Let me explain the four corner cases you mentioned in the following order:

1. Functions that have a distinct execution context
2. Functions of host objects (DOM related functions)
3. External code from different domains
4. Top-level dynamic code

1. Functions that have a distinct execution context. For a function having a distinct execution context, such as settimeout(),eval(), and Function(), the instrumentation program wraps these functions so that the code introduced through them will be instrumented. Using setTimeout() as the example, we have

```
settimeout = settimeout(Wrap(arguments[0]), arguments[1]);
//Wrap is the insert_probes() in the previous email.
```

2. Functions of host objects. In general, we cannot instrument/wrap these functions. However, we can specify those functions that can introduce code and treat them individually. Now, I consider only addEventListener/attachEvent. (Is there any other function to address?) For event listeners, the instrumentation program
    a. Parses the code to find addEventListener
    b. Creates an event-element-handler table that contains all the event handlers and the corresponding events and elements
    c. When an event is fired, checks elements on which the event fired, and uses the event and element to look up the correspond event handler
    d. Removes the event handler from the element
    e. Adds an new event handler that wraps the removed handler to the element

   For example, given aDomNode.addEventHandler('click', function(){}, true), there is an entry at the event-element-handler table for it. When a click event is fired on aDomNode, the code of the instrumentation program is

```
var handler = lookup_handler(aDomNode, 'click'); //handler is returned as a string
aDomNode.removeEventHandler('click', handler, true);
aDomNode.addEventHandler('click', wrap(handler), true);
```

   The instrumentation program keeps updating the event-element-handler table, using the monitoring thread to check addEventHandler or removeEventHandler in dynamic code. (Thanks for the tip of using DOMSubtreeModified in FireFox!)

3. External code from different domains. We cannot handle external code from other domains. As you mentioned, there is no way we can guarantee to get the source, even though XHR could help in certain cases. However, we can suggest the developer create a proxy code in the same domain for those external sources. In most Web sites, such proxy would likely be available, or acceptable, for the purpose of reducing the latency of loading the code to the browser, and thus, it does not introduce any extra work specifically for addressing this instrumentation issue.

4. Top-level dynamic code. Basically, there are two ways of dynamic code injection: one is eval() and its peers as mentioned in case 1; the other is innerHTML. The way we handle case 1 (i.e., wrap them) can help to instrument top-level dynamic code injected by eval() and others alike. However, if the dynamic code is injected by innerHTML, you're absolutely right that no information about which specific part of code or function was executed would be known. The available information is what *code* was injected and executed but not which *execution path* was taken.

However, the information that a piece of top-level dynamic code is injected and executed might still be useful, although the execution detail is unknown. Such information can help the developer perform post-mortem analysis using static techniques (i.e., at lexical analysis time as you pointed out). Current static analysis techniques for JavaScript programs don't have a sound solution for dynamic code, with the assumption that dynamic code is used limitedly. (The paper, An Analysis of the Dynamic Behavior of JavaScript Programs, has a summary of the assumptions of current static techniques about dynamic code.) The information provided by our technique would benefit these techniques to improve their soundness. (Would such arguments be reasonable to you?)