Enhancing JavaScript with Transactions

Mohan Dhawan[1], Chung-chieh Shan[2], and Vinod Ganapathy[1]

[1] Rutgers University
{mdhawan,vinodg}@cs.rutgers.edu
[2] University of Tsukuba
ccshan@post.harvard.edu

Abstract. Transcript is a system that enhances JavaScript with support for transactions. Hosting Web applications can use transactions 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 understand the behavior of untrusted guests, mediate their actions 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

It is now common for Web applications (*host*) to include untrusted, third-party JavaScript code (*guest*) of arbitrary provenance in the form of advertisements, libraries and widgets. Despite advances in language and browser technology, JavaScript still lacks mechanisms that enable Web application developers to debug and understand the behavior of third-party guest code. Using existing reflection techniques in JavaScript, the host cannot attribute changes in the JavaScript heap and the DOM to specific guests. Further, fine-grained context about a guest's interaction with the host's DOM and network is not supported. For example, the host cannot inspect the behavior of guest code under specific cookie values or decide whether to allow network requests by the guests.

This paper proposes to enhance the JavaScript language with builtin support for introspection of third-party guest code. The main idea is to extend JavaScript with a new `transaction` construct, within which hosts can speculatively execute guest code containing arbitrary JavaScript constructs. In addition to enforcing security policies on guests, a `transaction` would allow hosts to *cleanly recover from policy-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

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 383–408, 2012.
© Springer-Verlag Berlin Heidelberg 2012

```
1  <script type="text/javascript">
2  var editor = new Editor(); initialize(editor);
3  var builtins = [], tocommit = true;
4  for(var prop in Editor.prototype) builtins[prop] = prop;
5  var tx = transaction { [Guest code: Lines 6–9]
6      Editor.prototype.getKeywords = function(content) {...}
       ...
7      var elem = document.getElementById("editBox");
8      elem.addEventListener("mouseover", displayAds, false);
       ...
9      document.write('<div style="opacity:0.0; z-index:0; ... size/loc params">
          <a href="http://evil.com"> Evil Link </a></div>');
10 };
11 tocommit = gotoIblock(tx); [Implements the host's security policies]
12 if (tocommit) tx.commit();
13 ... /* rest of the Host Web application's code */
14 </script>
```

Fig. 1. Motivating example. This example shows how a host can mediate an untrusted guest (lines 6–9). The introspection block (invoked in line 11) enforces the host's security policies (see Figure 2) on the actions performed by the guest.

explicitly. The introspection mechanism (`transaction`) is built within the JavaScript language itself, thereby allowing guest code to contain arbitrary JavaScript constructs (unlike contemporary techniques [13,18,36,29,31]).

Let us consider an example of a Web-based word processor that hosts a third-party 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-style attack by overlaying the editor with transparent HTML elements pointing to malicious sites.

Traditional reference monitors [16], which mediate the action of guest code as it executes, 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 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 hosts to introspect all actions of untrusted guest code. In our example, the host speculatively executes the untrusted widget by enclosing it in 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 the Transcript system, that has the following 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 [13,18,36,29,31] (*e.g.,* `eval`, `this` and `with`).

In the rest of the paper, we discuss the design, implementation and evaluation of Transcript.

2 Overview of Transcript

Transcript enables hosts to understand the behavior of untrusted guests, detect attacks by malicious guests and recover from them, and perform forensic analysis. We briefly discuss Transcript's utility and then provide an overview of its functionality for confining a malicious guest.

(1) Understanding Guest Code. Analysis of third-party JavaScript code is often hard due to code obfuscation. Using Transcript, a host can set watchpoints on objects of interest. Coupled with suspend/resume, it is possible to perform a fine grained debug analysis by inspecting the read/write sets on every guest initiated object read/write and method invocation. Transcript's speculative execution provides an ideal platform for concolic unit testing [44,20] of guests. For example, using Transcript, a host can test a guest's behavior under different values of domain cookies.

(2) Confining Malicious Guests. Transcript's speculative execution permits buffering of network I/O and writing to a speculative DOM, thereby allowing unprecedented flexibility in confining untrusted guest code. For example, to prevent clickjacking-style attacks, the host can simply discard guest's modifications to the speculative DOM.

(3) Forensic Analysis. Since Transcript suspends on external and user-defined operations, the suspend/resume mechanism is an effective tool for forensic analysis of a suspected vulnerability exploited by the guest. For example, code-injection attacks using DOM or host APIs [4] can be analyzed by observing the sequence of suspend calls and their arguments.

Transcript in Action. We illustrate Transcript's ability to confine untrusted guests 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-style 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 can protect itself from attacks by defining and enforcing a suitable set of security policies. These may include policies to prevent prototype hijacks [41], clickjacking-style 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 within a `transaction` construct (line 5, Figure 1) and specifies its security policy (lines D–O, Figure 2). 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, Figure 1) 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 in a transaction to the cloned DOM. Thus, references to `document` within the guest resolve to the cloned DOM.

```
A  do { [Function gotoIblock implements the host's introspection block: Lines A–R]
B     var arg = tx.getArgs();   var obj = tx.getObject();
C     var rs = tx.getReadSet(); var ws = tx.getWriteSet();
D     for(var i in builtins) {
E        if (ws.checkMembership(Editor.prototype, builtins[i])) tocommit = false;
F     } ... /* definition of 'IsClickJacked' to go here */
G     if (IsClickJacked(tx.getTxDocument())) tocommit = false;
H     ... /* more policy checks go here */ [inlined code from libTranscript: Lines I–O]
I     switch(tx.getCause()) {
J        case "addEventListener":
K           var txHandler = MakeTxHandler(arg[1]);
L           obj.addEventListener(arg[0], txHandler, arg[2]); break;
M        case "write": WriteToTxDOM(obj, arg[0]); break; ... /* more cases */
N        default: break;
O     };
P     tx = tx.resume();
Q  } while(tx.isSuspended());
R  return tocommit;
```

Fig. 2. An iblock. An iblock consists of two parts: a host-specific part, which encodes the host's policies to confine the guest (lines D–H), and a mandatory part, which contains functionality that is generic to all hosts (lines I–O).

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 the host 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 2, 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 contains 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`. Depending on its 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 2 has two logical parts: a *host-specific part* that encodes host's policies (lines D–H), and a *mandatory part* that performs operations on behalf of suspended guests (lines I–O). The iblock in Figure 2 illustrates two policies:

(**1**) Lines D–E 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.

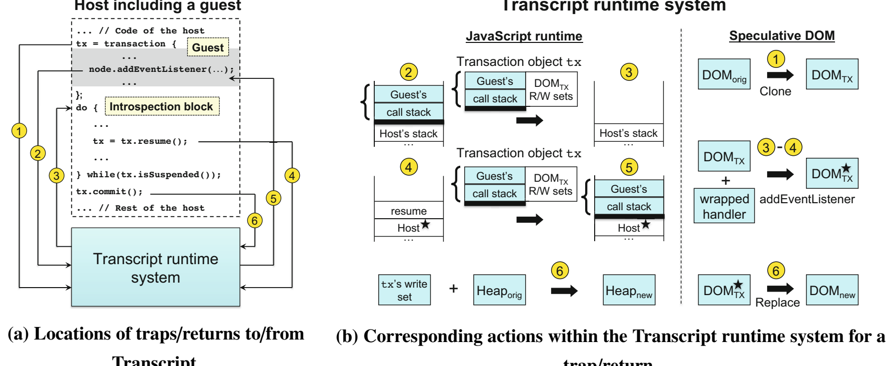
(**2**) Line G detects clickjacking-style attempts by checking the DOM for the presence of any transparent HTML elements introduced by the guest. (The body of `IsClickJacked`, which implements the check, is omitted for brevity).

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 J–L in Figure 2 create and attach an event handler to the cloned DOM when the guest suspends on line 8 in Figure 1. 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 M in Figure 2 speculatively executes the DOM modifications requested when the guest suspends on line 9 in Figure 1. 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,



(a) Locations of traps/returns to/from Transcript.

(b) Corresponding actions within the Transcript runtime system for a trap/return.

Fig. 3. Workflow of a Transcript-enhanced host. Part (a) of the figure shows a host enclosing a guest within a transaction and an inlined introspection block, while part (b) shows the JavaScript runtime and the DOM subsystem. The labels ①-⑥ in the figure show: ① the host's DOM being cloned at the start of the transaction, ② the host's call stack before a call that suspends the transaction, ③ the call stack after suspension, ④ the host's call stack when the transaction is about to resume; the speculative DOM has been updated with the requested changes, ⑤ the host's call stack just after resumption, ⑥ 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.

when the host 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 3, 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 included an untrusted 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 3). 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 delimiter* 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 ② and ③ in Figure 3).

The iblock logically extends from the end of the transaction to the last `resume` or `commit` call on the transaction object (*e.g.,* lines A–R in Figure 2). 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 ④ and ⑤). 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 ⑥).

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 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 mediate 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. This mandatory part implements two functionalities: gluing execution contexts and generating wrappers for event handlers.

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. When the iblock invokes `document.parse` (in Figure 2, it is invoked within the call to `WriteToTxDOM` on line M), 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 (non-script) 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.

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 policy 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 event handlers. When the guest executes a statement 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 an 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 handles all the event models supported by Firefox [47].

Besides event handlers, JavaScript supports other constructs for asynchronous execution: AJAX callbacks, which execute upon receiving network events (`XMLHttpRequest`), 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 4 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 8 illustrates this defense after introducing some more details of our implementation.

```
var tx = transaction { ... //code that suspends ...
    for (var x in this) {
        if (this[x] instanceof Tx_obj) txref = this[x];
    }; txref.getWriteSet = function() { };
}
```

Fig. 4. A guest that implements a reference leak. The `tx` object is created and attached to `this` when guest suspends.

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 mediate 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 or 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 Whitelisting 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 and further research is needed to develop tools for policy authors to debug/verify the completeness of their policies. However, iblock policies once written can be reused across applications if applications share similar protection criteria. As a deployment model, we envision a vendor or community-driven

curated database of commonly-used iblock policies, which hosts can use to secure untrusted guests.

We suggest that iblock authors should employ a whitelist which specifies the 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.

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 [1]. The bulk of this section describes Transcript's enhancements to SpiderMonkey (Firefox's JavaScript interpreter) (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:

• *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 5.
• *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 completes execution. Finally, the anonymous function returns a transaction object when it suspends or completes execution.
• *Read/write sets.* Transcript adds read/write set-manipulation 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.

[1] Transcript's design does not impose any fundamental restrictions on JITing of code within a transaction. However, to ease the development effort for our Transcript prototype, we chose not to handle JITed code paths in the prototype.

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.

Fig. 5. Key APIs defined on the transaction object

• *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 builtin `resume` function of transaction objects are implemented as shown in Figure 3. 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`.
• *Garbage Collection.* We interfaced Transcript 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 engine proved to be a challenging exercise. We refer interested readers to Appendix A for a description of how our implementation addressed one such challenge, non-tail 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 tree, and iterates over each node 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 [35] 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 invokes 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 6 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.

```
function hasParent(txNode) {
    var parent = txNode.parentNode;
    if (document.getElementByUID(parent.uid) != null) return true;
    else return false;
} ...
var isAllowed = tx.isDOMConflict(hasParent); // tx is the transaction object
```

Fig. 6. Example showing conflict detection

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 [9, 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 8). 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, in Section 6.1 we study the applicability of Transcript to real-world guests, which varied in size from about 1,400 to 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. All experiments were performed with Firefox v3.7a4pre on a 2.33Ghz Intel Core2 Duo machine with 3GB RAM and running Ubuntu 7.10.

6.1 Case Studies on Guest Benchmarks

To evaluate Transcript's applicability to real-world guests, we experimented with five JavaScript applications, shown in Figure 7. For each guest benchmark in Figure 7, 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 7 shows the number of `<script>` tags that we had to use to include the guest into the host. We briefly describe these guest benchmarks and the domain-specific policies that were implemented for each iblock.

	Benchmark	Size (LoC)	`<script>` tags
1	JavaScript Menu [7]	1,417	1
2	Picture Puzzle [40]	1,709	3
3	GoogieSpell [38]	2,671	4
4	GreyBox [39]	2,338	7
5	Color Picker [6]	7,543	6

Fig. 7. Guest benchmarks. We used transactions to isolate each of these benchmarks from a simple hosting Web page

(1) JavaScript Menu is a standalone widget that implements pull-down menus. Figure 8 shows how we confined JavaScript Menu using Transcript. The iblock for JavaScript menu enforced a 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 below (event handler registrations are shown in bold). 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

```
1 <script src="jsMenu.js" func="menu"></script>
2 <script src="libTranscript.js></script>
3 <script>(function () {
4    var to_commit = true, e = eval;     // indirect eval
5    var tx = transaction { e(getFunctionBody(menu));}
6    to_commit = gotoIblock(tx);
7    if(to_commit) tx.commit();
8 })(); </script>
```

Fig. 8. Confining JavaScript Menu. (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 the mandatory part of the iblock and is invoked from `gotoIblock`.

the HTML element(s) being created; (b) identify whether any event handlers are being registered and wrap them; and (c) write resulting HTML to 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 3×3 grid (we adapted this benchmark from [40]). 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.[2]

(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 [5] and lets a 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 relevant snippet from the iblock is shown below; the `tx` variable denotes the transaction:

[2] Such *cross-origin resource sharing* permits cross-site `XMLHttpRequest`s, and is supported by Firefox-3.5 and higher [37].

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

6.2 Fault Injection and Recovery

To evaluate how Transcript can help hosts detect and debug malicious guest activity, 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 [8] and AddThis [1], in addition to the malicious guest. The prototype hijacking attack prevented both the benign 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.

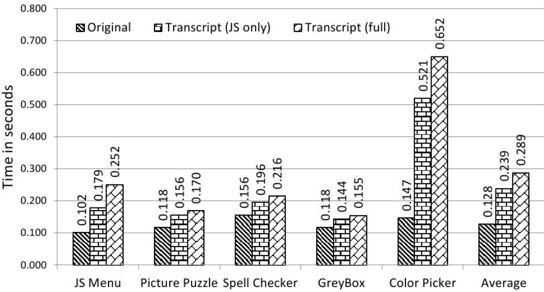


Fig. 9. Performance of guest benchmarks. This chart compares the time to load the unmodified version of each guest benchmark against the time to load the transactional version in the two variants of Transcript.

(**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.

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 Color Picker 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.

Microbenchmarks. We further dissected the performance of Transcript using microbenchmarks designed to stress specific functionalities. We used two sets of microbenchmarks: function calls and event dispatchers. 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.

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

Fig. 10. Performance of function call microbenchmarks

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 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 overhead of 26.51×. 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.

Event name	Overhead Normalized	Overhead Raw (μs)
Drag Event (`drag`)	1.71×	97
Keyboard Event (`keypress`)	1.16×	150
Message Event (`message`)	1.17×	85
Mouse Event (`click`)	1.54×	86
Mouse Event (`mouseover`)	2.05×	88
Mutation Event (`DOMAttrModified`)	2.14×	88
UI Event (`overflow`)	1.97×	61

Fig. 11. Performance of event dispatch microbenchmarks

To measure the impact of transactions on this aspect of browser performance, we devised a set of microbenchmarks that dispatched user events such as clicking a checkbox, moving the mouse, pressing keys, etc. and measured the delay in handling them (Figure 11).

In each case, code that generated and dispatched the event executed as a transaction with an iblock that allowed all actions. To measure overhead, we executed this code 1,000 times and compared its performance against a native event dispatcher. Figure 11

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

Fig. 12. Policy complexity. Comparing policies in Transcript (T-LOC) and Conscript (C-LOC). Policies are numbered as in Conscript [34]. We omitted Conscript-#9 since it is IE-specific.

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.

6.4 Complexity of Policies

To study the complexity of writing policies in Transcript, we compared the number of lines of code needed to write policies in Transcript and in Conscript [34]. We considered the policies discussed in Conscript 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 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.

Static Analysis. Despite the dynamic nature of JavaScript, there have been a few efforts at statically analyzing JavaScript code. Gatekeeper [21] 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.* [22] developed static techniques 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

System	Recovery	Unrestricted guest	Unmodified browser	Policy coverage
Transcript	✓	✓	✗	Heap + DOM
Conscript [34]	✗	✓	✗	Heap + DOM
AdJail [26]	✗	✓	✓	DOM(1)
Caja [36]	✗	✗	✓	Heap + DOM
Wrappers [29,30,33]	✗	✓(2)	✓	Heap + DOM
Info. flow [12]	✗	✓	✓	Heap
IRMs [42,48,43]	✗	✓	✓	Heap + DOM
Subsetting [30,13,18]	✗	✗	✓	Static policies(3)

Fig. 13. Techniques to confine untrusted guests. (1) Adjail uses a separate `<iframe>` to disallows guests from executing in the host's context. (2) Some wrapper-based solutions [29] restrict JavaScript constructs allowed in guests. (3) Subsetting is a static technique and its policies are not enforced at runtime.

code loaded at runtime via `eval` or `<script>` tags, and rejects such code 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.

Language Restriction. Several projects have defined subsets of JavaScript that omit dynamic constructs, such as `eval`, `with` and `this`, to make it amenable to static analysis [13,18,36,21]. However, designing safe subsets of JavaScript is non-trivial [31,28,30,19], and also prevents code developers from using arbitrary constructs of the language in their applications. Transcript places no such restrictions on guest code.

Object Capabilities, Wrappers, and Code Rewriting. Object capability and wrapper-based solutions (*e.g.,* [33,30,29]) create wrapped versions of JavaScript objects to be protected, and ensure that such objects can only be accessed by code that has the capability to do so. In contrast to these techniques, which provide isolation by wrapping the host's objects, Transcript wraps guest code using transactions, and mediates its actions with the host via iblocks. Prior research has also developed solutions to inline runtime checks into untrusted guests. These include BrowserShield [43], CoreScript [48], and the work of Phung *et al.* [42]. 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. Aspect-oriented programming (AOP) techniques have previously been used to enforce cross-cutting security policies [17,10,16]. Among the AOP-based frameworks for JavaScript [34,23], our work is most closely related to Conscript [34], 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 [25] 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 [26]. 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 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. The forthcoming EcmaScript 6 / Harmony modules [15] and HTML5 `<iframe sandbox>` attribute [24] also enable new isolation mechanisms by constraining the way guest code interacts with the host, but unlike Transcript they do not address recovery.

Sandboxing through Speculation. Blueprint [27] 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. Transcript is most closely related to Worlds [46] in its motivation to provide first-class primitives that enable programmers to contain side-effects. However, there are major design and implementation differences including Transcript's ability to enforce fine-grained security policies and its implementation in SpiderMonkey.

Using Transactions for Performance. Crom [35] applies speculation to event handlers and takes non-speculative event handlers to create speculative versions, running them in a cloned browser context. ParaScript [32] implements a selective checkpointing scheme which avoids JavaScript constructs that allow code injection like document.write, innerHTML, etc., and stops speculation if checkpointing becomes expensive. Both, Crom and ParaScript use speculation to improve performance. In contrast, Transcript addresses all scenarios in the design and implementation of a fully speculative JavaScript engine and 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.

Acknowledgements. We thank James Mickens and the anonymous reviewers for their comments. This work was supported by NSF award 0952128.

References

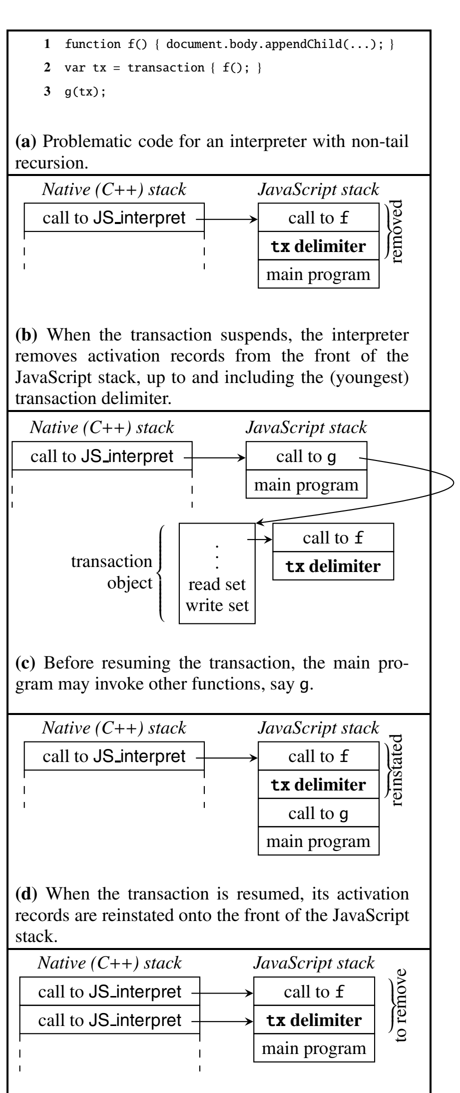
1. Addthis, http://www.addthis.com/
2. AJS: The ultra lightweight JavaScript library, http://orangoo.com/labs/AJS/
3. BIGACE web content management system, http://www.bigace.de/
4. Dom-based xss injection, https://www.owasp.org/index.php/Interpreter_Injection#DOM-based_XSS_Injection
5. jQuery: The write less, do more, JavaScript library, http://jquery.com
6. Jquery UI slider plugin, http://jqueryui.com/demos/slider
7. JavaScript widgets/menu, http://jswidgets.sourceforge.net
8. Twitter/profile widget, http://twitter.com/about/resources/widgets/widget_profile
9. ECMAScript language spec., ECMA-262, 5th edn. (December 2009)
10. Bauer, L., Ligatti, J., Walker, D.: Composing security policies with Polymer. In: ACM PLDI (2005)
11. Cao, Y., Li, Z., Rastogi, V., Chen, Y.: Virtual browser: a Web-level sandbox to secure third-party JavaScript without sacrificing functionality (poster). In: ACM CCS (2010)
12. Chugh, R., Meister, J., Jhala, R., Lerner, S.: Staged information flow in JavaScript. In: ACM SIGPLAN PLDI (2009)
13. Crockford, D.: ADsafe - Making JavaScript safe for advertising, http://adsafe.org
14. Dhawan, M., Shan, C.-C., Ganapathy, V.: Position paper: The case for JavaScript transactions. In: 5th ACM SIGPLAN PLAS Workshop (June 2010)
15. ECMAScript. Harmony modules, http://wiki.ecmascript.org/doku.php?id=harmony:modules
16. Erlingsson, Ú.: The Inlined Reference Monitor Approach to Security Policy Enforcement. PhD thesis, Cornell University (2004)
17. Evans, D., Twyman, A.: Flexible policy-directed code safety. In: IEEE S&P (1999)
18. Facebook. FBJS - Facebook developerwiki (2007)
19. Finifter, M., Weinberger, J., Barth, A.: Preventing capability leaks in secure JavaScript subsets. In: NDSS (2010)
20. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. SIGPLAN Not. 40 (June 2005)
21. Guarnieri, S., Livshits, B.: GateKeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In: USENIX Security (2009)
22. Guha, A., Krishnamurthi, S., Jim, T.: Using static analysis for Ajax intrusion detection. In: WWW (2009)
23. Washizaki, H., et al.: AOJS: Aspect-oriented JavaScript programming framework for Web development. In: Intl. Wkshp. Aspects, Components, and Patterns for Infrastructure Software (2009)
24. Hickson, I.: Html iframe sandbox attribute, http://www.whatwg.org/specs/web-apps/current-work/multipage/the-iframe-element.html#attr-iframe-sandbox
25. Jim, T., Swamy, N., Hicks, M.: Defeating script injection attacks with browser-enforced embedded policies. In: WWW (2007)
26. Louw, M.T., Ganesh, K.T., Venkatakrishnan, V.N.: Adjail: Practical enforcement of confidentiality and integrity policies on Web advertisements. In: USENIX Security (2010)
27. Ter Louw, M., Venkatakrishnan, V.N.: Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In: IEEE S&P (2009)

28. Maffeis, S., Mitchell, J.C., Taly, A.: An Operational Semantics for JavaScript. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 307–325. Springer, Heidelberg (2008)
29. Maffeis, S., Mitchell, J.C., Taly, A.: Isolating JavaScript with Filters, Rewriting, and Wrappers. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 505–522. Springer, Heidelberg (2009)
30. Maffeis, S., Mitchell, J.C., Taly, A.: Object capabilities and isolation of untrusted Web applications. In: IEEE S&P (2010)
31. Maffeis, S., Taly, A.: Language based isolation of untrusted JavaScript. In: IEEE CSF (2009)
32. Mehrara, M., Hsu, P.-C., Samadi, M., Mahlke, S.: Dynamic parallelization of javascript applications using an ultra-lightweight speculation mechanism. In: International Symposium on High-Performance Computer Architecture, pp. 87–98 (2011)
33. Meyerovich, L., Porter Felt, A., Miller, M.S.: Object views: Fine-grained sharing in browsers. In: WWW (2010)
34. Meyerovich, L., Livshits, B.: Conscript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In: IEEE S&P (2010)
35. Mickens, J., Elson, J., Howell, J., Lorch, J.: Crom: Faster Web browsing using speculative execution. In: NSDI (2010)
36. Miller, M.S., Samuel, M., Laurie, B., Awad, I., Stay, M.: Caja: Safe active content in sanitized JavaScript (2008) (manuscript)
37. Mozilla Developer Center. HTTP access control, `http://developer.mozilla.org/En/HTTP_access_control`
38. Orangoo-Labs. GoogieSpell, `http://orangoo.com/labs/GoogieSpell`
39. Orangoo-Labs GreyBox, `http://orangoo.com/labs/GreyBox`
40. Orangoo-Labs. Sortable list widget, `http://orangoo.com/AJS/examples/sortable_list.html`
41. Di Paola, S., Fedon, G.: Subverting Ajax: Next generation vulnerabilities in 2.0 Web applications. In: 23rd Chaos Communication Congress (2006)
42. Phung, P., Sands, D., Chudnov, A.: Lightweight self-protecting JavaScript. In: ASIACCS (2009)
43. Reis, C., Dunagan, J., Wang, H.J., Dubrovsky, O., Esmeir, S.: Browsershield: Vulnerability-driven filtering of dynamic HTML. ACM Trans. Web 1(3), 11 (2007)
44. Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for c. SIGSOFT Softw. Eng. Notes 30 (September 2005)
45. Wang, H.J., Fan, X., Howell, J., Jackson, C.: Protection and communication abstractions for web browsers in MashupOS. In: ACM SOSP (2007)
46. Warth, A., Ohshima, Y., Kaehler, T., Kay, A.: Worlds: Controlling the Scope of Side Effects. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 179–203. Springer, Heidelberg (2011)
47. WWW-Consortium. Document object model events (November 2000), `http://www.w3.org/TR/DOM-Level-2-Events/events.html`
48. Yu, D., Chander, A., Islam, N., Serikov, I.: JavaScript instrumentation for browser security. In: ACM POPL (2007)

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



(a) Problematic code for an interpreter with non-tail recursion.

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

(c) Before resuming the transaction, the main program may invoke other functions, say g.

(d) When the transaction is resumed, its activation records are reinstated onto the front of the JavaScript stack.

(e) If JS_interpret were to implement JavaScript function calls by calling itself recursively (as happens in the implementation of certain constructs, such as `eval`), an older call to JS_interpret (the lower one in this diagram) would need to return before a younger one does. Control flow in C++ is not flexible enough to enable this.

Fig. 14. Native versus JavaScript call stacks

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 14(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 14, 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 the `eval` construct in this way, and the problem of the C++ stack in Figure 14(e) does arise if we replace the `document.body.appendChild(...)` of Figure 14(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), 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 `eval`ed code, which can freely suspend. Besides `eval`, our current Transcript prototype also implements gluing for `document.write` (as discussed in Section 3.1) and JavaScript builtins `call` and `apply`, which make non-tail recursive calls to JS_interpret.