduced by Transcript to support transactions, including a suspend/resume mechanism for JavaScript and support for speculative DOM updates. Our evaluation presents case studies showing that Transcript can be used to enforce powerful security policies on untrusted JavaScript code, and reports its performance on real-world applications and microbenchmarks.

## 1 Introduction

The client-side of modern Web applications routinely contains JavaScript code in the form of advertisements, games, and libraries that may be developed by untrusted To include such third-party code (the third parties. guest), the author of such a Web application (the host) must choose to (a) place the guest code in its own protection domain (e.g., an iframe) or (b) include the guest in the same protection domain as the host. The first approach isolates the guest from the host, but has the downside of offering limited means for guest/host interaction. Any communication between the guest and the host must happen via explicit postMessage commands between the corresponding iframes. The second approach eases guest/host interaction, but also exposes the host to malicious guests. Hosts must therefore employ protection mechanisms to confine the actions of guest code. This paper addresses the problem of designing such protection mechanisms.

One way to protect against untrusted guests is to use *JavaScript reference monitors*. Reference monitors mediate the action of guest code as it executes and ensure that any accesses by the guest to the host's data structures conform to the host's security policies. Examples of reference monitors include Caja [34], object capabilities [29], inlined reference monitors [16, 40, 47], object wrappers [28], and aspect-oriented policy enforcement [32]. Some reference monitors also require the guest code to be written in safe subsets of JavaScript (*e.g.*, Caja). The main purpose of these reference monitors is to provide *access control* to the host's objects.

This paper proposes a novel approach to construct JavaScript reference monitors using speculative execution. The main idea is to extend JavaScript with a new *transaction* construct, within which hosts can speculatively execute guest code that contains arbitrary JavaScript constructs. In addition to enforcing access control policies, reference monitors that use speculative execution allow hosts to *cleanly recover from the security-violating actions of guest code*. When a host detects an offending guest, it simply chooses not to commit the transaction corresponding to the guest. Such an approach neutralizes any data and DOM modifications initiated earlier by the guest, without having to undo them explicitly.

To motivate the need for recovery, consider the example of a Web-based word processor that hosts a thirdparty widget to display advertisements (see Figure 1). During an editing session, this widget scans the document for specific keywords and displays advertisements relevant to the text that the user has entered. Such a widget may modify the host in several ways to achieve its functionality, *e.g.*, it could install event handlers to display advertisements when the user places the mouse over specific phrases in the text. However, as an untrusted guest, this widget may also contain malicious functionality, *e.g.*, it could implement a clickjacking attack by overlaying the editor with transparent HTML elements pointing to malicious sites.

Traditional reference monitors can detect and prevent such attacks. However, such reference monitors typically only enforce access control policies, and would have let the guest modify the host's heap and DOM (such as to install innocuous event handlers) until the attack is detected. When such a reference monitor reports an attack, the end-user faces one of two unpalatable options: (a) close the editing session and start afresh; or (b) continue with the tainted editing session. In the former case, the end-user loses unsaved work. In the latter case, the

(1)	<script type="text/javascript"></th><th>(14)</th><th>for(var i in builtins) {</th></tr><tr><td>(2)</td><td><pre>var editor = new Editor(); initialize(editor);</pre></td><td>(15)</td><td>if (ws.checkMembership(Editor.prototype, builtins[i])) tocommit = false:</td></tr><tr><td>(3)</td><td><pre>var builtins = [], tocommit = true;</pre></td><td>(16)</td><td><pre>} /* definition of 'IsClickJacked' to go here */</pre></td></tr><tr><td>(4)</td><td><pre>for(var prop in Editor.prototype) builtins[prop] = prop;</pre></td><td>(17)</td><td><pre>if (IsClickJacked(tx.getTxDocument())) tocommit = false;</pre></td></tr><tr><td>(5)</td><td><pre>var tx = transaction {     Guest code: Lines 6-9 </pre></td><td>(18)</td><td><pre> /* more policy checks go here */</pre></td></tr><tr><td>(6)</td><td>Editor.prototype.getKeywords = function(content) {}</td><td>(19)</td><td><pre>switch(tx.getCause()) {</pre></td></tr><tr><td></td><td></td><td>(20)</td><td>case "addEventListener":</td></tr><tr><td>(7)</td><td><pre>var elem = document.getElementById("editBox");</pre></td><td>(21)</td><td><pre>var txHandler = MakeTxHandler(arg[1]);</pre></td></tr><tr><td>(8)</td><td><pre>elem.addEventListener("mouseover", displayAds, false);</pre></td><td>(22)</td><td><pre>obj.addEventListener(arg[0], txHandler, arg[2]); break;</pre></td></tr><tr><td></td><td></td><td>(23)</td><td><pre>case "write": WriteToTxDOM(obj, arg[0]); break; /* more cases */</pre></td></tr><tr><td>(9)</td><td><pre>document.write('<div style="opacity:0.0; z-index:0; size/loc params"></pre></td><td>(24)</td><td>default: break;</td></tr><tr><td></td><td><a href="http://evil.com"> Evil Link </a></div>');</td><td>(25)</td><td><pre>}; tx = tx.resume();</pre></td></tr><tr><td>(10)</td><td>};</td><td>(26)</td><td><pre>} while(tx.isSuspended());</pre></td></tr><tr><td>(11)</td><td>do { Introspection block (iblock): Lines 11–27</td><td>(27)</td><td><pre>if (tocommit) tx.commit();</pre></td></tr><tr><td>(12)</td><td><pre>var arg = tx.getArgs(): var obj = tx.getObject():</pre></td><td>(28)</td><td><pre> /* rest of the Host Web application's code */</pre></td></tr><tr><td>(13)</td><td><pre>var rs = tx.getReadSet(); var ws = tx.getWriteSet();</pre></td><td>(29)</td><td></script>
-----	--

Figure 1: Motivating example. This example shows how a hosting Web application can mediate untrusted guest code (lines 6–9). The introspection block (lines 11–27) enforces the host's security policies on the actions performed by the guest. To ease exposition, (a) this example omits variable hiding, a concept introduced in Section 3.2, which ensures that the guest cannot tamper with sensitive host variables such as tx, tocommit and builtins; (b) we inlined the guest code from http://untrusted.com/guest.js in lines 6–9. Section 5.4 explains how the host can use transactions to mediate guests included using a <script> tag.

editing session is subject to the unknown and possibly undesirable effects of the heap and DOM changes that the widget initiated before being flagged as malicious. In our example, the event handlers registered by the malicious widget may also implement undesirable functionality and should be removed when the widget's clickjacking attempt is detected.

Speculative execution allows reference monitors to detect and recover from malicious guests. In our example, the host would speculatively execute the untrusted widget by enclosing it within a transaction. When the attack is detected, the host simply discards all changes initiated by the widget. The end-user can proceed with the editing session without losing unsaved work, and with the assurance that the host is unaffected by the malicious widget.

This paper describes Transcript, a speculative execution system for JavaScript with several novel features:

(1) JavaScript transactions. Transcript allows hosting Web applications to speculatively execute guests by enclosing them in transactions. Transcript maintains *read and write sets* for each transaction to record the objects that are accessed and modified by the corresponding guest. These sets are exposed as properties of a *transaction object* in JavaScript. Changes to a JavaScript object made by the guest are visible within the transaction, but any accesses to that object from code outside the transaction return the unmodified object. The host can inspect such speculative changes made by the guest and determine whether they conform to its security policies. The host must explicitly commit these changes in order for them to take effect; uncommitted changes simply do not take and need not be undone explicitly.

(2) **Transaction suspend/resume.** Guest code may attempt operations outside the purview of the JavaScript interpreter. In a browser, these *external operations* include AJAX calls that send network requests, such as XMLHttpRequest. Transcript introduces a *suspend and resume* mechanism that affords unprecedented flexibility to

mediate external operations. Whenever a guest attempts an external operation, Transcript suspends it and passes control to the host. Depending on its security policy, the host can perform the action on behalf of the guest, perform a different action unbeknownst to the guest, or buffer up and simulate the action, before resuming this or another suspended transaction.

(3) **Speculative DOM updates.** Because JavaScript interacts heavily with the DOM, Transcript provides a speculative DOM subsystem, which ensures that DOM changes requested by a guest will also be speculative. Together with support for JavaScript transactions, Transcript's DOM subsystem allows hosts to cleanly recover from attacks by malicious guests.

Transcript provides these features without restricting or modifying guest code in any way. This allows reference monitors based on Transcript to mediate the actions of legacy libraries and applications that contain constructs that are often disallowed in safe JavaScript subsets [13, 18, 28, 30, 34] (*e.g.*, eval, this and with).

We have implemented a prototype of Transcript by adding these features to Firefox's JavaScript interpreter. We evaluated our prototype by using it to build reference monitors to mediate several guests, the largest of which was a jQuery-based application containing about 7,500 lines of code. Our results show that Transcript is expressive and effective at isolating untrusted guests. On average, the use of Transcript increased the load time of these guests by only 0.16 seconds.

## 2 Overview of Transcript

We provide a high-level overview of Transcript in action by further elaborating on the example introduced in Section 1. Suppose that the word processor hosts the untrusted widget using a <script> tag, as follows: <script src="http://untrusted.com/guest.js">. In Figure 1, lines 6–9 show a snippet from guest.js, which displays advertisements relevant to keywords entered in the editor. Line 6 registers a function to scan for keywords in the editor window by adding it to the prototype of the Editor object. Lines 7 and 8 show the widget registering an event handler to display advertisements on certain mouse events. While lines 6–8 encode the core functionality related to displaying advertisements, line 9 implements a clickjacking attack by creating a transparent <div> element, placed suitably on the editor with a link to an evil URL.

When hosting such a guest, the word processor must protect itself from attacks by defining and enforcing a suitable set of security policies. These may include policies to prevent prototype hijacks [39], clickjacking attacks, drive-by downloads, stealing cookies, snooping on keystrokes, *etc.* Further, if an attack is detected and prevented, it should not adversely affect normal operation of the word processor. We now illustrate how the word processor can use Transcript to achieve such protection and cleanly recover from attempted attacks.

The host protects itself by embedding the guest in a transaction construct (line 5) and specifies its security policy (lines 11–27). When the transaction executes, Transcript records all reads and writes to JavaScript objects in per-transaction read/write sets. Any attempts by the guest to modify the host's JavaScript objects (*e.g.*, on line 6) are speculative; *i.e.*, these changes are visible only to the guest itself and do not modify the host's view of the JavaScript heap. To ensure that DOM modifications by the guest are also speculative, Transcript's DOM subsystem clones the host's DOM at the start of the transaction and resolves all references to DOM objects within a transaction to the cloned DOM. Thus, references to document within the guest resolve to the cloned DOM.

When the guest performs DOM operations, such as those on lines 7-9, and other external operations, such as XMLHttpRequest, Transcript suspends the transaction and passes control to the host. This situation is akin to a system call in a user-space program causing a trap into the operating system. Suspension allows hosts to mediate external operations as soon as the guest attempts them. When a transaction suspends or completes execution, Transcript creates a transaction object in JavaScript to denote the completed or suspended transaction. In Figure 1, the variable tx refers to the transaction object. Transcript then passes control to the host at the program point that syntactically follows the transaction. There, the host implements an introspection block (or iblock) to enforce its security policy and perform operations on behalf of a suspended transaction.

**Transaction objects.** A transaction object records the state of a suspended or completed transaction. It stores the read and write sets of the transaction and the list of activation records on the call stack of the transaction when it was suspended. It provides builtin methods, such as getReadSet and getWriteSet shown in Figure 1, that the

host can invoke to access read and write sets, observe the actions of the guest, and make policy decisions.

When a guest tries to perform an external operation and thus suspends, the resulting transaction object may contain arguments passed to the operation. For example, a transaction that suspends due to an attempt to modify the DOM, such as the call document.write on line 9, will contain the DOM object referenced in the operation (document), the name of the method that caused the suspension (write), and the arguments passed to the method. (Recall that Transcript's DOM subsystem ensures that document referenced within the transaction will point to the cloned DOM.) The host can access these arguments using builtin methods of the transaction object, such as getArgs, getObject and getCause. These arguments are analogous to system-call arguments passed from a userspace program to the operating system. Depending on its security policy, the host can either perform the operation on behalf of the guest, simulate the effect of performing it, defer the operation for later, or not perform it at all.

The host can resume a suspended transaction using the transaction object's builtin resume method. Transcript then uses the activation records stored in the transaction object to restore the call stack, and resumes control at the program point following the instruction that caused the transaction to suspend (akin to resumption of program execution following a system call). Transactions can suspend an arbitrary number of times until they complete execution. The builtin isSuspended method determines whether the transaction is suspended or has completed.

A completed transaction can be committed using the builtin commit method. This method copies the contents of the write set to the corresponding objects on the host's heap, thereby publishing the changes made by the guest. It also synchronizes the host's DOM with the cloned version that contains any DOM modifications made by the guest. A completed transaction's call stack is empty, so attempts to resume a completed transaction will have no effect. Note that Transcript does not define an explicit abort operation. This is because the host can simply discard changes made by a transaction by choosing not to commit them. If the transaction object is not referenced anymore, it will be garbage-collected.

**Introspection blocks.** When a transaction suspends or completes, Transcript passes control to the instruction that syntactically follows the transaction in the code of the host. At this point, the host can check the guest's actions by encoding its security policies in an *iblock*. The iblock in Figure 1 spans lines 11–27 and has two logical parts: a *host-specific part* that encodes the host's security policies (lines 14–18), and a *mandatory part* that performs operations on behalf of suspended guests (lines 19–25).

The iblock in Figure 1 illustrates two policies:

(1) Lines 14–15 detect prototype hijacking attempts on

the Editor object. To do so, they check the transaction's write set for attempted redefinitions of builtin methods and fields of the Editor object.

(2) Line 17 detects clickjacking attempts by checking the DOM for the presence of any transparent HTML elements introduced by the guest. (The body of IsClickJacked, which implements the check, is omitted for brevity).

The body of the switch statement encodes the mandatory part of the iblock and implements two key functionalities, which are further explained in Section 3.1:

(1) Lines 20–22 create and attach an event handler to the cloned DOM when the guest suspends on line 8. The MakeTxHandler function creates a new *wrapped* handler, by enclosing the guest's event handler (displayAds) within a transaction construct. Doing so ensures that the execution of any event handlers registered by the guest is also speculative, and mediated by the host's security policies. The iblock then attaches the event handler to the corresponding element (elem) in the cloned DOM.

(2) Line 23 speculatively executes the DOM modifications requested when the guest suspends on line 9. The WriteToTxDOM function invokes the write call on obj, which points to the document object in the cloned DOM.

If a transaction does not commit because of a policy violation, the host's DOM and JavaScript objects will remain unaffected by the guest's modifications. For instance, when the host in Figure 1 aborts the guest after it detects the clickjacking attempt, the host's DOM will not contain any remnants of the guest's actions (such as event handlers registered by the guest). The host's JavaScript objects, such as Editor, are also unaffected. Speculatively executing guests therefore allows hosts to cleanly recover from attack attempts.

Iblocks offer hosts the option to postpone external operations. For example, a host may wish to defer all network requests from an untrusted advertisement until the end of the transaction. It can do so using an iblock that buffers these requests when they suspend, and thereafter resume the transaction; the buffered requests can be processed after the transaction has completed. Such postponement will not affect the guest if the buffered requests are asynchronous, *e.g.*, XMLHttpRequest.

Because a transaction may suspend several times, the iblock is structured as a loop, whose body executes each time the transaction suspends and once when the transaction completes. This way, the same policy checks apply whether the transaction suspended or completed.

### **3** Design of Transcript

We now describe the design of Transcript's mechanisms using Figure 2, which summarizes the workflow of a Transcript-enhanced host. The figure shows the operation of the Transcript runtime system at key points during the execution of the host, which has confined a guest akin to the one in Figure 1 using a transaction.

When a transaction begins execution, Transcript first provides the transaction with its private copy of the host's DOM tree. It does so by cloning the current state of the host's DOM, including any event handlers associated with the nodes of the DOM (① in Figure 2). When a guest references nodes in the host's DOM, Transcript redirects these references to the corresponding nodes in the transaction's private copy of the DOM.

Next, the Transcript runtime pushes a *transaction de-limiter* on the JavaScript call stack. Transcript places the activation records of methods invoked within the transaction above this delimiter. It also records the locations of JavaScript objects accessed/modified within the transaction in read/write sets. If the transaction executes an external operation, the runtime suspends the transaction. To do so, it creates a transaction object and (a) initializes the object with the transaction's read/write sets; (b) pops all the activation records on the JavaScript call stack until the topmost transaction delimiter; (c) stores these activation records in the transaction object; (d) saves the program counter; and (e) sets the program counter to immediately after the end of the transaction, *i.e.*, the start of the iblock (steps <sup>(2)</sup>) and <sup>(3)</sup> in Figure 2).

The iblock logically extends from the end of the transaction to the last resume or commit call on the transaction object (*e.g.*, lines 11–27 in Figure 1). The iblock can access the transaction object and its read/write sets to make policy decisions. If the iblock invokes resume on a suspended transaction, the Transcript runtime (a) pushes a transaction delimiter on the current JavaScript call stack; (b) pushes the activation records saved in the transaction object; and (c) restores the program counter to its saved value. Execution therefore resumes from the statement following the external operation (see (a) and (b)). If the iblock invokes commit instead, the Transcript runtime updates the JavaScript heap using the values in the transaction object's write set. The commit operation also replaces the host's DOM with the cloned DOM (step (b)).

The Transcript runtime behaves in the same way even when transactions are nested: Transcript pushes a new delimiter on the JavaScript call stack for each level of nesting encountered at runtime. Each suspend operation only pops activation records until the topmost delimiter on the stack. Nesting is important when a guest itself wishes to confine code that it does not trust. This situation arises when a host includes a guest from a first-tier advertising agency (1sttier.com), which itself includes code from a second-tier agency (2ndtier.com). Whether the host confines the advertisement using an outer transaction, 1sttier.com may itself confine code from 2ndtier.com using an *inner* transaction using its own security policies. If code from 2ndtier.com attempts to modify the DOM, that call suspends and traps to the iblock defined by 1sttier.com. If this iblock attempts



(a) Locations of traps/returns to/from Transcript. (b) Corresponding actions within the Transcript runtime system for a trap/return.

Figure 2: Workflow of a Transcript-enhanced host. Part (a) of the figure shows a host enclosing a guest within a transaction, while part (b) shows the JavaScript runtime and the DOM subsystem. The labels O-O in the figure show: O the host's DOM being cloned at the start of the transaction, O the host's call stack before a call that suspends the transaction, O the call stack after suspension, O the host's call stack when the transaction is about to resume; the speculative DOM has been updated with the requested changes, O the host's call stack just after resumption, O shows the transaction committing, which copies all speculative changes to the host's DOM and JavaScript heap. The thick lines on the call stacks denote transaction delimiters. Arrows show control transfer from the transaction to the iblock and back.

to modify the DOM on behalf of 2ndtier.com, the outer transaction suspends in turn and passes control to the host's iblock. In effect, the DOM modification succeeds only if it is permitted at *each* level of nesting.

#### **3.1** Components of an iblock

As discussed in Section 2, an iblock consists of two parts: a *host-specific* part, which codifies the host's policies to confine guests, and a *mandatory* part, which contains functionality that is generic to all hosts. In our implementation, we have encoded the second part as a JavaScript library (libTranscript) that can simply be included into the iblock of a host.<sup>1</sup> This mandatory part implements two functionalities: gluing execution contexts and generating wrappers for event handlers.

#### 3.1.1 Gluing execution contexts

Guests often use document.write or similar calls to modify the host's DOM, as shown on line 9 of Figure 1. When such guests execute within a transaction, the document.write call traps to the iblock, which must complete the call on behalf of the guest and render the HTML in the cloned DOM. However, the HTML code in document.write may contain scripts, *e.g.*, document.write('<script src=code.js>'). The execution of code.js, having been triggered by the guest, must then be mediated by the same security policy that governs the guest.

Thus, code.js should be executed in the same context as the transaction where the guest executes. To achieve this goal, the mandatory part of the iblock encapsulates the content of code.js into a function and uses a builtin glueresume method of the transaction object to instruct the Transcript runtime to invoke this function when it resumes the suspended transaction. The net effect is similar to fetching and inlining the content of code.js into the transaction. We call this operation *gluing*, because it glues the code in code.js to that of the guest.

To implement gluing, the iblock must recognize that the document.write includes additional scripts. This in turn requires the iblock to parse the HTML argument to document.write. We therefore exposed the browser's HTML parser through a new document.parse API to allow HTML (and CSS) parsing in iblocks. This API accepts a HTML string argument, such as the argument to document.write, and parses it to recognize <script> elements and other HTML content. It also recognizes inline event-handler registrations, so that they can be wrapped as described in Section 3.1.2. When the iblock invokes document.parse (in Figure 1, it is invoked within the call to WriteToTxDOM on line 23), the parser creates new functions that contain code in <script> elements. It returns these functions to the host's iblock, which can then invoke them by gluing. The parser also renders other (nonscript) HTML content in the cloned DOM.

Guest operations involving innerHTML are handled simlarly. Transcript suspends a guest that attempts an innerHTML operation, parses the new HTML code for any scripts, and glues their execution into the guest's context.

### 3.1.2 Generating wrappers for event handlers

Guests executing within a transaction may attempt to register functions to handle asynchronous events. For example, line 8 in Figure 1 registers displayAds as an onMouseOver handler. Because displayAds is guest code, it is important to associate it with the iblock for the transaction that registered it and to subject it to the same pol-

<sup>&</sup>lt;sup>1</sup>The body of the switch statement in Figure 1 shows a snippet from this JavaScript library. Figure 7 illustrates an iblock that simply includes and invokes functions from the library.

(1)	var tx = transaction { //code that suspends
(2)	for (var x in this) {
(3)	<pre>if (this[x] instanceof Tx_obj) txref = this[x];</pre>
(4)	<pre>}; txref.getWriteSet = function() { };</pre>
(5)	}

Figure 3: A guest that implements a reference leak. The tx object is created and attached to this when the guest suspends on line 1.

icy checks. Transcript does so by creating a new function  $tx\_displayAds$  that *wraps* displayAds within a transaction guarded by the same iblock, and registering  $tx\_displayAds$  as the event handler for the onMouseOver event.

To this end, the mandatory part of the iblock includes creating wrappers (such as tx\_displayAds) for When the guest executes a stateevent handlers. ment such as elem.addEventListener(...), it would trap to the iblock, which can then examine the arguments to this call and create a wrapper for the event handler. Guests can alternatively use document.write calls to register event handlers e.g., document.write ('<div onMouseOver="displayAds();">'). In this case, the iblock recognizes that a new event handler is being registered by parsing the HTML argument of the document.write call (using the document.parse API) when it suspends, and wraps the call. Our wrapper generator deals with all the event handling models currently supported by Firefox [46]. We refer readers interested in the details to Appendix A.

Besides event handlers, JavaScript supports other constructs for asynchronous execution: AJAX callbacks (*i.e.*, XMLHttpRequest), which execute upon receiving network events, and features such as setTimeOut and setInterval that trigger code execution based upon timer events. The mandatory part of the iblock also handles these constructs by wrapping callbacks as just described.

## 3.2 Hiding sensitive variables

The iblock of a transaction checks the guest's actions against the host's policies. These policies are themselves encoded in JavaScript, and may use methods and variables (*e.g.*, tx, tocommit and builtins in Figure 1) that must be protected from the guest. Without precautions, the guest can use JavaScript's extensive reflection capabilities to tamper with these sensitive variables. Figure 3 presents an example of one such attack, a reference leak, where the malicious guest obtains a reference to the tx object by enumerating the properties of the this object, and redefines the method tx.getWriteSet speculatively. As presented, the example in Figure 1 is vulnerable to such a reference leak.

To protect such sensitive variables, we adopt a defense called *variable hiding* that eliminates the possibility of leaks by construction. This technique mandates that guests be placed outside the scope of the iblock's variables, such as tx. The basic idea is to place the guest and the iblock in separate, lexically scoped functions, so that

variables such as tx, tocommit and builtins are not accessible to the guest. By so hiding sensitive variables from the guest, this defense prevents reference leaks. Figure 7 illustrates this defense with a code snippet, after introducing some more details of our implementation.

## 4 Security Assurances

Transcript's ability to protect hosts from untrusted guests depends on two factors: (a) the assurance that a guest cannot subvert Transcript's mechanisms, *i.e.*, the robustness of the trusted computing base; and (b) host-specific policies used to confine guests.

## 4.1 Trusted computing base

Transcript's trusted computing base (TCB) consists of the runtime component implemented in the browser and the mandatory part of the host's iblock. The TCB provides the following security properties: (a) *complete mediation*, *i.e.*, control over all JavaScript and external operations performed by a guest; and (b) *isolation*, *i.e.*, the ability to confine the effects of the guest.

(1) Complete mediation. The Transcript runtime and the mandatory part of the host's iblock together ensure complete mediation of guest execution. The runtime: (a) records all guest accesses to the host's JavaScript heap in the corresponding transaction's read/write sets; (b) causes a trap to the host's iblock when the guest attempts an external operation; and (c) redirects all guest references to the host's DOM to the cloned DOM. The mandatory part of the iblock, consisting of wrapper generators and the HTML parser, ensures that any additional code fetched by the guest or scheduled for later execution (e.g., event handlers and callbacks for XMLHttpRequest) will itself be enclosed within transactions mediated by the same iblock. This process recurs so that the host's policies mediate all guest code, even event handlers installed by callbacks of event handlers.

(2) Isolation. Transcript isolates guest operations using speculative execution. It records changes to the host's JavaScript heap within the guest transaction's write set, and changes to the host's DOM within the cloned DOM. The host then has the opportunity to review these speculative changes within its iblock and ensure that they conform to its security policies. Observe that a suspended/completed transaction may provide the host with references to objects modified by the guest, e.g., in Figure 1, a reference to elem is passed to the iblock via the getObject API. Speculative execution ensures that if the transaction has not yet been committed, then accesses to the object's methods and fields via this reference will still resolve to their values at the beginning of the transaction. Thus, for instance, a call to the toString method of the elem object in the iblock of Figure 1 would still work as intended if even if the guest had redefined this method within the transaction. Note that variables hidden from the guest cannot even be *speculatively* modified, thereby automatically isolating them from the guest.

Together, the above properties ensure the following invariant: At the point when a transaction suspends or completes execution and is awaiting inspection by the host's iblock, none of the host's JavaScript objects or its DOM would have been modified by the guest. Further, host variables hidden from the guest will not be modified even after the transaction has committed. Overall, executing a transaction never incurs any side effect, and any side effect that would be incurred by committing a transaction can be first vetted by inspecting the transaction.

## 4.2 Best practices for host policies

Hosts can import the speculative changes made by a guest after inspecting them against their security policies. Even though complete mediation and isolated execution ensure that the core *mechanisms* of Transcript cannot be subverted by guest execution (*i.e.*, they ensure that all of the guest's speculative actions will be available for inspection by the host), the ability of the host to isolate itself from the guest ultimately depends on its *policies*.

Host policies are necessarily domain-specific and have to be written manually in our current prototype. Though our experiments (Section 6.4) suggest that the effort required to write policies in Transcript is comparable to that required in other systems, writing policies is admittedly a difficult exercise. Further research is needed to develop tools for policy authors to debug/verify the completeness of their policies. Nevertheless, we outline a set of "best practices" below to avoid common pitfalls.

(1) Use whitelisting. Iblocks should employ a whitelist that specifies host objects that can legitimately be modified by the guest and reject attempts to modify objects outside the whitelist. This guideline may cause false positives if the whitelist is not comprehensive. For example, both window.location and window.location.href can be used to change the location field of the host, but a whitelist that includes only one will reject guests that modify guest location using the other. Nevertheless, whitelisting allows hosts to be conservative when allowing guests to modify their objects. The following two guidelines are subsumed if whitelisting is employed, but should be followed if it is not convenient to write a whitelist-based policy.

(2) Validate global objects and caller arguments. A guest executing in a transaction can speculatively modify any objects visible in its scope. These may include host-defined global variables, as well as arguments passed to functions currently on the call stack (*e.g.*, via arguments.callee.caller). The host must therefore check whether any of its objects that are visible to the guest appear in the write set of the transaction, and vet these changes against its security policy.

Our prototype implementation provides a library of util-

API	Description
getReadSet	Exports transaction's read set to JavaScript.
getWriteSet	Exports transaction's write set to JavaScript.
getTxDocument	Returns a reference to the speculative document object.
isSuspended	Returns true if the transaction is suspended.
getCause	Returns cause of a transaction suspend.
getObject	Returns object reference on which a suspension was invoked.
getArgs	Returns set of arguments involved in a transaction suspend.
resume	Resumes suspended transaction.
glueresume	Resumes suspended transaction and glues execution contexts.
isDOMConflict	Checks for conflicts between the host's and cloned DOM.
isHeapConflict	Checks for conflicts between the host and guest heaps.
commit	Commits changes to host's JavaScript heap and DOM.

Figure 4: Key functions in the API to access transaction objects.

ities that policy authors can use to detect modifications to objects that are in-scope. This library implements modifGlobals, modifCallers and modifNatives, which an iblock author can use to obtain a list of host-defined globals, caller arguments and native objects (*e.g.*, String, Object, Array) that the guest has speculatively modified. These utilities can ease the burden of writing iblocks. As an example, the following snippet included in an iblock would reject guests that inspect and modify arguments on the call stack: if (modifCallers(tx).length != 0) tocommit = false;, where tx is the transaction object.

(3) Defend against prototype hijacks and type forgery. A malicious guest can redefine builtin object properties to implement prototype hijacking and type forgery attacks. For example, a guest can redefine the toString and valueOf methods of an object to return entities of arbitrary type. If the host commits the transaction despite this redefinition, it will be vulnerable to type forgery when it subsequently invokes these methods.

The host's iblocks must therefore include code to check the write set for redefinitions of builtin methods such as toString and valueOf or attempts to modify an object via its prototype field. We include this guideline as a best practice rather than including it as a defense in the mandatory part of the iblock because there may be domain-specific scenarios where violations are acceptable, *e.g.*, in Figure 1, where the host allows limited modifications to the Editor object via its prototype field.

#### **5** Implementation in Firefox

We implemented Transcript by modifying Firefox (version 3.7a4pre). Overall, our prototype adds or modifies about 6,400 lines of code in the browser. The bulk of this section describes Transcript's enhancements to SpiderMonkey (the JavaScript interpreter used by Firefox) (Section 5.1) and its support for speculative DOM updates (Section 5.2). We also discuss Transcript's support for conflict detection (Section 5.3) and the need to modify the <script> tag (Section 5.4).

### 5.1 Enhancements to SpiderMonkey

Our prototype enhances SpiderMonkey in five ways:

(1) *Transaction objects*. We added a new class of JavaScript objects to denote transactions. This object stores a pointer to the read/write sets, activation records

of the transaction, and to the cloned DOM. It implements the builtin methods shown in Figure 4.

(2) A transaction keyword. We added a transaction keyword to the syntax of JavaScript. When the Transcript-enhanced JavaScript parser encounters this keyword, it (a) compiles the body of the transaction into an anonymous function; (b) inserts a new instruction, JSOP\_BEGIN\_TX, into the generated bytecode to signify the start of a transaction; and (c) inserts code to invoke the anonymous function. The transaction ends when the anonymous function returns a transaction object when it suspends or completes execution.

(3) *Read/write sets.* Transcript adds read/write setmanipulation to the interpretation of several JavaScript bytecode instructions. We enhanced the interpreter so that each bytecode instruction that accesses or modifies JavaScript objects additionally checks whether its execution is within a transaction (*i.e.*, if an unfinished JSOP\_-BEGIN\_TX was previously encountered in the bytecode stream). If so, the execution of the instruction also logs an identifier denoting the JavaScript object (or property) accessed/modified in its read/write sets, which we implemented using hash tables. We used SpiderMonkey's identifiers for JavaScript objects; references using aliases to the same object will return the same identifier.

(4) Suspend. We modified the interpreter's implementation of bytecode instructions that perform external operations and register event handlers to suspend when executed within a transaction. The suspend operation and the built-in resume function of transaction objects are implemented as shown in Figure 2. We also introduced a suspend construct that allows hosts to customize transaction suspension. Hosts can include this construct within a transaction (before including guest code) to register custom suspension points. The call suspend [obj.foo] suspends the transaction when it invokes foo (if it is a method) or attempts to read from or write to the property foo of obj.

(5) *Garbage Collection.* We interfaced the JavaScript interpreter with the garbage collector to traverse and mark all heap objects that are reachable from live transaction objects. This avoids any inadvertent garbage collection of objects still reachable from suspended transactions that could be resumed in the future.

Integrating these changes into a legacy JavaScript interpreter proved to be a challenging exercise. We refer interested readers to Appendix B for a description of how our implementation addressed one such challenge, nontail recursive calls in SpiderMonkey.

## 5.2 Supporting speculative DOM updates

Transcript provides each executing transaction with its private copy of the host's document structure and uses

this copy to record all DOM changes made by guest code. This section presents notable details of the implementation of Transcript's DOM subsystem.

Transcript constructs a replica of the host's DOM when it encounters a JSOP\_BEGIN\_TX instruction in the bytecode stream. It clones nodes in the host's DOM to copy references to any event handlers and dynamically-attached JavaScript properties associated with the node. If a guest attempts to modify an event handler associated with a node, the reference is rewritten to point to the function object in the transaction's write set.

Crom [33] also implemented DOM cloning for speculative execution (albeit not for the purpose of mediating untrusted code). Unlike Crom, which implemented DOM cloning as a JavaScript library, Transcript implements cloning in the browser itself. This feature simplifies several issues that Crom's designers faced (*e.g.*, cloning DOM-level 2 event handlers) and also allows efficient cloning.

When a guest references a DOM node within a transaction, Transcript transparently redirects this reference to the cloned DOM. It achieves this goal by modifying the browser to tag each node in the host's DOM with a unique identifier (uid). During cloning, Transcript assigns each node in the cloned DOM the same uid as its counterpart in the host's DOM. When the guest attempts to access a DOM node, Transcript retrieves the uid of the node and walks the cloned DOM for a match. We defined a getElementByUID API on the document object to return a node with a given uid.

If the guest's operations conform to the host's policies, the host commits the transaction, upon which Transcript replaces the host's DOM with the transaction's copy of the DOM, thereby making the guest's speculative changes visible to the host.

### 5.3 Conflict detection

When a host decides to commit a transaction, Transcript will replace the host's DOM with the guest's DOM. Objects on the host's heap are also overwritten using the write set of the guest's transaction. During replacement, care must be taken to ensure that the host's state is consistent with the guest's state. Consider, for instance, a guest that performs an appendchild operation on a DOM node (say node N). This operation causes a new node to be added to the cloned DOM, and also suspends the guest transaction. However, the host may delete node N before resuming the transaction; upon resumption, the guest continues to update a stale copy of the DOM (*i.e.*, the cloned version). When the transaction commits, the removed DOM node will be added to the host's DOM.

Transcript adds the isDOMConflict and isHeapConflict APIs to the transaction object, which allow host developers to register conflict detection policies. When invoked in the host's iblock, the isDOMConflict API in-

(1)	<pre>function hasParent(txNode) {</pre>	
(2)	<pre>var parent = txNode.parentNode;</pre>	
(3)	<pre>if (document.getElementByUID(parent.uid) != null) return true;</pre>	
(4)	else return false;	
(5)	}	
(6)	<pre>var isAllowed = tx.isDOMConflict(hasParent); // tx is the transaction object</pre>	
Figure 5: Example showing conflict detection.		

vokes the conflict detection policy on each DOM node speculatively modified within the transaction (using the transaction's write set to identify nodes that were modified). The isHeapConflict API likewise checks that the state of the host's heap matches the state of the guest's heap at the start of the transaction. The snippet in Figure 5 shows one example of such a conflict detection policy (using isDOMConflict) encoded in the host's iblock that verifies that each node speculatively modified by the guest (txNode) has a parent in the host's DOM.

While Transcript provides the core *mechanisms* to detect transaction conflicts, it does not dictate any *policies* to resolve them. The host must resolve such conflicts within the application-specific part of its iblocks.

## 5.4 The <script> tag

The examples presented thus far show hosts including guest code by inlining it within a transaction. However, hosts typically include guests using <script> tags, *e.g.*, <script src="http://untrusted.com/guest.js>. Transcript also supports code inclusion using <script> tags. To do so, it extends the <script> tag so that the fetched code can be encapsulated in a function rather than run immediately. The host application can use the modified <script> tag as: <script src="http://untrusted.com/guest.js" func="foobar">>.

This tag encapsulates the code in foobar, which the host can then invoke within a transaction.

By itself, this modification unfortunately affects the scope chain in which the fetched code is executed. JavaScript code included using a <script> tag expects to be executed in the global scope of the host, but the modified <script> tag would put the fetched code in the scope of the function specified in the func attribute (*e.g.*, foobar).

We addressed this problem using a key property of eval. The ECMAScript standard [8, Section 10.4.2] specifies that an *indirect* eval (*i.e.*, via a reference to the eval function) is executed in the global scope. We therefore extracted the body of the compiled function foobar and executed it using an indirect eval call within a transaction (see Figure 7). This transformation allowed all variables and functions declared in the function foobar to be speculatively attached to the host's global scope.

## 6 Evaluation

We evaluated four aspects of Transcript. First, we studied the applicability of Transcript to confine real-world guests. In Section 6.1, we show how Transcript confined five guests, which varied in size from about 1,400 to

	Benchmark	Size (LoC)	<script></script>
--	-----------	------------	-------------------

Figure 6: Guest benchmarks. We used transactions to isolate each of these benchmarks from a simple hosting Web page.

(1)	<script func="menu" src="jsMenu.js"></script>
(2)	<script src="libTranscript.js>
(3)	<script>(function () {</th></tr><tr><th>(4)</th><th><pre>var to_commit = true, e = eval; // indirect eval</pre></th></tr><tr><th>(5)</th><th><pre>var tx = transaction { e(getFunctionBody(menu));}</pre></th></tr><tr><th>(6)</th><th><pre>do { <application-specific-policies></pre></th></tr><tr><th>(7)</th><th>processDOMEvents(tx);</th></tr><tr><th>(8)</th><th><pre>tx = tx.resume();</pre></th></tr><tr><th>(9)</th><th><pre>} while(tx.isSuspended());</pre></th></tr><tr><th>(10)</th><th>if(to_commit) tx.commit();</th></tr><tr><th>(11)</th><th><pre>})(); </script>

Figure 7: Confining JavaScript Menu. This figure illustrates several concepts: (a) lines 1 and 5 demonstrate the enhanced <script> tag and the host's use of indirect eval to include the guest, which is compiled into a function (called menu; line 1) (Section 5.4). getFunctionBody extracts the code of the function menu; (b) line 3 implements variable hiding (Section 3.2), making tx invisible to the guest; (c) our supporting library libTranscript (line 2) implements processDOMEvents (called on line 7). This function processes the guest's DOM operations that trap into the iblock.

7,500 lines of code. Second, we show in Section 6.2 that a host that uses Transcript can protect itself and recover gracefully from malicious and buggy guests. Third, we report a performance evaluation of Transcript in Section 6.3. Last, in Section 6.4, we study the complexity of writing policies for Transcript.

## 6.1 Case studies on guest benchmarks

To evaluate Transcript's applicability to confine realworld guests, we experimented with five JavaScript applications, shown in Figure 6. For each guest benchmark in Figure 6, we played the role of a host developer attempting to include the guest into the host, *i.e.*, we created a Web page and included the code of the guest into the page using <script> tags. Most of the guests were implemented in several files; the <script> column in Figure 6 shows the number of <script> tags that we had to use to include the guest into the host. We briefly describe the guest benchmarks and the domain-specific policies that were implemented for each iblock.

(1) JavaScript Menu is a standalone widget that implements pull-down menus. Figure 7 shows how we confined JavaScript Menu using Transcript. The iblock for JavaScript menu enforced a security policy that disallowed the guest from accessing the network (XMLHttpRequest) or domain cookies.

JavaScript Menu makes extensive use of document.write to build menus, with several of these calls used to register event handlers, as shown in the snippet in Figure 8 (event handler registrations are shown in bold font). Each document.write call causes the transaction to suspend and pass control to the iblock. The iblock uses document.parse to (a) parse the arguments to identify the HTML element(s) being created; (b) identify whether

(1)	<pre>document.write("&lt;" + blInfo.divType + ' id="' + mName);</pre>
(2)	<pre>document.write('" class="' + mClass + '" ');</pre>
(3)	<pre>document.write(setStyle(blInfo.divType, blInfo.fontSize, , "inherit"));</pre>
(4)	<pre>document.write(' onclick="' + mAction + '" ');</pre>
(5)	<pre> // more code elided</pre>
(6)	<pre>document.write('onmouseover="popMenu('+mLevel+",'"+mName+"','"+mPopup+"'"+');" ');</pre>
(7)	<pre>document.write('onmouseout="setColorPassive(' + "'" + mName + "'" + ');"&gt;');</pre>

Figure 8: Code from JavaScript Menu.

any event handlers are being registered and wrap them; and (c) write the resulting HTML into the transaction's speculative DOM.

(2) *Picture Puzzle* uses the drag-and-drop features provided by the AJS JavaScript library [2] to build an application that prompts the user to arrange jumbled pieces of a picture within a  $3 \times 3$  grid (we adapted this benchmark from [38]). We ran the benchmark within a transaction and enforced a domain-specific security policy that prevented the transaction from committing its changes if it attempted to install a handler to capture the user's keystrokes (*e.g.*, any event with onkey as a substring).

(3) *GoogieSpell* extends the AJS library to provide a spell-checking service. When a user clicks the "check spelling" button, GoogieSpell sends an XMLHttpRequest to a third-party server to fetch suggestions for misspelled words. We created a transactional version of GoogieSpell, whose iblock implemented a domain-specific policy that prevents an XMLHttpRequest once the benchmark has read domain cookies or if the target URL of XMLHttpRequest does not appear on a whitelist.<sup>2</sup>

(4) *GreyBox* is content-display application that also extends the AJS library. It can be used to display external pages, build image galleries, receive file uploads and even show video or Flash content. The application creates an <iframe> to load new content. Our transactional version of the GreyBox application encoded a domain-specific iblock policy that only allowed the creation of <iframe>s to whitelisted URLs.

(5) Color Picker builds upon the popular jQuery library [4] and lets an end-user pick a color by moving sliders depicting the intensities of red, blue and green. We executed the entire benchmark (including all the supporting jQuery libraries) as a transaction and encoded an iblock that disallowed modifications to the innerHTML property of arbitrary <div> nodes.

However, for this guest, it turns out that an iblock that disallows any changes to the sensitive innerHTML property of *any* <div> element is overly restrictive. This is because Color Picker modified the innerHTML property of a <div> element that it created. We therefore loosened our policy into a history-based policy that let the benchmark change innerHTML properties of <div> elements that it created. The iblock determines whether a <div> element was created by the transaction by querying its write set. The

<sup>2</sup>Such *cross-origin resource sharing* permits cross-site XMLHttpRequests, and is supported by Firefox-3.5 and higher [35].

relevant snippet from the iblock is shown below; the tx variable denotes the transaction:

```
    var ws = tx.getWriteSet(); ...
    if (tx.getGause().match("innerHTML") && ws.checkMembership(tx.getObject(), "*") && (tx.getObject() instanceof HTMLBodyElement))
    // perform action on behalf of untrusted code
```

## 6.2 Fault injection and recovery

To evaluate how Transcript can help hosts detect and recover from attacks, we performed a set of fault-injection experiments on a real Web application that allows integration of untrusted guest code. We used the Bigace Web content management system [3] running on our Web server as the host, and created a Web site that mashed content from Bigace with content provided by untrusted guests (each guest was included into the mashup using the <script> tag). We wrote guests that emulated known attacks and studied host behavior when the host (1) directly included the guest in its protection domain; and (2) used Transcript to isolate the guest.

Our experiments show that with appropriate iblock policies, speculative execution ensured clean recovery; neither the JavaScript heap nor the DOM of the host was affected by the misbehaving guest.

(1) *Misplaced event handler*. JavaScript provides a preventDefault method that suppresses the default action normally taken by the browser as a result of the event. For example, the default action on clicking a link is to fetch the page corresponding to the URL referenced in the link. Several sites use preventDefault to encode domain-specific actions instead, *e.g.*, displaying a popup when a link is clicked.

In this experiment, we created a buggy guest that displays an advertisement within a <div> element. This guest mistakenly registers an onClick event handler that uses preventDefault with the document object instead of with the <div> element. The result of including this guest directly into the host's protection domain is that all hyperlinks on the Web page are rendered unresponsive. We then modified the host to isolate the guest using a policy that disallows a transaction to commit if it attempts to register an onClick handler with the document object. This prevented the advertisement from being displayed, *i.e.*, the <div> element containing the misbehaving guest was not even created, but otherwise allowed the host to function correctly. JavaScript reference monitors proposed in prior work can prevent the registration of the onClick handler, but leave the div element of the misbehaving guest on the host's Web page.

(2) *Prototype hijacking*. We implemented a prototype hijacking attack by writing a guest that set the Array.prototype.slice function to null. To illustrate the ill-effects of this attack, we modified the host to include two popular (and benign) widgets, namely Twitter [7] and AddThis [1], in addition to the malicious guest. The prototype hijacking attack prevented both the benign



Figure 9: Performance of guest benchmarks. This chart compares the time to load the unmodified version of each guest benchmark against the time to load the transactional version in the two variants of Transcript. On average, Transcript (JS only) increases the load time by just 0.11s, while Transcript (full) increases the load time by 0.16s.

#### widgets from functioning properly.

However, when the malicious guest is enclosed within a transaction whose iblock prevents a commit if it detects prototype hijacking attacks, the host and both benign widgets worked normally. We further inspected the transaction's write set and verified that none of the heap operations attributed to the malicious guest were actually applied to the host. Although traditional JavaScript reference monitors can detect and prevent prototype hijacking attacks by blocking further <script> execution, they do not allow the hosts to cleanly recover from all heap changes.

(3) Oversized advertisement. We created a guest that displayed an interactive JavaScript advertisement within a <div> element. In an unprotected host, this advertisement expands to occupy the full screen on a mouseover event, *i.e.*, the guest registered a misbehaving event-handler that modifies the size of the <div>. We modified the host to isolate this guest using a transaction and an iblock that prevents a commit if the size of the <div> element increased beyond a pre-specified limit. With this policy, we observed that the host could successfully prevent the undesired <div> modification by discarding the speculative DOM and JavaScript heap changes made by the event handler executing within the transaction.

#### 6.3 Performance

We measured the overhead imposed by Transcript both using guest benchmarks, to estimate the overall cost of using transactions, and with microbenchmarks, to understand the impact on specific JavaScript operations. All experiments were performed with Firefox v3.7a4pre on a 2.33Ghz Intel Core2 Duo machine with 3GB RAM and running Ubuntu 7.10. We turned off JavaScript JIT compilation support in the browser during these experiments.

#### 6.3.1 Guest benchmarks

To evaluate the overall performance impact of Transcript, we measured the increase in the load time of each guest benchmark. Recall that each benchmark is included in the Web page using a set of <script> tags; the version that uses Transcript executes the corresponding JavaScript code within a single transaction using modified <script> tags. The onload event fires at the end of the document loading process, *i.e.*, when all scripts have completed execution. We therefore measured the time elapsed from the moment the page is loaded in the browser to the firing of the onload event.

To separately assess the impact of speculatively executing JavaScript and DOM operations, each experiment involved executing the benchmarks on two separate variants of Transcript, namely Transcript (full) which supports both speculative DOM and JavaScript operations and Transcript (JS only) which only supports speculative JavaScript operations (and therefore does not isolate DOM operations of the guest). Figure 9 presents the results averaged over 25 runs of this experiment. On average, Transcript (JS only) increased load time by just 0.11 seconds while Transcript (full) increased the load time by 0.16 seconds. These overheads are typically imperceptible to end users. Only the Color Picker benchmark had above-average overheads. This was because (a) the guest heavily interacted with the DOM, causing frequent suspension of its transaction; and (b) the guest had several Array operations that referenced the length of the array. Each such operation triggered a traversal of read/write sets to calculate the array length.

Note that Transcript only degrades performance of JavaScript code executing within transactions (*i.e.*, guests). The performance of code executing outside transactions (*i.e.*, hosts) is not affected by our prototype.

#### 6.3.2 Microbenchmarks

We further dissected the performance of Transcript using microbenchmarks designed to stress specific functionalities. We used three sets of microbenchmarks: function calls, event dispatchers, and the Sunspider suite. In our experiments, we executed each microbenchmark within a transaction whose iblock simply permitted all actions and resumed the transaction without enforcing additional security policies, and compared its performance against the non-transactional version.

**Function calls.** We devised a set of microbenchmarks (Figure 10) that stress the performance of Transcript's function call-handling code. Each benchmark invoked the code in the first column of Figure 10 10,000 times.

Recall that Transcript suspends on function calls that cause external operations and for certain native function calls, such as eval. Each suspend operation requires Transcript to save the state of the transaction, execute the iblock, and restore the transaction state upon the execution of a resume call. Most of the benchmarks in Figure 10 trigger a suspension, which induces significant overheads. In particular, addEventListener had an over-

Microbenchmark	Overhead				
Native Functions					
(1) eval("1")	6.69×				
<pre>(2) eval("if (true)true;false")</pre>	6.87×				
(3) fn.call(this, i)	1.89×				
External operations					
<pre>(4) getElementById("checkbox")</pre>	6.78×				
(5) getElementsByTagName("input")	6.89×				
<pre>(6) createElement("div")</pre>	3.69×				
(7) createEvent("MouseEvents")	3.82×				
(8) addEventListener("click", clk, false)	26.51×				
<pre>(9) dispatchEvent(evt)</pre>	1.20×				
<pre>(10) document.write("<b>Hi</b>")</pre>	1.26×				
<pre>(11) document.write("<script>x=1;</script>")</pre>	2.01×				

Figure 10: Performance of function call microbenchmarks. Note: The call to document.write on line 11 invokes gluing of the <script>'s execution context (Section 3.1.1) when it suspends, while call to document.write on line 10 only suspends.

	Overhead		
Event name	Normalized	Raw Delay (µs)	
(1) Drag Event (drag)	1.71×	97	
(2) Keyboard Event (keypress)	1.16×	150	
(3) Message Event (message)	1.17×	85	
<li>(4) Mouse Event (click)</li>	1.54×	86	
(5) Mouse Event (mouseover)	2.05×	88	
(6) Mutation Event (DOMAttrModified)	2.14×	88	
(7) UI Event (overflow)	1.97×	61	

Figure 11: Performance of event dispatch microbenchmarks.

head of  $26.51 \times$ . The bulk of the overhead was induced by code in the iblock that generates wrappers for the event handler registered using addEventListener.

**User events.** A JavaScript application executing within a transaction may dispatch user events, such as mouse clicks and key presses, which must be processed by the event handler associated with the relevant DOM node. The promptness with which events are dispatched typically affects end-user experience.

To measure the impact of transactions on this aspect of browser performance, we devised a set of microbenchmarks that dispatched events and measured the delay in handling them (Figure 11). Each microbenchmark used JavaScript code to simulate user actions, such as clicking on a checkbox, hovering and moving the mouse, pressing keys, and events such as drag, message, DOMAttrModified and overflow. In each case, code that generated and dispatched the event executed as a transaction with an iblock that allowed all actions. To measure performance, we executed this code 1,000 times and compared its performance against a native event dispatcher.

Figure 11 presents the results, which show the normalized overhead as well as the raw delay to process a single event. As this figure shows, although the normalized overheads range from 16% to 114%, the raw delays average about 94 microseconds, which is imperceptible to end users.

**SunSpider.** Finally, we also tested Transcript with the SunSpider JavaScript benchmark suite by executing each of its benchmarks within a transaction. This benchmark suite reported an average overhead of  $3.94 \times$  across all benchmarks. In particular, we observed high overheads for benchmarks that had tight loops operating over many array elements. The overhead primarily stems from hav-

Policy	T-LOC	C-LOC	Policy	T-LOC	C-LOC
Conscript-#1	7	2	Conscript-#2	5	6
Conscript-#3	6	3	Conscript-#4	9	7
Conscript-#5	9	9	Conscript-#6	5	8
Conscript-#7	7	5	Conscript-#8	5	6
Conscript-#10	9	16	Conscript-#11	12	17
Conscript-#12	5	4	Conscript-#13	4	6
Conscript-#14	3	5	Conscript-#15	6	7
Conscript-#16	6	4	Conscript-#17	7	5

Figure 12: Policy complexity. Comparing policies in Transcript (T-LOC) and Conscript (C-LOC). Policies are numbered as in the Conscript paper [32]. We omitted Conscript#9 because it is IE-specific.

ing to consult the write set for every read operation and updating the read set itself even though the iblock's permissive security policy did not consult read/write sets.

## 6.4 Complexity of policies

To study the complexity of writing security policies in Transcript, we compared the number of lines of code needed to write policies in Transcript and in Conscript [32], the work most closely related to ours. We considered the policies discussed in the Conscript paper and wrote equivalent policies in Transcript; Figure 12 compares the source lines of code (counting number of semi-colons) of policies in Transcript and Conscript. This figure shows that the programming effort required to encode policies in both systems is comparable.

#### 7 Related Work

This paper builds upon the idea of extending JavaScript with transactions, which was proposed in a recent position paper [14]. While that paper focused on the semantics of the extended language, this paper is the first to report the design and implementation of a complete speculative execution system for JavaScript.

There is much prior work in the broad area of isolating untrusted guests. Transcript is unique because it allows hosts to recover cleanly and easily from the effects of malicious or buggy guests (Figure 13). In exchange for requiring no modification to the guest, Transcript requires modifications both to the host (*i.e.*, the server side) and to the browser (*i.e.*, the client side) to enhance the JavaScript language.

Language restriction. Several projects have defined subsets of JavaScript that omit dynamic constructs, such as eval, with and this, to make JavaScript amenable to static code inspection [13, 18, 20, 34]. However, safe subsets of JavaScript are non-trivial to design [19, 27, 29, 30], and also restrict code developers from using arbitrary constructs of the language in their applications. Transcript places no such restrictions on guest code.

**Static analysis.** Despite the dynamic nature of JavaScript, there have been a few efforts at statically analyzing JavaScript code. Gatekeeper [20] presents a static analysis to validate widgets written in a subset of JavaScript. It does so by matching widget source code against a database of patterns denoting unsafe programming practices. Guha *et al.* [21] developed static tech-

		Unrestricted	Unmodified	Policy
System	Recovery	guest	browser	coverage
Transcript			X	Heap + DOM
Conscript [32]	X	X	X	Heap + DOM
AdJail [25]	X	1	1	DOM <sup>(1)</sup>
Caja [34]	X	X	1	Heap + DOM
Wrappers [28, 29, 31]	X	✓ <sup>(2)</sup>	1	Heap + DOM
Info. flow [12]	X	1	1	Heap
IRMs [40, 42, 47]	X	1	1	Heap + DOM
Subsetting [13, 18, 29]	X	×	1	Static policies <sup>(3)</sup>

Figure 13: Techniques to confine untrusted guests. (1) Adjail executes guests in a separate <i frame> from the host and disallows guests from executing in the host's context. (2) Some wrapper-based solutions [28] do restrict JavaScript constructs allowed in guests. (3) Subsetting is a static technique and its policies are not enforced at runtime.

niques to improve AJAX security. Their work uses static analysis to enhance a server-side proxy with models of AJAX computation on the client. The proxy then ensures that AJAX requests from the client conform to these models.

Chugh *et al.* [12] developed a staged information-flow tracking framework for JavaScript to protect hosts from untrusted guests. Its static analysis identifies constraints on host variables that can be read or written by guests. It validates these constraints on code loaded at runtime via eval or <script> tags, and rejects it if it violates these constraints. Unlike Transcript, which tracks changes to both the heap and DOM, Chugh *et al.*'s work only tracks changes to the heap.

**Object capabilities, wrappers, and code rewriting.** Object capability and wrapper-based solutions (*e.g.*, [28, 29, 31]) create wrapped versions of JavaScript objects to be protected, and ensure that they can only be accessed by code that has the capabilities to do so. In contrast to these techniques, which provide isolation by requiring the host's objects to be wrapped, Transcript wraps guest code using transactions, and mediates its actions with the host via iblocks. The research literature has also developed solutions to inline runtime checks into untrusted guests. These techniques include BrowserShield [42], CoreScript [47], and the work of Phung *et al.* [40]. Unlike these works, Transcript simply wraps untrusted code in a transaction, and does not modify it. These works also do not explicitly address recovery.

Aspect-oriented policy enforcement. Aspectoriented programming (AOP) techniques have previously been used to enforce cross-cutting security policies [9, 15, 17]. Among the AOP-based frameworks for JavaScript [22, 32], our work is most closely related to Conscript [32], which uses runtime aspect-weaving to enforce policies on untrusted guests. Both Conscript and Transcript require changes to the browser to support their policy enforcement mechanisms. However, unlike Transcript, Conscript does not address recovery from malicious guests, and also requires guests to be written in a subset of JavaScript. While recovery may also be possible in hosts that use Conscript, the hosts would have to encode these recovery policies explicitly. In contrast, hosts that use Transcript can simply discard the speculative changes made by a policy-violating guest.

**Browser-based sandboxing.** Both BEEP [24] and MashupOS [45] enhance the browser with new HTML constructs. BEEP's constructs allow the browser to detect script-injection attacks, while MashupOS provides sandboxing constructs to improve the security of client-side mashups. While Transcript requires modified <script> tags as well, it provides the ability to speculatively execute and observe the actions of untrusted code, which neither BEEP nor MashupOS provide.

AdJail aims to protect hosts from malicious advertisements [25]. It confines advertisements by executing them in a separate <iframe>, and uses postMessage to allow the <iframe> to communicate with the host. Hosts use access control policies to determine the set of DOM modifications allowed by an advertisement. AdJail is effective at confining advertisements, which cannot affect the host's heap. However, it is unclear whether this approach will work well in scenarios where hosts and guests need to interact extensively, *e.g.*, in the case where the guest is a library that the host wishes to use.

Blueprint [26] and Virtual Browser [11] confine guests by setting up a virtual environment for their execution. This environment is itself written in JavaScript and parses HTML and script content, thereby mediating the execution of guests on unmodified browsers. However, unlike Transcript, they do not address recovery.

Using transactions for security. Transactions and speculative execution mechanisms have previously been used to improve software security and reliability (e.g., [10, 23, 41, 43]) and Web browser performance [33]. The work most closely related to Transcript is the one by Sun et al. [44] on one-way isolation. This work describes a sandboxing mechanism that allows isolated execution of untrusted code. As in Transcript, code within the sandbox cannot modify the state of code outside, but the reverse is possible. However, their work focused on implementing such a sandbox at the granularity of OS-level artifacts, such as processes and files. In contrast, Transcript applies a similar approach to the problem of isolating JavaScript code. Our work therefore required several new contributions, such as the ability to suspend/resume transactions and wrap event handlers.

## 8 Conclusion

Our research shows that extending JavaScript with support for transactions allows hosting Web applications to speculatively execute and enforce security policies on untrusted guests. Speculative execution allows hosts to cleanly and easily recover from the effects of malicious and misbehaving guests. In building Transcript, we made several contributions, including suspend/resume for JavaScript, support for speculative DOM updates, and novel strategies to implement transactions in commodity JavaScript interpreters.

### References

- [1] Addthis. http://www.addthis.com/.
- [2] AJS: The ultra lightweight JavaScript library. http://orangoo. com/labs/AJS/.
- [3] BIGACE web content management system. http://www.bigace. de/.
- [4] jQuery: The write less, do more, JavaScript library. http://jquery. com.
- [5] Jquery UI slider plugin. http://jqueryui.com/demos/slider.
- [6] JavaScript widgets/menu. http://jswidgets.sourceforge.net.
- [7] Twitter/profile widget. http://twitter.com/about/resources/ widgets/widget\_profile.
- [8] ECMAScript language spec., ECMA-262, 5th edition, Dec 2009.
- [9] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In ACM PLDI, 2005.
- [10] A. Birgisson, M. Dhawan, U. Erlingsson, V. Ganapathy, and L. Iftode. Enforcing authorization policies using transactional memory introspection. In ACM CCS, 2008.
- [11] Y. Cao, Z. Li, V. Rastogi, and Y. Chen. Virtual browser: a Weblevel sandbox to secure third-party JavaScript without sacrificing functionality (poster). In ACM CCS, 2010.
- [12] R. Chugh, J. Meister, R. Jhala, and S. Lerner. Staged information flow in JavaScript. In ACM SIGPLAN PLDI, 2009.
- [13] D. Crockford. ADsafe Making JavaScript safe for advertising. http://adsafe.org.
- [14] M. Dhawan, C.-C Shan, and V. Ganapathy. Position paper: The case for JavaScript transactions. In 5<sup>th</sup> ACM SIGPLAN PLAS Workshop, June 2010.
- [15] Ú. Erlingsson. The Inlined Reference Monitor Approach to Security Policy Enforcement. PhD thesis, Cornell University, 2004.
- [16] U. Erlingsson, B. Livshits, and Y. Xie. End-to-end Web application security. In *HotOS*, 2007.
- [17] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE S&P*, 1999.
- [18] Facebook. FBJS Facebook developerwiki. 2007.
- [19] M. Finifter, J. Weinberger, and A. Barth. Preventing capability leaks in secure JavaScript subsets. In NDSS, 2010.
- [20] S. Guarnieri and B. Livshits. GateKeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In USENIX Security, 2009.
- [21] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In WWW, 2009.
- [22] H. Washizaki et al.. AOJS: Aspect-oriented JavaScript programming framework for Web development. In Intl. wkshp. Aspects, components, and patterns for infrastructure software, 2009.
- [23] S. Jana, V. Shmatikov, and D. Porter. TxBox: Building secure, efficient sandboxes with system transactions. In *IEEE S&P*, 2011.
- [24] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In WWW, 2007.
- [25] M. T. Louw, K. T. Ganesh, and V.N. Venkatakrishnan. Adjail: Practical enforcement of confidentiality and integrity policies on Web advertisements. In USENIX Security Symposium, 2010.
- [26] M. Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *IEEE S&P*, 2009.
- [27] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In APLAS, October 2008.
- [28] S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with filters, rewriting and wrappers. In *ESORICS*, 2009.
- [29] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted Web applications. In *IEEE S&P*, 2010.
- [30] S. Maffeis and A. Taly. Language based isolation of untrusted JavaScript. In *IEEE CSF*, 2009.
- [31] L. Meyerovich, A. Porter Felt, and M. S. Miller. Object views: Fine-grained sharing in browsers. In WWW, 2010.
- [32] L. Meyerovich and B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE S&P*, 2010.

- [33] J. Mickens, J. Elson, J. Howell, and J. Lorch. Crom: Faster Web browsing using speculative execution. In NSDI, 2010.
- [34] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. 2008. Manuscript.
- [35] Mozilla Developer Center. HTTP access control. http:// developer.mozilla.org/En/HTTP\_access\_control.
- [36] Orangoo-Labs. GoogieSpell: Gmail like spell checker for your own application. http://orangoo.com/labs/GoogieSpell.
- [37] Orangoo-Labs. GreyBox: A popup window that doesn't suck. http://orangoo.com/labs/GreyBox.
- [38] Orangoo-Labs. Sortable list widget. http://orangoo.com/AJS/ examples/sortable\_list.html.
- [39] S. Di Paola and G. Fedon. Subverting Ajax: Next generation vulnerabilities in 2.0 Web applications. In 23rd Chaos Communication Congress, 2006.
- [40] P. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting JavaScript. In ASIACCS, 2009.
- [41] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating systems transactions. In ACM SOSP, 2009.
- [42] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic HTML. ACM Trans. Web, 1(3):11, 2007.
- [43] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In OSDI, 1996.
- [44] W. Sun, Z. Liang, R. Sekar, and V. N. Venkatakrishnan. Oneway isolation: An effective approach for realizing safe execution environments. In NDSS, 2005.
- [45] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. In *ACM SOSP*, 2007.
- [46] WWW-Consortium. Document object model events, Nov 2000. http://www.w3.org/TR/DOM-Level-2-Events/events.html.
- [47] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In ACM POPL, 2007.

All URLs were last accessed on February 26, 2011.

## A Event-handler Wrapper Generators

This section describes the implementation of the event handler wrapper generator, which is included in the mandatory part of a host's iblock. Firefox currently supports three event-handling models [46]. For each model, the goal of the wrapper generator is to obtain a reference to the handler being registered, and wrap it suitably. We describe the three models briefly and discuss how Transcript obtains a reference to the handler in each case.

(1) The *DOM-level 0/Traditional* model registers an event handler for a DOM node, *e.g.*, a <div> element, as follows: node.onclick = clkhandler. Here node represents the <div> element and clkhandler is registered as a handler for onclick events. Transcript modifies the interpreter to suspend transactions that change properties containing event handlers, such as onclick and onload. Once the transaction suspends, the iblock obtains a reference to clkhandler.

(2) The *DOM-level 0/Inline* model registers an event handler using code such as: document.write("<div onclick="/\*handler code\*/>"), which sets an attribute (*e.g.*, onclick) of the DOM node. Transcript handles such cases by suspending the execution of document.write.

(1)	<pre>tx_clkhandler = function(evt) {</pre>
(2)	<pre>evttx = transaction { node.evtH (evt); } iblock_func (evttx);</pre>
(3)	}

Figure 14: Generating wrapped event handlers.

The argument to this call is HTML code, which the transaction's iblock parses to obtain a reference to the event handler.

(3) The *DOM-level* 2 model registers an event handler as follows: node.addEventListener("click", clkhandler, false). Transcript suspends the execution of addEventListener, thereby allowing the iblock to obtain a reference to clkhandler.

In addition to obtaining a reference to the event handler, the iblock also obtains a reference to node, which is the DOM node for which the handler was being registered. The iblock then initializes a new property node.evtH with clkhandler, and defines a new function tx\_clkhandler as shown in Figure 14, which it registers as the event handler. Here, iblock\_func is a function that contains the iblock itself, while evt is a JavaScript object that the browser uses to denote the event. As a result of this transformation, tx\_clkhandler is invoked when the onclick event is triggered, which then executes clkhandler within a transaction, thereby allowing Transcript to mediate its operation as well.

#### **B** Non-tail-recursive Interpreters

A key challenge in enhancing a legacy JavaScript interpreter, such as SpiderMonkey, with support for transactions is in how the interpreter uses recursion. To support the suspend/resume mechanism for switching control flow between a transaction and its iblock, the interpreter must not accumulate any activation records in its native stack (*e.g.*, the C++ stack, for SpiderMonkey) between when a transaction starts and when it suspends. In particular, the interpreter must not represent JavaScript function calls by C++ function calls. The same issue also arises when a compiler or JIT interpreter is used to turn JavaScript code into machine code.

To illustrate this point, consider SpiderMonkey, which implements the bytecode interpreter in C++. The main entry point to the bytecode interpreter is the C++ function JS\_interpret, which maintains the JavaScript stack as a linked list of activation records, each of which is a C++ structure. When one function calls another in JavaScript, the JS\_interpret function does not call itself in C++; instead, it adds a new activation record to the front of the linked list and continues with the same bytecode interpreter loop as before. Similarly, when a function returns to another in JavaScript, JS\_interpret does not return in C++; instead, it removes an old activation record from the front of the linked list and continues with the same bytecode interpreter loop as before. For the



(b) When the main JavaScript program starts the transaction, the C++ function JS\_interpret grows the JavaScript stack but does not call itself, so the native stack does not grow.



(c) When the transaction suspends, the interpreter removes activation records from the front of the JavaScript stack, up to and including the (youngest) transaction delimiter.







Figure 15: Native versus JavaScript call stacks. The implementation of suspend/resume relies on an interpreter without non-tail recursion.

most part, SpiderMonkey does not represent JavaScript calls by C++ calls.

The fact that SpiderMonkey does not represent JavaScript calls by native calls helps us add transactions to it without making invasive changes, as the following example illustrates. Suppose a transaction invokes a function f that suspends for some reason, *e.g.*, in Figure 15(a), the function f calls appendChild. If the C++ call to JS\_interpret that executes the transaction were not same as the one that executes the called function f, then

the former, although older, would have to return before the latter returns. As detailed in Figure 15, the former has to return when suspending the transaction, whereas the latter has to return when resuming the transaction. Even exception handling in C++ does not allow such control flow.

Unfortunately, JS\_interpret in SpiderMonkey does call itself in a few situations. For example, it handles JavaScript's eval in this way, and the problem of the C++ stack in Figure 15(f) does arise if we replace the document.body.appendChild(...) of Figure 15(a) by eval ("document.body.appendChild(...)"). One way to solve this problem requires applying the continuation-passing-style transformation to the interpreter to put it into tail form, *i.e.*, convert all recursive calls to JS\_interpret to tail calls. However, this transformation is invasive, especially if done manually on legacy interpreters.

Transcript uses a less invasive mechanism to enable suspend/resume in SpiderMonkey. This mechanism is similar in functionality to gluing (see Section 3.1.1), and we explain it with an example. Consider the eval construct, whose functionality is to parse its input string, compile it into bytecode, and then execute the bytecode as usual. Because only the last step, *i.e.*, that of executing the bytecode, can suspend, we simply changed the behavior of eval so that, if invoked inside a transaction, it suspends the transaction right away. The iblock of the transaction can then compile the string into bytecode and include the bytecode into the execution of the transaction. This is achieved by adding a new activation record to the front of the transaction's JavaScript stack and modifying the program counter to execute this code when the transaction resumes. When the suspended transaction resumes, it transfers control to the evaled code, which can freely suspend. Besides eval, our current Transcript prototype also implements gluing for document.write (as discussed in Section 3.1.1) and JavaScript builtins call and apply, which make non-tail recursive calls to JS\_interpret.