

# Position Paper: The Case for JavaScript Transactions

Mohan Dhawan Chung-chieh Shan Vinod Ganapathy

Rutgers University

## Abstract

Modern Web applications combine and use JavaScript-based content from multiple untrusted sources. Without proper isolation, such content can compromise the security and privacy of these Web applications. Prior techniques for isolating untrusted JavaScript code do so by restricting dangerous constructs and inlining security checks into third-party code.

This paper makes the case that JavaScript must be extended to make isolation a language-level primitive. We propose to extend the language using a new *transaction* construct that allows a Web application to speculatively execute untrusted code and isolate the changes and effects it performs. The Web application can then inspect these speculative actions and commit them only if they comply with the application's security policies. We discuss use-cases that can benefit from JavaScript support for transactions, present a formalization of JavaScript transactions and conclude with implementation considerations.

## 1. Introduction

Modern Web applications combine and use content from multiple, untrusted third parties. For example, *host* Web sites, such as iGoogle, Facebook or Blogger, include third-party code in the form of widgets and advertisements. This third-party code consists largely of JavaScript and HTML content that can be embedded easily in Web pages. As a second example, Web applications increasingly use third-party JavaScript libraries to simplify code development.

We focus on isolating Web applications (especially host Web sites) from untrusted, third-party JavaScript code. One way to achieve this goal is for Web applications to embed such content within a `<frame>` element. However, such isolation is too rigid, and hinders the Web application from easily using features provided by the third-party JavaScript code, such as libraries. Thus, it is more common for the Web application to include third-party content using a `<script>` element. This approach includes third-party code into the same browser sandbox as the Web application itself, and enables the creation of rich Web applications, such as mashups. However, this approach also exposes the Web application to the third-party code, which may be malicious, thereby introducing a security hole. For example, the widget in Figure 1, if included in a Web application using a `<script>` element, can snoop key strokes intended for that application.

Untrusted third-party JavaScript code can potentially be vetted to reveal malicious content, *e.g.*, using static code analysis, but such analysis can be defeated by JavaScript constructs, such as `eval`, that allow for code generation on the fly. Moreover, the code can be obfuscated, which further complicates its analysis. *Subsetting* and *rewriting* have recently

```
1. function keylogger(e) {
2.   document.images[0].src =
3.     "http://evil.com/logger?key="+ e.keyCode;
4. }
5. document.body.addEventListener("keyup", keylogger, false);
```

Figure 1. A JavaScript widget that logs keystrokes.

```
1. var z = transaction {
2.   function keylogger(e) {
3.     document.images[0].src =
4.       "http://evil.com/logger?key="+ e.keyCode;
5.   }
6.   document.body.addEventListener("keyup", keylogger, false);
7. };
8. do { // Introspection block, which encodes security policy
9.   if (z.isSuspended()) {
10.    if (z.getCause().match("addEventListener"))
11.      alert("A script adding an event handler");
12.    else
13.      performAction(z); // perform requested operation
14.    z = z.resume();
15.  }
16. } while(z.isSuspended());
17. z.commit();
```

Figure 2. Isolating the widget in Figure 1.

emerged as popular solutions to address these problems. Subsetting defines a restricted subset of JavaScript, one that excludes hard-to-reason constructs such as `eval`, thereby making third-party code easier to analyze. This approach has been popularized in AdSafe [5] and FBJS [6], among others (*e.g.*, [11, 14]). In contrast, rewriting techniques as employed by Caja [14] and others (*e.g.*, [18, 10, 21]) allow the insertion of inline checks that constrain the runtime behavior of the code. Caja also defines and operates on a subset of JavaScript.

In this paper, we propose a new approach to the problem of isolating untrusted JavaScript content in Web applications. We propose to *extend* the JavaScript language with a new *transaction* construct, which can be used to speculatively execute JavaScript code. In essence, the transaction construct creates a sandbox such that code executing within the transaction cannot modify data outside the sandbox unless the transaction is committed. However, code outside the sandbox can inspect the changes speculatively made by the transaction. A Web application can isolate third-party code, such as a library or a widget, by enclosing it within a transaction, and inspect the changes made by this code before committing them.

Figure 2 illustrates the use of the transaction construct to isolate the untrusted widget from Figure 1. This figure shows how a host application can include the untrusted widget using a transaction in lines 1–7.<sup>1</sup> The transaction itself is a JavaScript object, and the host application can apply policies to inspect or commit the transaction using API calls exported by the transaction object (*e.g.*, `getCause`, `resume` and `commit`).

<sup>1</sup>The third-party code would typically be included using a `script` tag. We support this model as well (see Section 4). However, for ease of exposition, examples in this paper will inline the third-party code.

The policies themselves are written in JavaScript, as shown in lines 8–16 of this example.

However, there are several challenges that must be overcome to add transactions to JavaScript. These challenges stem primarily from the interaction of JavaScript with the browser, such as modifications to the DOM [1] and AJAX requests (*i.e.*, XMLHttpRequest). Such interactions often constitute side-effects, *i.e.*, actions that cannot be undone by aborting a transaction, and call for additional mechanisms. We propose a novel JavaScript *suspend/resume* mechanism to handle such side-effects. When code running within a transaction causes control to transfer beyond the purview of the transaction machinery, *e.g.*, calls `document.write`, the transaction suspends. A suspended transaction indicates that an action with side effects has been requested, which must be inspected by the host Web application’s security policy before being allowed. For example, the transaction in Figure 2 suspends when it calls `addEventListener` on line 6, and transfers control to the *introspection* code in lines 8–16, which encodes the security policy. In this case, the policy allows the third-party code to perform all actions except adding an event listener. When the transaction completes, its actions are committed on line 17.

## 2. Motivating Examples

This section presents two examples that further illustrate the use of transactions to isolate third-party code.

### 2.1 Illustrating JavaScript Suspend/Resume

Consider the code in lines 2–6 of Figure 3. This code opens a pop-up window with a pre-defined URL pointing to an untrusted Web site. If this code is part of a widget or a library included by a Web application, the pop-up could also redirect the parent window by modifying its `location` property. Such behavior can turn an unsuspecting client into the victim of a drive-by download attack.

Figure 3 shows how the Web application can restrict the behavior of this code using transactions. When the code within the transaction executes, it in turn calls `window.open` on line 3. Successful execution of this call would open the pop-up window. However, because the code executes within a transaction, the call suspends, creates a JavaScript object `z` denoting the suspended transaction, and transfers control to the code following the transaction (line 8). This code, called the *introspection block*, encodes the policy: it examines why the transaction suspended and reacts accordingly. (The introspection block appears in a `do...while` loop because a transaction may suspend several times.)

The policy encoded by this introspection block disallows all pop-ups with URLs not in a whitelist. The function `performAction`, which is called on line 13, is supplied by the Web application and performs the action that caused the transaction to suspend. The `z.resume()` call on line 15 resumes the transaction from where it was suspended. When the transaction completes, control passes to the introspection block. Because the transaction is no longer suspended, the loop terminates, so the application executes `z.commit()`, which commits the changes made by the transaction.

```
1. var z = transaction {
2.   function openPopup(url) {
3.     var win = window.open(url, "WindowName",
4.       "resizable=yes, scrollbars=yes, status=yes");
5.   }
6.   openPopup("http://www.untrusted.com");
7. };
8. do { // Introspection block with security policy
9.   if (z.isSuspended()) {
10.    if((z.getObject() instanceof Window) &&
11.      z.getCause().match("open") &&
12.      isWhiteListedDomain(z.getArgs()[0])) {
13.      performAction(z); // perform requested operation
14.    }
15.    z = z.resume();
16.  }
17. } while (z.isSuspended());
18. z.commit();
```

Figure 3. Limiting pop-ups with transactions.

This example illustrates the suspend/resume feature. Code that either modifies the DOM or sends AJAX requests suspends if executed within a transaction, causing control to transfer to the introspection block, where the policy filters the action. The actual DOM modification or XMLHttpRequest is performed by the introspection block (in the `performAction` function) on behalf of the transaction. This situation is analogous to a user-space process executing a system call to access a device, thereby trapping into the operating system, which then accesses the device on behalf of the user-space process.

The suspend/resume feature also extends to nested transactions in a natural way. Consider a situation where a hosting Web application includes code from `adagency.com`. In turn, this code may itself include code from an untrusted fourth-party, `ad-delegate.com`, and so on. In this case, the hosting Web application can use an *outer* transaction to isolate code from `adagency.com`, which in turn can use an *inner* transaction to isolate code from `ad-delegate.com`. If `ad-delegate.com` attempts to issue an AJAX request (or modify the DOM), the inner transaction suspends, and “traps” into the introspection code provided by `adagency.com`. If this code chooses to execute the AJAX request on behalf of `ad-delegate.com`’s code, the request traps to the outer introspection block, which in turn applies its own security policy to the AJAX request. In effect, the AJAX request executes successfully only if it is allowed by the security policies at *each* level of nesting.

The ability to suspend and resume transactions can also be used to prioritize requests made by third-party code. Modern Web applications extensively use AJAX requests to fetch both code and data over the network. When a hosting Web application includes third-party code in the form of a widget or an advertisement, the latter’s AJAX requests are queued along with the host’s requests. However, the host application may wish to remain responsive under network latency and bandwidth constraints, and may wish to issue its own AJAX requests before third parties’. The host application can easily prioritize requests in this way by executing third-party code in a transaction, where AJAX requests suspend the transaction. The host’s introspection policy can then wait until the host has no AJAX request pending, and only then issue the third-party request and resume the transaction. A similar policy can also

```

1. <button id="Search" onclick="doSearch()">
2. <script>
3. var z = transaction {
4.   // includes code from adnetwork.com/insert-ad.js
5.   // which can modify the global variable 'searchUrl'
6. };
7. do { // Introspection block with security policy
8.   var ws = z.getWriteSet();
9.   if (!ws.checkMembership(window, "searchUrl"))
10.    if (z.isSuspended()) performAction(z);
11.    else z.commit();
12.   if (z.isSuspended()) z = z.resume();
13. } while(z.isSuspended());
14. var doSearch = function() {
15.   var searchBox = document.nodes.SearchBox.value;
16.   var searchStr = searchUrl + searchBox;
17.   document.location.assign(searchStr);
18. } </script>

```

**Figure 4.** A snippet of JavaScript code adapted from [www.wsj.com](http://www.wsj.com). The policy examines the transaction’s write set before committing it.

be used to abort slow to load because their AJAX requests take too long to complete.

## 2.2 Illustrating Transaction Read/Write Sets

Figure 4 shows a code snippet from [www.wsj.com](http://www.wsj.com) (adapted from [4]). This snippet includes code from an advertiser ([adnetwork.com/insert-ad.js](http://adnetwork.com/insert-ad.js)). The code defines a search form with a button, which executes the `doSearch` function when clicked. In turn, the execution of this function causes a redirection to a URL obtained by concatenating the query obtained from the search form with `searchUrl`. However, a study by Google [17] showed that first-tier advertising agencies (such as [adnetwork](http://adnetwork.com)) could delegate to other agencies, which in turn could result in the execution of code that redefines the global `searchUrl` variable. If the code in `insert-ad.js` executes within a `<script>` tag in the same browser sandbox as the code from [www.wsj.com](http://www.wsj.com), then redefining `searchUrl` can result in the user being redirected to a malicious website.

Figure 4 shows how transactions can be used to contain the effects of the code included from [adnetwork.com](http://adnetwork.com). The introspection block checks whether the transaction tried to modify the sensitive `searchUrl` variable, and performs actions on behalf of the transaction only if `searchUrl` has not been modified (line 10). Changes made within the transaction are committed to memory only if the untrusted code completes execution without modifying the `searchUrl` variable (line 11). To support the enforcement of such policies, JavaScript transactions maintain *read/write sets*, which track the set of memory locations accessed/modified by the transaction. The read/write sets are exposed at the language level by an API (e.g., `z.getWriteSet`), thereby allowing security policies to inspect their contents (e.g., `checkMembership`).

## 3. A Lambda Calculus with Transactions

To explain concisely and formally how transactions underpin the motivating examples above, we present a call-by-value lambda calculus with transactions and specify its operational semantics [7]. The essential idea is to use the evaluation context to delimit transactions and isolate them from external resources [8].

### 3.1 Formalization

The syntax of our core language is defined by the following grammar. A value  $V$  is a special case of an expression  $M$ .

Expressions	$M ::= n \mid \ell \mid x \mid \lambda x. M \mid M+M \mid MM \mid RW\{M\}$ $\mid \text{commit } M \mid \text{introspect } M(x.M)(x.M)$ $\mid \text{new } M \mid \text{read } M \mid \text{write } MM$ $\mid \text{suspend } M \mid \text{resume } MM$
Values	$V ::= n \mid \ell \mid \lambda x. M$ $\mid RW\{V\} \mid RW\{C[\text{suspend } V]\}$
Contexts	$C ::= \square \mid C+M \mid V+C \mid CM \mid VC$ $\mid \text{commit } C \mid \text{introspect } C(x.M)(x.M)$ $\mid \text{new } C \mid \text{read } C \mid \text{write } CM \mid \text{write } VC$ $\mid \text{suspend } C \mid \text{resume } CM \mid \text{resume } VC$
Metacontexts	$D ::= \square \mid D[RW\{C\}]$

Here we assume integer constants  $n$  (for illustration), an infinite supply of heap locations  $\ell$ , and lexically scoped variables  $x$ .

A read/write set  $RW$  consists of the read set  $R$  and the write set  $W$ . Whereas  $R$  is a relation between locations and values,  $W$  is a partial function from locations to values. For example, suppose that the global heap comprises three locations  $\ell_1, \ell_2, \ell_3$ , containing 10, 20, 30 respectively. The global write set is then  $\{\ell_1 \mapsto 10, \ell_2 \mapsto 20, \ell_3 \mapsto 30\}$ . Suppose now that a transaction reads 10 from  $\ell_1$ , writes 25 to  $\ell_2$ , reads 30 from  $\ell_3$ , writes 35 to  $\ell_3$ , reads the new value 25 back from  $\ell_2$ , and initializes a new heap location  $\ell_4$  to 45. Then, the global write set stays the same, but the read set of the transaction changes from the empty set to  $\{\ell_1 \mapsto 10, \ell_3 \mapsto 30\}$ , and the write set of the transaction changes from the empty set to  $\{\ell_2 \mapsto 25, \ell_3 \mapsto 35, \ell_4 \mapsto 45\}$ .

A context  $C$  is a special case of an expression in which a subexpression next to be evaluated is replaced by a hole  $\square$ . Roughly speaking, whereas a read/write set represents the heap state of an ongoing transaction (akin to the contents of private pages in the address space of a thread), a context represents the control state of an ongoing transaction (akin to the sequence of activation frames on the execution stack of a thread). Whereas many operational semantics (including Maffeis *et al.*’s for JavaScript [9]) leave control state implicit in contextual or congruence rules, we make it explicit so as to specify how transactions suspend. We write  $C[M]$  for the expression obtained by replacing the hole in  $C$  with  $M$ . For example, if  $C = \square$  then  $C[\lambda x. x] = (\lambda x. x)0$ .

A transaction expression  $RW\{M\}$  is formed by *delimiting* an (untrusted) expression  $M$  with a read/write set  $RW$  (initially empty). This formation is similar to how, in a typical language with exception handling, a try-expression is formed by delimiting an expression with a handler. The delimiter is akin to the boundary between a user process and an OS kernel. In particular, if the expression  $M$  is actually a value  $V$ , then the transaction is finished; if  $M$  has the form  $C[\text{suspend } V]$ , then the transaction is suspended. These are the two cases of transaction expressions that are values.

A metacontext  $D$  is a sequence of pairs of read/write sets  $RW$  and contexts  $C$ , which are the heap states and control

$$\begin{aligned}
D[n_1 + n_2] & \quad \sim D[n] \\
& \text{where } n \text{ is the sum of } n_1 \text{ and } n_2 \\
D[(\lambda x. M)V] & \quad \sim D[(x \mapsto V)M] \\
D[RW\{C[\text{commit } R' W'\{M\}]\}] & \quad \sim D[RW''\{C[0]\}] \\
& \text{where } W'' = \text{Write}(W, W') \\
D[\text{introspect } (RW\{M\}) (x_1.M_1) (x_2.M_2)] & \quad \sim D[(x_i \mapsto V)M_i] \\
& \text{where } M = V \text{ and } i = 1 \text{ or } M = C[\text{suspend } V] \text{ and } i = 2 \\
D[RW\{C[\text{new } V]\}] & \quad \sim D[RW'\{C[\ell]\}] \\
& \text{where } \ell \text{ is fresh and } W' = \text{Write}(W, \{\ell \mapsto V\}) \\
D[RW\{C[\text{read } \ell]\}] & \quad \sim D[R'W\{C[V]\}] \\
& \text{where } V = W(\ell) \text{ and } R' = R \text{ if } W(\ell) \text{ is defined,} \\
& \quad V = \text{Read}(D, \ell) \text{ and } R' = R \cup \{\ell \mapsto V\} \text{ otherwise} \\
D[RW\{C[\text{write } \ell V]\}] & \quad \sim D[RW'\{C[V]\}] \\
& \text{where } W' = \text{Write}(W, \{\ell \mapsto V\}) \\
D[\text{resume } (RW\{C[\text{suspend } V]\}) V'] & \quad \sim D[RW\{C[V']\}]
\end{aligned}$$

**Figure 5. The transition relation  $\sim$ .**

states of a sequence of nested ongoing transactions. A meta-context is also an expression in which a subexpression next to be evaluated is replaced by a hole  $\square$ .

To manipulate read/write sets, we define two auxiliary functions. The partial function `Read` maps a metacontext and a location to a value, by looking up the location in the meta-context's read/write sets:

$$\text{Read}(D[RW\{C\}], \ell) = \begin{cases} W(\ell) & \text{if } W(\ell) \text{ is defined,} \\ \text{Read}(D, \ell) & \text{otherwise.} \end{cases}$$

The function `Write` combines two write sets  $W$  and  $W'$  into one, preferring entries in  $W'$  over those in  $W$ :

$$\text{Write}(W, W')(\ell) = \begin{cases} W'(\ell) & \text{if } W'(\ell) \text{ is defined,} \\ W(\ell) & \text{otherwise.} \end{cases}$$

Finally, in Figure 5, we define a (small-step) transition relation  $\sim$  between machine states. A machine state is just a transaction expression  $RW\{M\}$ ; thus, we treat the entire machine as executing a top-level transaction (whose read set does not matter). In the transitions, we write  $(x \mapsto V)M$  to denote the (capture-avoiding) substitution of  $V$  for  $x$  in  $M$ . The transition relation so defined is patently deterministic modulo the renaming of locations and variables. We denote the transitive closure of  $\sim$  by  $\sim^+$ .

The `introspect` facility defined here is very simple: it only lets a policy observe whether a transaction is finished or suspended, and with what value. In our proposed implementation, transaction objects feature JavaScript methods that provide access to their read/write sets. These methods can be used, for example, to see if the transaction has read any sensitive information, *e.g.*, cookies, that should not be leaked.

More generally, the model of locations and variables in our lambda calculus is much simpler than JavaScript's, which involves, for example, looking up variables along scope chains and properties along prototype chains. These complications

can be modeled without any fundamental difficulty—either using the `Read` and `Write` functions defined above, or by writing a JavaScript interpreter in our lambda calculus.

### 3.2 Examples

To illustrate the transition relation, we present some small example programs. For clarity, we write `var x = M1; M2` to abbreviate the expression  $(\lambda x. M_2)M_1$ . To express loops (which typical policies are), we also write function  $f(x)M$  to abbreviate the value

$$\lambda x. (\lambda y. \text{var } f = \lambda x. yyx; \lambda x. M)(\lambda y. \text{var } f = \lambda x. yyx; \lambda x. M)x.$$

The latter abbreviation has the crucial property that

$$\begin{aligned}
& D[(\text{function } f(x) M)V] \\
& \sim^+ D[(f \mapsto \text{function } f(x) M)(x \mapsto V)M].
\end{aligned}$$

Take for example the policy  $P_1$ , defined as the value

$$\text{function } p(t) \text{ introspect } t (r. r) (a. p(\text{resume } t(a + 1))).$$

Ignoring the use of `resume` for the moment, suppose we apply this policy to the trivial transaction  $\{\{\{3 + 4\}\}$  (that is, the expression  $3 + 4$  delimited by an empty read set and an empty write set). This transaction immediately finishes with the result 7, which is observed by the policy due to  $(r. r)$ :

$$\begin{aligned}
& \{\{\{P_1(\{\{\{3 + 4\}\})\}\} \\
& \sim \{\{\{P_1(\{\{\{7\}\})\}\} \\
& \sim^+ \{\{\{\text{introspect } (\{\{\{7\}\})\}\} \\
& \quad (r. r) \\
& \quad (a. P_1(\text{resume } t(a + 1)))\} \\
& \sim \{\{\{7\}\}
\end{aligned}$$

Suppose that  $\ell$  is a location shared between the host application and the contained transaction. Even if the transaction reads and writes  $\ell$  in the course of its computation, as long as the policy does not commit the transaction—which  $P_1$  does not—the changes will not be reflected in the global write set. For example, the transaction below increments the content of  $\ell$  and returns the result.

$$\begin{aligned}
& \{\{\{\text{var } l = \text{new } 1; P_1(\{\{\{\text{write } l(\text{read } l + 1)\}\})\}\} \\
& \sim \{\{\{\ell \mapsto 1\}\{\text{var } l = \ell; P_1(\{\{\{\text{write } l(\text{read } l + 1)\}\})\}\} \\
& \sim \{\{\{\ell \mapsto 1\}\{P_1(\{\{\{\text{write } \ell(\text{read } \ell + 1)\}\})\}\} \\
& \sim \{\{\{\ell \mapsto 1\}\{P_1(\{\{\{\ell \mapsto 1\}\}\{\text{write } \ell(1 + 1)\}\})\}\} \\
& \sim \{\{\{\ell \mapsto 1\}\{P_1(\{\{\{\ell \mapsto 1\}\}\{\text{write } \ell 2\}\})\}\} \\
& \sim \{\{\{\ell \mapsto 1\}\{P_1(\{\{\{\ell \mapsto 1\}\}\{\ell \mapsto 2\}\{2\}\})\}\} \\
& \sim^+ \{\{\{\ell \mapsto 1\}\{\text{introspect } (\{\{\{\ell \mapsto 1\}\}\{\ell \mapsto 2\}\{2\}\})\}\} \\
& \quad (r. r) \\
& \quad (a. P_1(\text{resume } t(a + 1)))\} \\
& \sim \{\{\{\ell \mapsto 1\}\{2\}\}
\end{aligned}$$

The finished transaction has the read set  $\{\ell \mapsto 1\}$  and the write set  $\{\ell \mapsto 2\}$ . They are discarded by `introspect` in the policy, even though the result 2 of the transaction, computed using them, is retained.

To allow the transaction's write set to take global effect, the policy must commit the transaction explicitly, as in the

following policy  $P_2$ .

```
function  $p(t)$  introspect  $t$ 
    ( $r$ . var  $z = \text{commit } t$ ;  $r$ )
    ( $a$ .  $p(\text{resume } t(a + 1))$ )
```

(The variable  $z$  above is just to receive the dummy result 0 returned by `commit`.) Applying  $P_2$  to the same transaction modifies  $\ell$  globally to 2:

```
{\ell \mapsto 1}\{\text{var } l = \text{new } 1; P_2(\{\}\{\text{write } l(\text{read } l + 1))\}\}
\rightsquigarrow^+ \{\ell \mapsto 1\}\{\text{introspect } (\ell \mapsto 1)\{\ell \mapsto 2\}\{2\}\}
    ( $r$ . var  $z = \text{commit}$ 
        ( $\ell \mapsto 1\}\{\ell \mapsto 2\}\{2\}$ );
    ( $r$ )
    ( $a$ .  $P_2(\text{resume } t(a + 1))$ )
\rightsquigarrow \{\ell \mapsto 1\}\{\text{var } z = \text{commit}(\ell \mapsto 1)\{\ell \mapsto 2\}\{2\}; 2\}
\rightsquigarrow \{\ell \mapsto 2\}\{\text{var } z = 0; 2\}
\rightsquigarrow \{\ell \mapsto 2\}\{2\}
```

Hence, we have no rollback operation—to roll back a transaction is simply to never commit it.

Finally, we illustrate the use of `suspend` and `resume` using the transaction

```
 $T = \text{var } z = \text{write } \ell(\text{suspend } (\text{read } \ell));$ 
     $\text{write } \ell(\text{suspend } (\text{read } \ell)).$ 
```

Twice in a row, this transaction sends the content of  $\ell$  to the host as a request and puts the host's response back into  $\ell$ . The policies  $P_1$  and  $P_2$  above implement an integer incrementation service, so applying  $P_1$  or  $P_2$  to  $T$  increments the content of  $\ell$  twice in a row:

```
{\ell \mapsto 1}\{P_1(\{\}\{T\})\}
\rightsquigarrow^+ \{\ell \mapsto 1\}\{\text{introspect } T' (r.r) (a.P_1(\text{resume } T'(a+1)))\}
\rightsquigarrow \{\ell \mapsto 1\}\{P_1(\text{resume } T'(1 + 1))\}
\rightsquigarrow \{\ell \mapsto 1\}\{P_1(\text{resume } T' 2)\}
\rightsquigarrow \{\ell \mapsto 1\}\{P_1(\ell \mapsto 1)\}\{\text{var } z = \text{write } \ell 2;
     $\text{write } \ell(\text{suspend } (\text{read } \ell))\}\}
\rightsquigarrow^+ \{\ell \mapsto 1\}\{P_1(\ell \mapsto 1)\{\ell \mapsto 2\}\{\text{write } \ell 3\}\}
\rightsquigarrow^+ \{\ell \mapsto 1\}\{3\}$ 
```

where  $T'$  is short for  $\{\ell \mapsto 1\}\{\text{var } z = \text{write } \ell(\text{suspend } 1);$   
 $\text{write } \ell(\text{suspend } (\text{read } \ell))\}$ .

## 4. Implementation Considerations

In this section, we outline some practical problems that must be addressed in a browser-based implementation of JavaScript transactions.

**Modifying the `<script>` tag.** The code snippets used earlier in the paper showed how *inline* scripts could be isolated using transactions, *i.e.*, the JavaScript code to be isolated was available in its entirety, and could be included within the transaction construct. However, third-party scripts are often included using a `<script>` tag whose `src` attribute specifies the URL from which the code must be fetched. For example, the code on lines 4 and 5 in Figure 4 would be fetched using `<script src="adnetwork.com/insert-ad.js">`. In such

```
var corefunc = getFunctionBody(core.toString());
var morefunc = getFunctionBody(more.toString());
var calfunc = getFunctionBody(calendar.toString());
var e = eval; // indirect eval
var z = transaction {
    e(corefunc); e(morefunc); e(calfunc);
}; // Introspection code goes here.
```

Figure 6. Ensuring script execution in the global scope.

cases, the browser fetches the script code from the URL and executes it as soon as the code has been fetched over the network. Since the `<script>` tag is HTML code, it cannot directly be placed within a JavaScript transaction.

To ensure that code fetched using a `<script>` tag executes within a transaction, we propose to add a new `txfunc` attribute to the `<script>` tag. The idea is to convert the fetched script into a JavaScript function object, and then execute this function within a JavaScript transaction. For the example in Figure 4, the code would be fetched as `<script src="adnetwork.com/insert-ad.js" txfunc="adcode">`, which will encapsulate the code from `insert-ad.js` within a function `adcode`. The function `adcode` can then be invoked within a JavaScript transaction.

While seemingly straightforward, the above modification to the `<script>` tag introduces an additional complication. When a script is included in a Web application using a `<script>` tag, the JavaScript code in the script executes in the scope defined by the Web application, *i.e.*, the global scope. However, the modification to the `<script>` tag described above causes the fetched code to execute in the scope of the function specified in the `txfunc` attribute. This problem becomes apparent when scripts are included using multiple `<script>` tags, and each script defines/modifies variables that are used by others. For example, consider the code snippet shown below, which uses two files from the Mootools library [2] and a third `calendar` script that uses functions defined by this library. The functions and variables defined in each of these `<script>` tags would be defined in the corresponding functions defined in the `txfunc` tags, and would not be visible in the global scope, thereby breaking functionality.

```
<script src="mootools-1.2.4-core.js" txfunc="core"></script>
<script src="mootools-1.2.4.2-more.js" txfunc="more"></script>
<script src="calendar-v1.0.1.js" txfunc="calendar"></script>
```

This problem can be solved by first extracting the code of each of the `txfunc` functions and executing them in the global scope within the transaction. Figure 6 illustrates the code to achieve this. The `getFunctionBody` function extracts the code of the function, which is then executed using an indirect `eval` within the transaction. The use of an indirect `eval` is crucial—the ECMAScript standard specifies that indirect `evals` are executed in the global scope. However, note that any changes to the global scope made within the transaction will only appear in its write set unless the transaction commits.

**Executing event handlers in transactions.** Event handlers are callbacks that execute when specific events, such as mouse clicks, happen. JavaScript code enclosed in a transaction may define event handlers that are added to the global scope when the transaction commits. Because these event handlers are defined by the (potentially untrusted) JavaScript

code, their execution must be subject to the same security policy as the transaction. However, an event handler can also execute after the transaction that defined it has completed execution. The original execution context of the transaction that defined it may no longer be available when the event handler is triggered.

To ensure that the execution of an event handler is monitored using the same security policy of the transaction that defines it, we create a wrapper around the event handler. The goal of the wrapper is to execute the handler within a transaction, and associate the same security policy with that transaction. That is, suppose that a transaction defines an `onClick` handler named `clickhand`. In the introspection block of this transaction, we include code that does the following: (1) create a function `tx_clickhand`, which wraps `clickhand` in a transaction; (2) associate the same introspection block with `tx_clickhand`; and (3) register `tx_clickhand` as the `onClick` handler. Therefore, `tx_clickhand` is triggered when the `onClick` event happens, which in turn ensures that the execution of the event handling code is subject to the same security policy as the transaction that defined it.

## 5. Related Work

We are not aware of prior work on JavaScript transactions. Our discussion of related work focuses on the use of transactions for security/reliability and on recent work on restricting untrusted JavaScript code.

**Restricting untrusted JavaScript code.** Several recent research projects and commercial efforts have investigated techniques to restrict the execution of untrusted JavaScript code. Notable commercial efforts include ADSafe [5], FBJS [6] and Caja [14], which define subsets of JavaScript that are easier to reason about using static analysis. The work of Maffeis *et al.* [12, 9, 11] formalizes and presents a security analysis of such language subsetting techniques. Other research projects, including BrowserShield [18], CoreScript [21], and the works of Phung *et al.* [15] and Maffeis *et al.* [10], propose to rewrite untrusted JavaScript code to insert runtime checks and wrappers that restrict the behavior of the code.

As discussed in Section 1, we propose an alternative approach—that of *extending* the language with transactions. Since our approach provides isolation by extending JavaScript, rather than subsetting or rewriting, it supports the execution of unmodified JavaScript and does not place any restrictions on the constructs that third-party code can use.

Our work is most closely related to ConScript [13], which proposes an aspect-oriented approach to restrict the execution of untrusted JavaScript code. Like our work, ConScript also enhances the JavaScript language and supporting HTML tags (*e.g.*, `<script>`) to specify policies that govern the execution of third-party code. However, unlike our work, which specifies policies at the granularity of data accesses, *e.g.*, using the transactions read/write sets, ConScript’s policies are specified at the granularity of functions, which serve as pointcuts. A second difference is that our work proposes speculative execution of JavaScript, whereas ConScript inlines policy checks with the execution of the code that it monitors. Further

research is needed to determine whether these two techniques compare in their ability to enforce security policies on untrusted code execution.

**Using transactions for security.** Transactions and speculative execution mechanisms have previously been used to improve software security and reliability (*e.g.*, [19, 16, 3]). However, the work most closely related to ours is the one by Sun *et al.* [20] on one-way isolation. This work describes a sandboxing mechanism that allows isolated execution of untrusted code. As in our work, 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, this paper discussed a similar approach but applied it to the problem of isolating JavaScript code. Accordingly, their work is realized by making changes to the OS, whereas ours requires changes to the JavaScript interpreter.

## References

- [1] Document object model. <http://www.w3.org/DOM>.
- [2] Mootools—a compact JavaScript framework. <http://mootools.net>.
- [3] A. Birgisson, M. Dhawan, U. Erlingsson, V. Ganapathy, and L. Iftode. Enforcing authorization policies using transactional memory introspection. In *ACM CCS*, 2008.
- [4] R. Chugh, J. Meister, R. Jhala, and S. Lerner. Staged information flow in JavaScript. In *ACM SIGPLAN PLDI*, 2009.
- [5] D. Crockford. ADSafe - Making JavaScript safe for advertising. <http://adsafe.org>.
- [6] Facebook. FBJS - Facebook developerwiki. 2007.
- [7] M. Felleisen. *The Calculi of  $\lambda_v$ -CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Comp. Sci. Dept., Indiana Univ., 1987.
- [8] O. Kiselyov, C.-C. Shan, and A. Sabry. Delimited dynamic binding. In *ICFP*, pages 26–37, 2006.
- [9] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *APLAS*, October 2008.
- [10] S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with filters, rewriting and wrappers. In *ESORICS*, 2009.
- [11] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted Web applications. In *IEEE S&P*, 2010.
- [12] S. Maffeis and A. Taly. Language based isolation of untrusted JavaScript. In *IEEE CSF*, 2009.
- [13] L. Meyerovich and B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE S&P*, 2010.
- [14] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. 2008. Manuscript.
- [15] P. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting JavaScript. In *ASIACCS*, 2009.
- [16] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating systems transactions. In *ACM SOSP*, 2009.
- [17] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser analysis of web-based malware. In *First Hot-Bots workshop*, 2007.
- [18] 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.
- [19] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *OSDI*, 1996.
- [20] W. Sun, Z. Liang, R. Sekar, and V. N. Venkatakrishnan. One-way isolation: An effective approach for realizing safe execution environments. In *NDSS*, 2005.
- [21] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *ACM POPL*, 2007.