1. The Goal of the Technique

The goal of the technique is to instrument code at run-time to handle the dynamism of JavaScript.

2. The limitation of AJAXScope

AJAXScope has the limitation of handling dynamic behavior of JavaScript code. AJAXScope instruments a JavaScript program by inserting policy nodes (i.e., the instrumentation code) to the program before it is loaded to the browser. However, a JavaScript program could be dynamically changed. Such changes can happen when the properties or the prototype of an object are changed or a function is dynamically wrapped, just name a few.

Example 1 given an object instrumented using AJAXScope as

```
var obj={
   a: wrap(a),
   b: wrap(b)
}
```

Later on at run-time,

```
1  obj.a = run_time_value();
2  delete obj.b;
3  obj.c=new_func();
4  obj.prototype=p;// p is a run-time object
```

Figure 1

In this case, "a" is assigned a new value and the instrumentation of "a" is removed, "b" is all deleted, and a new property "c" is added; a run-time object "p" is assigned as the prototype. Now, no any policy node exists for "obj".

Example 2 given a function "func" instrumented using AJAXScope as

```
func = wrap(func);
```

At run-time, "func" may be further wrapped as

```
func = function(cond){
   some code handling cond;
   wrap(func);
}
```

Figure 2

In this case, although wrap(func) is still available, the instrumentation is incomplete because the behavior of "func" is different.

One possible way of AJAXScope to cope with the dynamism (e.g., code shown in Figures 1 and 2) is to insert such "global" policy nodes that monitor every operation and check the use and change of instrumented code (i.e., implementing the definition-use analysis of the data-flow).

However, such a possible solution is very inefficient. For example, in Figure 1, after each of the four lines of code, the "global" police node checks the change of "obj", and inserts new instrumentation code for the new properties and prototypes. Nevertheless, some monitoring actions are unnecessary, such as checking line 2. Even worse, the cost of such monitoring actions can be prohibitive. For example, one of the worst cases is line 4: if "p" itself has a prototype chain, the "global" police node needs to inert new police code to each of every prototype in the prototype chain of "p". Moreover, if "p" and the prototypes in its prototype chain have their own policy nodes that are conflicted with the policy node of "obj" (i.e., the policy for instrumenting "p" and its prototypes is different from the policy of instrumenting "obj"), all the code introduced by the policy nodes of "p" and its prototypes needs be removed.

3. Our Solution

Our technique instruments code at run-time to handle the dynamism of JavaScript. The technique instruments a function when the function is invoked. In the two examples of Section 2, our technique instruments "obj" and "func" whose code is updated as shown in Figures 1 and 2.

4. Instrumentation Procedure

The developer specifies the event and the element for instrumentation. If a specified event fired on a corresponding specified element, the technique instruments this event's handlers.

The procedure requires all the handlers of an event be known. The current DOM model does not support reflection on event handlers (i.e., addEventListener). Thus, the technique needs to (1) trace dynamic event handlers in the source code and (2) instrument the code that adds event handlers to put dynamic handlers into the element-event-handler table. For example, given the code

```
if(Math.random()){
   domNode.addEventListener(.,f1,.);
}else{
   domNode.addEventListener(.,f2,.);
}
```

The instrumented code is

```
if(Math.random()){
   domNode.addEventListener(.,f1,.);
   add_to_ele_evt_hndlr_tbl(domNode, f1);
}else{
   domNode.addEventListener(.,f2,.);
   add_to_ele_evt_hndlr_tbl(domNode, f2);
}
```

Figure 3

5. Handle Dynamic Code Injection

There are many tricks for dynamic code injection, but all these tricks finally either need eval() to execute the injected code or append a new "script" object containing the injected code for execution (i.e., appendChild(script) to DOM).

eval() injection

The technique instruments eval() by inserting instrumentation code to the string parameter and performs on the string parameter any necessary tracing tasks mentioned in Section 4. Thus, all information about the code injected by eval() is known.

appendChild() injection

Suppose the injected code is known (i.e., injected code is either a string or from a source in the same domain), we can *always* trace appendChild(script) in the source because, as the entry point of executing the innerHTML code, appendChild(script) can *never* be an arbitrary string at run-time of JavaScript programs. (appendChild(script) can be coded in an arbitrary string format as part of an innerHTML string. However, if a string containing appendChild(script) is to be executed, the string has to be injected to DOM by another appendChild(). Thus, there is a "root" appendChild() that is *always* in the executed format.)

(Jim, please have a doubt-check the above claim.)

Thus, the technique can parse the code for searching appendChild() and if its parameter is a "script" object, the technique instruments the script.src/innerHTML, and then hook the instrumented source to DOM.

If the code is from an external source, it needs a trick to hook up the instrumented code: using XHR to get the source code, instrumenting XHR.ResponseText, and appending to the DOM tree the instrumented text instead of the external source.

(So we don't even need the monitoring thread to check the change of the script objects.)

6. Limitations of the Technique

There are three limitations of the technique. One limitation is that the technique does not guarantee to handle all dynamically added event handlers. The technique parses add/removeEventListener because in the current DOM model, there is no reflection of event handlers. If add/removeEventListener is added by arbitrary strings of innerHTML, it could fail to find them by parsing the source code. Nevertheless, in the latest DOM specification,[1] event handlers can be reflected. Thus, it does not need to parse event handlers in the source code in the near future when browsers implement the new DOM specification.

Another limitation is that the technique doesn't guarantee to handle the code that is dynamically injected from other domains. However, we suggest the developer create a proxy code for the code in other domains. Besides assisting in instrumentation using our technique, such a proxy can reduce the latency of loading JavaScript to the browser. Thus, such a proxy would be very acceptable in the real-world applications.

The third limitation is that the technique misses the dynamic behavior occurred before any DOM event is fired. The technique instruments code when the handlers of a DOM event are invoked. Code that is executed not for responding DOM events is not instrumented. If the developer needs to monitor the dynamic behavior of such code, the developer has to use AJAXScope style instrumentation to perform def-use analysis in the data-flow. However, the majority of such code is at the top-level, which is executed as soon as it is loaded to the browser. Maybe limited amount of code is both dynamically changed and then invoked at the top-level. If it is true, the necessity of monitoring such code's dynamic behavior would be limited. (It would be very interesting to see some empirical experiment results about this issue.)

[1] http://www.w3.org/TR/2001/WD-DOM-Level-3-Events-20010823/events.html#Events-EventListenerList