Statically Locating Web Application Bugs Caused by Asynchronous Calls

Yunhui Zheng, Tao Bao, Xiangyu Zhang
Department of Computer Science, Purdue University
West Lafayette, IN 47907
{zheng16, tbao, xyzhang}@cs.purdue.edu

ABSTRACT

Ajax becomes more and more important for web applications that care about client side user experience. It allows sending requests asynchronously, without blocking clients from continuing execution. Callback functions are only executed upon receiving the responses. While such mechanism makes browsing a smooth experience, it may cause severe problems in the presence of unexpected network latency, due to the non-determinism of asynchronism. In this paper, we demonstrate the possible problems caused by the asynchronism and propose a static program analysis to automatically detect such bugs in web applications. As client side Ajax code is often wrapped in server-side scripts, we also develop a technique that extracts client-side JavaScript code from server-side scripts. We evaluate our technique on a number of real-world web applications. Our results show that it can effectively identify real bugs. We also discuss possible ways to avoid such bugs.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; D.3.4 [**Programming Languages**]: Processors—*Debuggers*

General Terms

Algorithms, Reliability, Experimentation

Keywords

JavaScript, Ajax, Static Analysis, Automatic Debugging

1. INTRODUCTION

JavaScript (JS) is becoming more and more important as web applications are providing more and more complex functionalities on the client side. In Web 1.0, browsers simply render pages received from servers so that little JavaScript is needed. With the concept of using browsers as the regular computation platform, complex computations and interactions with the server side are becoming a trend. Consequently, JS starts to play a substantial role in Web 2.0 applications. In the early stage of Web 2.0, an annoying problem is that whole-page reloads are required during the client server communications and end-clients are not allowed to perform any actions in the meantime. Such experience degrades users' satisfaction.

WWW 2011, March 28–April 1, 2011, Hyderabad, India.
ACM 978-1-4503-0632-4/11/03.

```
1    var req = new XMLHttpRequest();
2    req.open('GET', 'login.php?name='+ username, true);
3    req.onreadystatechange = function () {
4      if(req.readyState == 4) {
        // do something
       }
     };
5    req.send(null);
```

Figure 1: Ajax Request and Response Handler.

To address such problems, the *Asynchronous JavaScript and XML* (Ajax) technique is introduced. Ajax allows sending requests asynchronously, without blocking the current browsing. Upon receiving data from the server, callback functions are invoked to process the response in background. Since then, more and more complex and interactive web applications are powered by Ajax, such as Gmail or Google Maps, inspired by the smooth browsing experience.

A typical Ajax request and its response handler are shown in Fig. 1. It constructs a *GET* request to a server side page *login.php* with an argument *name*. The third parameter of method *req.open()* at line 2 is *true*, meaning the request is sent asynchronously. Then at line 3, an anonymous function is registered as the handler to event *onreadystatechange*. Attribute *readyState* holds the status of the request and ranges from 0 to 4 to indicate communication stages. When the status changes, the event handler will be invoked.

JS execution is sequential and event driven. Events such as user interactions and server side responses are queued and handled one by one. However, asynchronous requests allow arbitrary user actions, such as exercising interfaces and sending new requests, happen between a request and the corresponding response. Depending on the order of the user actions and the response, non-deterministic behavior may be observed. Some may lead to serious problems.

Fig. 2 presents execution models with and without asynchronous requests and the possible problems. It describes the interactions of three entities: the user, the browser, and the server. Each box represents an action. A user can perform an action (denoted as $\boxed{A}$), such as clicking a link or pressing a button, or seeing the outcome of a previous action (denoted as $\boxed{V}$). The browser can send a request ($\boxed{R}$) or process the response from the server ($\boxed{P}$). The server can serve a request ($\boxed{X}$). An arrow between two boxes represents they have happens-before relationship. Actions that are atomic are circled.

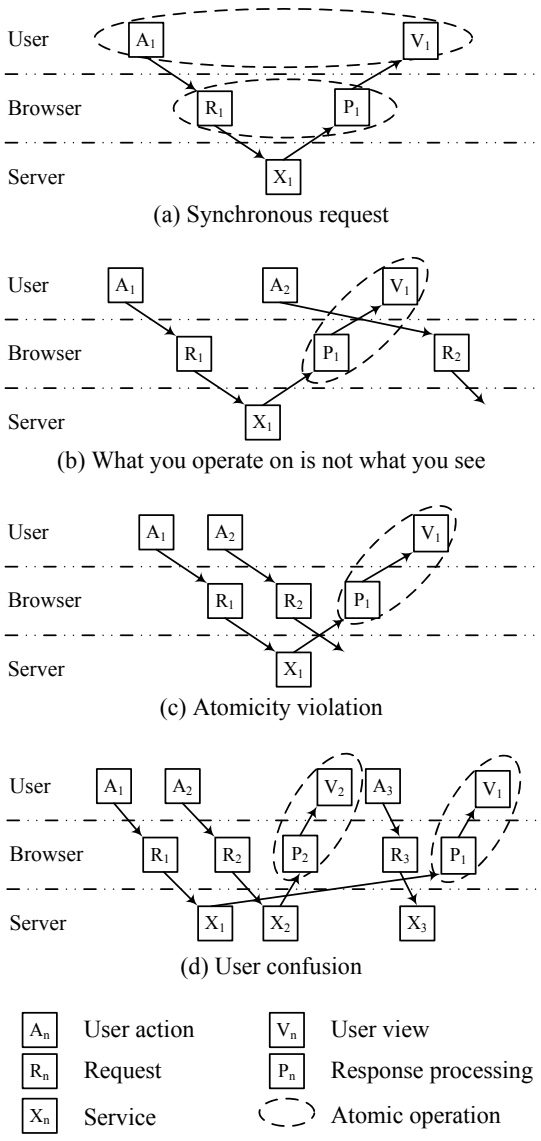

Figure 2: User, browser, and server interactions and possible problems.

(a) In the first case, requests are sent synchronously. In such a scenario, once the user triggers a request, the browser starts to refresh. The user can not take any further actions until the request is served, processed, and visualized. In other words, there is atomicity between what the user triggers and what she sees. Such atomicity is ensured by the atomicity of the request and its response at the browser level. In this execution model, user actions are processed one after the other. There is no out-of-order execution and hence no non-determinism.

(b) The remaining three cases correspond to asynchronous requests. With asynchronous requests, there is no longer atomicity between a request and its corresponding response. Consequently, the user level atomicity is not promised. The arbitrary interleavings of actions lead to various problems.

Fig. 2 (b) presents the buggy case of "what you operate on is not what you see". We use subscripts to represent the origin of actions. For instance, R_2 is the request send originating from the user action A_2. In case (b), *the user can perform an action* A_2 *before she sees the outcome of the previous action* A_1 due to the client-server latency and the nature of Ajax. When the user performs A_2, she thinks she is operating on the data that she sees, which is still the data when A_1 happens. However, note that JS execution is facilitated by an event queue that admits events in their arrival order. Assume from the perspective of the browser, A_2 arrives after X_1. Hence, R_2 happens after P_1. If P_1 updates some data that is used by R_2, R_2 is indeed operating on the new data instead of what the user sees. The inconsistent data may be used to compose the request so that the inconsistency gets propagated permanently to the server side. We call it the *inconsistency* problem.

For instance, the user clicks a button to delete an image that she sees. But when the request is composed and sent, the image to be deleted is undesirably updated. As a result, a wrong image is deleted *on the server side*. Note that such damage is irrevocable. We will see a real example in Section 2.

(c) In case (c), request R_2 happens before response P_1. The atomicity of R_1 and P_1 may be broken by R_2 if they operate on the same data. This may lead to runtime exceptions. For example, assume an object is tested to be not null in R_1. As JS execution is largely sequential, programmers often mistakenly assume that the object remains not null till it is used in P_1 to access its attributes. However, an interleaving R_2 may set the object to null and lead to an exception in P_1. Exceptions can lead to strange browser behavior if not properly captured. We call it the *atomicity violation* problem.

(d) In case (d), whereas user action A_1 happens before A_2, their outcomes V_1 and V_2 are visualized in the opposite order. Assume these two actions are performed on the same object. When the user performs an action A_3 according to what she sees, she might get confused and think that she is operating on the result of A_1. We call it the *user confusion* problem.

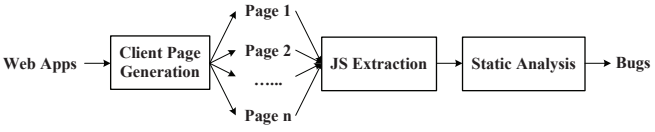

Figure 3: System Overview

In this paper, we develop a static program analysis to detect the *inconsistency* and *atomicity violation* problems. The analysis is directly applicable to web applications without the need of installing them on a server or using a browser to explore pages. In particular, our technique takes a web application and automatically extracts client side pages by exploring the relevant paths of the server side code. For

each extracted client side page, the non-JS content (such as the HTML part) is filtered out. We finally perform program analysis on the remaining JS code and report faults. Fig. 3 gives an overview of the system.

We observe that the *inconsistency* problem can be formulated as a special data race problem in JS code. It involves a response handler and a function that modifies permanent state, such as a request sender that modifies server content or a JS method that changes cookies. We call them the *side-effect function*. In other words, depending on the order of the handler and the side-effect function, undesirable modifications may be made to the permanent state. Traditional data race detection techniques assume multiple threads or processes, and rely on reasoning about synchronizations and shared variables [7, 19, 16]. In our case, JS execution is essentially single-threaded and there are no synchronizations.

The *atomicity violation* problem defined in case (c) is similar to that for regular programs. Our problem is unique in the sense that we don't have synchronizations to leverage. Furthermore, traditional atomicity violation detection relies on users to provide an approximation of atomic regions in subject programs [9]. In our case, such information can be precisely derived from JS code as we know where the atomicity should hold. That is the duration between a request and its response. Thus, our technique is fully automated.

Our JS analysis identifies all the handlers that process server responses for asynchronous requests, and all the event handlers that handle user actions. The executions of these handlers (and the functions that they call) can arbitrarily interleave. We then analyze the global variables that can be accessed in these handlers. Race and atomicity violation detections are performed on the handlers that can interleave and operate on the same variables.

Note that the problems we are addressing can not be handled by existing web application testing tools. There are two kinds of client side testing techniques. The first kind is record-and-replay, such as Seleninum, WebKing and Sahi. For most of these tools, at the beginning, testers need to manually exercise input controls on client GUI. The user actions are recorded and replayed. Replay could be faithful or more intelligent with automatic manipulations. Such techniques require lots of manual efforts. The other kind of techniques is to automatically discover user interfaces and exercise them to reduce the human efforts, such as ATUSA [14]. In general, these testing techniques focus on achieving high coverage on the user action sequences. They hardly serve our purpose as our problems are due to non-determinism. Even with the same action sequence, a bug may or may not manifest itself, depending on the interleaving.

Our contributions are summarized as follows.

- We formally model the possible problems in the presence of asynchronous requests for web applications. We show that they can be formulated as special cases of data race and atomicity violation problems.
- We propose static program analysis for JS code to detect the problems. The analysis is automatic. It is interprocedural, handles aliases, and supports various popular third-party libraries.
- We develop an approach to extract JS code from server side scripts. It avoids deploying and executing the subject web application. The technique takes a server side script, rewrites it to a C program. During rewriting, we replace predicates in the script with calls to our functions. Our technique then automatically executes the transformed program. During execution, our functions gain control at each predicate point and guide the execution to go through relevant paths. Consequently, various client side pages are generated.
- We evaluate our technique on a set of real world web applications. Our technique can find a number of real bugs on these applications with reasonable cost.

```
@ admin/inc/admin.js

003 var thumbArray;

227 function populateThumbArray(id, name){
229   var request = initializeXMLHttpRequest();
      ...
233   thumbArray = Array();
235   if(request){
236     request.open('GET',
                     '../ajallerix.php?action=3&catid=' + id,
                     true);
      // Register an Ajax response handler
237     request.onreadystatechange = function(){
238       if(request.readyState==4){
239         var todo = request.responseText.split(':');
241         for(x=0; x < todo.length - 1 ; x++){
              ...
248           thumbArray[x] = ...;
249         }
250         showThumbs(name);  // render received images
251       }
252     }
253     request.send(null);
254   }
255 }

552 function doDelete(id){
      ...
555   var request = initializeXMLHttpRequest();
556   if(request){
557     url = 'admin.php?...&imgname=' + thumbArray[id].name;
558     request.open('GET', url, true);  // Exception!
559     request.onreadystatechange = function(){ ... };
      ...
563     request.send(null);
564   }
566 }

568 function showThumbs(name) {
581   for(i = 0; i < thumbArray.length ; i++){
664     myDeleteDC.innerHTML += '<input type="button" onclick="'
                              + 'doDelete(\''+ i '\');">'
                              + ...;
      }
681 }
```

Figure 4: Code snippet from Ajallerix[4]

2. MOTIVATING EXAMPLE

We use bugs in Ajallerix [4] to motivate our technique. Ajallerix is a web image gallery. A user can be either a guest or an administrator. The administrator can upload, modify and organize images. The guest can view published images. Images are classified into groups and can be presented in different styles, such as full screen and slide-show.

Ajax is used to make browsing smooth. For example, when presenting images in slide-show, images are automatically loaded one by one. The functionality is implemented by asynchronously sending requests to fetch images periodically and rendering them when the responses arrive.

Fig. 4 shows the relevant code snippet. Function *populateThumbArray*() is an *onclick* event handler. It will be invoked when a category link is clicked. Function *doDelete*() is associated with the *onclick* event to delete an image.

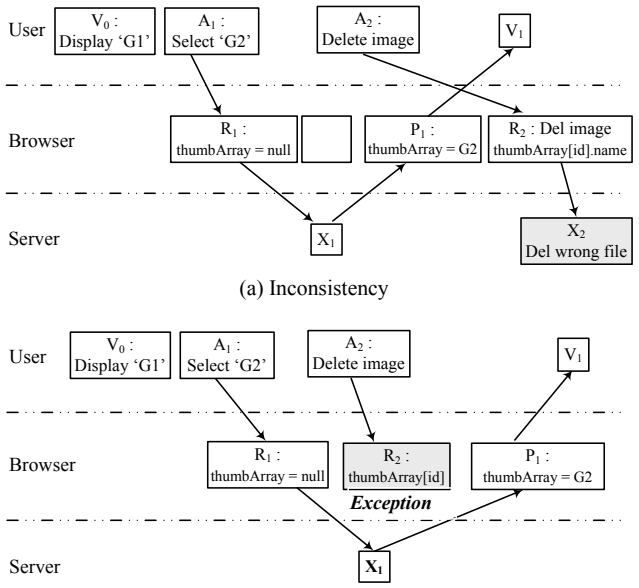

Figure 5: Buggy Behavior Example. Bugs are manifested at the highlighted places.

In function *popluateThumbArray*(), an Ajax object is created at line 229. The Ajax request is set as '*GET*' and sent to *ajallerix.php* with two arguments *action* and *catid*, which tells the server to send images with group id *catid*. At line 237, an anonymous function is registered as an Ajax response handler.

Note that the global variable *thumbArray* has two definition points (lines 233 and 248) and gets used in *url* at line 557. In particular, *thumbArray* will be initialized to an empty array immediately when function *populateThumbArray*() is invoked. It is set to the values returned from the server side in the response handler. The code has both an inconsistency bug and an atomicity violation that leads to runtime exceptions.

Inconsistency. With the inconsistency bug, the user may delete a wrong image on server. The bug inducing interaction sequence is shown in Fig. 5 (a). At first, images of group *G1* are shown on the current page. The user requests to show images of *G2* in action $\boxed{A_1}$. While waiting for the display of the images, the user issues another request $\boxed{A_2}$ to delete an image that she sees, which is an image of *G1*.

Assume the response $\boxed{X_1}$ arrives before $\boxed{A_2}$. The JS engine queues the events in their arrival order. Later, the response handler $\boxed{P_1}$ at line 237 is executed before the delete request is sent in *doDelete()* $\boxed{R_2}$. Note, at line 557 inside *doDelete()*, the delete request is composed by *thumbArray[id].name* such that the request sent will use the *name* of an image belonging to *G2* because *thumbArray* is redefined in the preceding response handler. The server is stateless and will process such a request. Undesirably, the user deletes an image that was not intended. Please note the bug is *not* caused by the delay between the client internal state update and the UI repaint. Instead, it is induced by the network latency as well as the delay between the response arrival and the state update if JS engine is occupied when the response arrives.

Exception Caused by Atomicity Violation. In another scenario (Fig. 5 (b)), the delete request is processed ($\boxed{R_2}$) before the response of the earlier request is handled ($\boxed{P_1}$). Variable *thumbArray* is set to empty at line 233 in $\boxed{R_1}$, but gets used at line 557 in $\boxed{R_2}$. A runtime out-of-bound exception is generated. The root cause is that the empty array is not supposed to be visible to other methods before it is properly defined at line 248 according to the response. It is an atomicity VIOLATION. The bug occurs when the server has a heavy workload or the network is busy so that there is non-trivial delay on receiving the response. If the exception is not properly handled, strange browser behavior can be observed. In this case, we observed FireFox invokes the edit window while the user clicks the delete button. Besides, other controls become RESPONSELESS.

3. JAVASCRIPT STATIC ANALYSIS

In this section, we introduce the JS static analysis that detects the inconsistency and the atomicity violation problems. Previously, these problems are illustrated as the interleavings of actions at the user, browser and server levels as in Fig. 2. Due to the difficulty of co-analysis on the three levels, we aim to detect them only from the perspective of the browser.

Projected to the view of browser, the inconsistency problem is essentially a data race problem between write accesses in the response handler and reads in event handlers that react to user actions (such as the write in $\boxed{P_1}$ and the read in $\boxed{R_2}$ in Fig. 5 (a)). In other words, the user may take some action based on what she sees, but the racy condition determines that the action is not processed until the data seen by the user is undesirably updated by a response, causing inconsistency. Note that JS execution is event driven. Due to the uncertainty of the arrival time of a response, a response handler can interleave with any user event handler in an arbitrary order. Therefore, the essence of the analysis for the inconsistency problem is to check *if a piece of JS code has an asynchronous response handler that writes to a global variable and a user event handler that reads the same global variable*. In this paper, we particularly focus on reads (in a user event handler) that can cause permanent damage such as data lost on the server side or change of cookie state on the client side. Specifically, we consider user event handlers that send requests to the server and the requests are composed of the global variable in a racy condition. We also consider handlers that update cookies based on the variable.

The atomicity violation problem is caused by the intended atomicity between a request (e.g. $\boxed{R_1}$ in Fig. 5) and the corresponding response ($\boxed{P_1}$) being violated by the interleaving of non-serializable accesses from other event handlers ($\boxed{R_2}$). Note that the three parties involved: the request, the corresponding response, and the violating event handler, must access the same variable. Otherwise, although there are user event handlers that execute in between a request and the corresponding response, the interleavings do not have any effect. In other words, the interleaved execution is equivalent to the one without any interleavings. A more thorough analysis is presented in Table 1, which shows some of the possible access patterns in the three parties. Case one corresponds to the request does not access the variable,

Table 1: Assume event handler s executes in between a request r and its response. Different access patterns in the three parties may or may not lead to atomicity violations. R and W mean read and write to the same variable.

case	request r	interleaving event handler s	response to r	serializable
1		R	R	yes
2		R	W	yes
3		W	W	yes
4	R	W	W	no
5	R	R	W	yes
6	W	W	W	yes
7	W	R	W	no
8	W	W	R	no
...	...			

```
1   var seed, prefix;
2   function bar(b1, b2) {
3     var myId = b1;
4     var req = new XMLHttpRequest();
5     req.open('GET', 'login.php?id='+ myId, true);
6     req.onreadystatechange = function () {
7       if(req.readyState == 4) { prefix = req.responseBody; }
8      };
9     req.send(null);
10  }
11
12  function foo(f1, f2, f3) {
13    var temp = prefix + f3;
14    bar(temp, f1);
15  }
```

Figure 6: Inter-procedure analysis example

the interleaving handler reads it, and the response reads it too. This interleaving is serializable as it is equivalent to the request and the corresponding response execute atomically first and then the handler s. Case 2 can be serialized to s first and then the atomic execution of r and its response. Case 5 can be serialized to the same order. In comparison, patterns 4, 7, and 8 are not serializable. Although case 8 is not serializable, in our context, it is unlikely that a request writes to a variable and the corresponding response reads it. Because a response handler is mostly to process the retrieved data and update the client side variables. Hence, our analysis essentially searches for patterns 4 and 7.

In the following, we describe our analysis for the inconsistency problem in details. The analysis is interprocedural. It assumes call graphs and points-to set [1]. For each variable x in the program our analysis computes the set of global variables that x is directly or transitively dependent on. In other words, the value of x may be affected by the values of those global variables. We further compute the global variables that are defined by each function, including response handlers. A consistency problem is found *if there is a request or a cookie update using a variable dependent on a global variable, and the same global variable is in the definition set of a response handler.*

Take the code in Fig. 6 as an example, in which function *foo*() calls function *bar*() at line 14. The global variable dependence set of *myId* at line 3 is {*prefix*}. This is because *temp* is computed from *prefix* at line 13, then used as a parameter at line 14 to call *foo()*. Hence, we know that the request at line 5 is composed of the global variable *prefix*. Due to the definition at line 7, the global variable definition set of the anonymous handler function is {*prefix*}. According to our buggy pattern, it is an inconsistency problem.

We describe our analysis formally in the *datalog* language [5]. Datalog uses a Prolog-like notation for relation computation. It provides a neat representation for program analysis, especially flow analysis. Data flow facts can be formulated as relations. Analysis is represented as inference rules on these relations.

Relations are in the form $p(X_1, X_2, ..., X_n)$ with p being a predicate. $X_1, X_2, ..., X_n$ are terms of variables or constants. In our context, variables are essentially program artifacts such as statements, program variables and function calls. A predicate is a declarative statement on the variables. For example, $path(L_1, L_2)$ denotes if there are paths from statement L_1 to L_2. If so, we say the pair L_1 and L_2 is in the path relation.

Rules are a way of expressing logic inferences. The form of a rule is

$$H \text{ :- } B_1 \text{ \& } B_2 \text{ \& ... \& } B_n$$

H and B_1, B_2,...B_n are either relations or negated relations. We should read the :- symbol as "if". The meaning of a rule is if B_1, B_2,...B_n are true then H is true. A sample inference rule is that $path(L_1, L_3)$:- $path(L_1, L_2)\&path(L_2, L_3)$.

Relations can be either inferred or atoms. In program analysis, we often start with a set of atoms that are basic facts and then infer other more interesting relations.

Our analysis is presented in Fig. 7. We have atoms describing facts that can be directly acquired from the WALA infrastructure. For instance, the atom $invoke(F, H, L)$ denotes a function F calls function H at call site L. It essentially encodes the call graph of the JS code. One can intuitively understand that we have a relation with three attributes. A tuple is present in the relation when the predicate is true.

The $def(L, X)$ and $use(L, X)$ relations describe if a variable X is defined and used at L, respectively. L is a label that represents a program point. Relation $depOnArgument(L, X, M, A)$ determines if variable X defined at L is (transitively) dependent on the M^{th} formal argument A of the residence function. The first inference rule for this relation means that if X is directly defined from A, then the predicate is true. The second rule says if X is defined from Y, Y is dependent on a formal argument A, and the definition of Y is reachable to the definition of X, then X is dependent on A.

Relation $depOnGlobal(L, X, G)$ determines if X defined at L is (transitively) dependent on a global variable G. The first two rules are similar to those for $depOnArgument$. The third rule describes that if X is dependent on an argument A and there is a call to the function that provides an actual argument dependent on a global variable G, then X is dependent on G.

Relation $defGlobal(F, G)$ represents the global variable definition set of a function. It dictates function F defines a global variable G. It simply aggregates global definitions in F and those in F's children.

Relation $inconsistency(L, F, X, G)$ computes the set of inconsistency errors in the program. It means the use of variable X at the request sending point L is (transitively) dependent on a global variable G and G is defined in a response handler F or its children. The first rule handles a simple case in which the global variable is directly used in the request. The second rule handles that the request is transitively dependent on the global variable. The third one

[1]The infrastructure WALA we used is able to generate call graphs and perform points-to analysis for JS code.

Atoms

$invoke(F, H, L)$: F calls H at program point L.
$inFunction(L, F)$: program point L is in function F.
$global(X)$: variable X is a global variable
$formal(F, M, X)$: variable X is the M^{th} formal argument of function $F()$
$actual(L, M, X)$: variable X is the M^{th} actual argument at call site L.
$path(L_1, L_2)$: there is an intraprocedural path from L_1 to L_2.
$defFree(X, L_1, L_2)$: X is not defined along a path from L_1 and L_2.

Rules

$def(L, X)$:- "$L:\ X = Y$" /* X is defined at program point L */
$use(L, Y)$:- "$L:\ X = Y$" /* Y is used at program point L */
$def(L, X)$, $use(L, Y)$, $use(L, Z)$:- "$L:\ X = Y\ op\ Z$"

/* var X defined at L depends on the M^{th} argument A of the function.*/
$depOnArgument(L, X, M, A)$:- $def(L, X)$ & $use(L, A)$ & $formal(F, M, A)$ & $inFunction(L, F)$
$depOnArgument(L, X, M, A)$:- $def(L, X)$ & $use(L, Y)$ & $depOnArgument(L_1, Y, M, A)$ & $path(L_1, L)$ & $defFree(Y, L_1, L)$

/* var X defined at L depends on global var G. */
$depOnGlobal(L, X, G)$:- $def(L, X)$ & $use(L, G)$ & $global(G)$
$depOnGlobal(L, X, G)$:- $def(L, X)$ & $use(L, Y)$ & $depOnGlobal(L_1, Y, G)$ & $path(L_1, L)$ & $defFree(Y, L_1, L)$
$depOnGlobal(L, X, G)$:- $depOnArgument(L, X, M, A)$ & $inFunction(L, F)$ & $argDepOnGlobal(F, M, A, G)$

/* the M^{th} formal argument A of F depends on global var G. */
$argDepOnGlobal(F, M, A, G)$:- $formal(F, M, A)$ & $invoke(H, F, L)$ & $actual(L, M, Y)$ & $depOnGlobal(L, Y, G)$

/* global var G is defined in F. */
$defGlobal(F, G)$:- $def(L, G)$ & $global(G)$ & $inFunction(L, F)$
$defGlobal(F, G)$:- $invoke(F, H, L)$ & $defGlobal(H, G)$

/* inconsistency exists for global var G used in request sent at L_1 and defined in the response handler F. */
$inconsistency(L_1, F, G, G)$:- $usedInRequest(L_1, G)$ & $defGlobal(F, G)$ & $ajaxResponseHandler(F)$
$inconsistency(L_1, F, X, G)$:- $usedInRequest(L_1, X)$ & $depOnGlobal(L_2, X, G)$ & $path(L_2, L_1)$ & $defFree(X, L_2, L_1)$ & $defGlobal(F, G)$ & $ajaxResponseHandler(F)$
$inconsistency(L_1, F, A, G)$:- $usedInRequest(L_1, A)$ & $inFunction(L_1, H)$ & $argDepOnGlobal(H, M, A, G)$ & $defGlobal(F, G)$ & $ajaxResponseHandler(F)$

Native JS

$isAjax(X)$:- "$X = newXMLHttpRequest()$" /* X is an ajax object. */
$isAjax(X)$:- "$X = new\ ActiveXObject(''Msxml2.XMLHTTP'')$"
$isAjax(X)$:- "$L:\ X = Y$" & $isAjax(Y)$
$usedInRequest(L, X)$:- $isAjax(R)$ & "$L:\ G = R.post()$" & $actual(L, 1, X)$ /* X is used in a request composed at L */
$usedInRequest(L, X)$:- $isAjax(R)$ & "$L:\ G = R.open()$" & $actual(L, 2, X)$
$ajaxResponseHandler(F)$:- $isAjax(R)$ & "$R.onreadystatechange = F$"

Prototype

$isAjax(X)$:- "$X = newAjax.Request()$"
$isAjax(X)$:- "$L:\ X = Y$" & $isAjax(Y)$
$usedInRequest(L, X)$:- $isAjax(R)$ & "$L:\ R.url := X$"
$usedInRequest(L, X)$:- $isAjax(R)$ & "$L:\ R.parameters := X$"
$ajaxResponseHandler(F)$:- $isAjax(R)$ & "$R.OnSuccess := F$"

jQuery and YUI

Similar to *prototype*, omitted

Figure 7: Datalog Rules for Detecting Inconsistency Bugs.

```
   <?php
1    switch($_GET[‘action’]) {
2      case 1:
3  ?>
4      <html> ...<script>...</script>... </html>
5  <?php
6        break;
7      case 2:
8  ?>
9      <html> ...<script>...</script>... </html>
10 <?php
11       break;
12     default:
13       die("no action"); break; }
   ?>
```

Figure 8: PHP sample code

handles that the request is transitively dependent on the global variable through an argument. Rules for inconsistency caused by cookie updates are elided for brevity.

The relation $usedInRequest(L, X)$, dictating X be used in a request sent out at L, and $ajaxResponseHandler(F)$, dictating F be a response handler are library dependent. Most websites make use of third-party JS libraries to provide rich functionalities and speed up the development procedure. Study in [3] shows that *JQuery*, *Prototype* and *Yahoo User Interface* are the three most frequently used libraries for Ajax development. We support all these libraries. Requests and response handlers are in different forms in these libraries. Hence, we develop different inferences rules as shown in Fig. 7. Our technique does not analyze the body of libraries. Instead, we only provide abstraction for the interface functions. Recursions do not impose problems for datalog inference as a fixed point would be reached at the end as our analysis is monotonic.

The rules for detecting atomicity violations are omitted due to the space limitation.

4. HANDLE SERVER-SIDE SCRIPT

In the previous section, we presented the analysis on client-side pages. In many cases, client side pages can be generated dynamically by server-side scripts. As a server side script may have conditional statements, a piece of server code may generate several different client pages. In order to make our testing approach practical, we develop a technique to extract JS from the server code without deploying the application.

4.1 Dynamic generated JavaScript

On the aspect of developer, HTML and JavaScript are embedded in the server-side pages. When server script is evaluated, HTML as well as JS will be printed out to client.

Server side scripts are like normal programs so that they have conditional statements and loops, which entail different control flow paths and thus different versions of client side pages. In the example shown in Fig. 8, the execution path is dynamically decided by the value of *action* sent in the client's request. Thus, two different pages can be generated.

Several existing techniques are relevant to solving client side page extraction problem. Minamide's work [15] analyzes server scripts and presents the possible client pages in regular expressions. However, since such regular expressions are not valid JS code, program analysis would not be directly applicable. Besides, [6] and [11] apply symbolic execution technique to generate inputs to cover feasible control flow paths in server side scripts. Thus, the pages generated by their technique are precise. However, symbolic execution is usually expensive and may have scalability problems. In our context, the requirement on feasible paths is not as stringent. We can adopt a light-weight approach as long as the JS code is completely captured and the syntax is correct.

if P then $trueBranch$ else $falseBranch$	→	if $choice(2)$ then $trueBranch$ else $falseBranch$
switch P { case c_1: ... case c_2: ... ... case c_{n-1}: ... default: ...	→	switch $choice(n)$ { case 0: ... case 1: ... ... case n-2: ... case n-1: ...
while P do $LOOP$ end	→	if $choice(2)$ then $LOOP$

Figure 9: PHP Rewriting Rules

To get client pages, A naive way is to filter out the php code in a server page. However, it is problematic. For example in Fig. 8, simply filtering out the php code results in the page having two group of *<html>* labels. Hence, a constraint is that the control flow should be respected. Then another thought is to traverse all possible control flow paths. Although server page is not complex, in large programs, traversing all paths does not scale.

We leverage the observation that JS are usually embedded by two means. The first one is to put them in a separate '*.js*' file and then include the file at run-time. The second one is directly to put in a page outside the range of server script tags, like *<?php?>*. From the perspective of the server script, JS code or JS file location are present as a long constant string or a constant string with a fixed pattern. Based on such assumptions, we can ignore paths that do not encounter such strings. It is rare to load JS from database or to construct them via string concatenation. Instead, strings dynamically generated by server-script are usually user data or DOM objects. They are irrelevant to our analysis.

4.2 PHP Rewriting

Then, the key idea of our client page extraction is to brute-forcely traverse control flow paths that have JS relevant strings.

We still assume all paths are feasible in a piece of server script for the sake of static analysis. We blacklist branches that can not produce JS relevant strings. Then, replace all predicates in the script with calls to our runtime functions. Upon execution, our runtime functions steer executions to cover all the paths that are not blacklisted.

We use the open source PHP compiler[2] for the above purpose. It allows us to manipulate *Abstract Syntax Tree* (AST) to conduct predicate replacement and eventually translates the script to a stand-alone C code that can be compiled and executed independently. Each execution explores an unblocked path and generates one version of the client page.

The rewriting is done on AST. In particular, the technique traverses the tree to look for two things: constant strings containing JS as well as predicates on the path to reach those strings. These predicates are replaced with a function $choice(n)$, where n denotes the number of choices allowed in the predicate. For instance, a boolean predicate has two choices. At runtime, all the choices are explored except those that can not lead to JS strings.

The rules for rewriting are shown in Fig. 9. Please note *loop* is transformed into *if* because we don't need repetitive client page content to be generated. Instead, one iteration or no-iterations are the two choices we are interested in.

Let's look at how the extraction works for the example shown in Fig. 8. There are two constant strings containing JS code and they share the same *switch* predicate. Then in transformed C program, the two corresponding branches will be traversed, generating two client pages.

To handle external functions such as database related functions *mssql_connect()*. We replace the function with an empty string based on the assumption that it's not a common practice to store JS in databases.

Note that not all generated pages are valid because of functions such as *die*() or infeasible control flow paths. Thus, before feeding the pages to our JS analysis, a HTML syntax checker is used to filter out invalid pages.

5. EVALUATION

Our system is implemented based on the WALA infrastructure[1] and the open source PHP compiler (phc) [2]. The implementation consists of two parts.

The client page extraction part is implemented as a plug-in of the phc compiler. The plug-in traverses the AST provided by the phc parser and replaces the predicates. We also modify phc's PHP-to-C translation plug-in to insert our path manipulation function. Then the transformed C code is compiled and executed to generate pages.

The JS analysis is implemented on top of WALA. We leverage WALA's existing analysis passes to construct call graphs and control flow graphs. We also make use of WALA's points-to analysis.

We apply our technique on a set of real world web applications and websites. They are listed as follows.

Ajallerix: online image gallery.
http://developer.novell.com/wiki/index.php/Ajallerix
AjaxLogin: secure login plug-in.
http://www.jamesdam.com/ajax_login/login.html
XHTML Chat: online chatting plug-in.
http://chat.plasticshore.com/
Ajax File Browser: view, add, edit or delete files.
http://sourceforge.net/projects/ajaxfb/
Tixean chat: web chat based on AJAX.
http://sourceforge.net/projects/tixean-chat/
Phormer: PHP without MySQL PhotoGallery.
http://sourceforge.net/projects/rephormer/
Quizzy: PHP AJAX quiz library.
http://sourceforge.net/projects/quizzy/
Rogozhka chat: Ajax Easy Customizable Web Chat plug-in.
http://sourceforge.net/projects/php-ajax-chat/

All experiments are run on an Intel Dual Core 2.5GHz machine with 2GB memory. The OS is Linux-2.6.35.

Table 2: Program characteristics.

Program	#file	total LOC	JS LOC *
XHTML Chat	5	706	138
AjaxLogin	2	614	517
Tixean chat	11	2989	596
Rogozhka chat	34	6542	951
Phormer	6	8145	1755
Quizzy	8	5561	5165
Ajallerix	16	10960	3927
Ajax File Browser	204	76024	3776
www.msn.com	1	6378 +	3921

* For compressed JS where line breaks and spaces are removed, use "JS Beautifier" to pretty print it.
+ Pretty printing using "HTML Beautifier" for the same reason

Table 2 presents the characteristics of the benchmark programs. Column *'#file'* is the number of files, including *.php*, *.inc* and *.html* in the program. The *'total LOC'* column lists the total lines of code in the program files. *'JS LOC'* lists the total lines of independent JS code. The MSN website is just the client side web page as we don't have the access to the server side scripts. Our technique can nonetheless analyze individual client side pages.

Table 3: Client page extraction.

Program	#php / rel.	php LOC / rel.	#page gen.	JS LOC gen.
XHTML Chat	4 / 0	307 / 0	1+	138
AjaxLogin	1 / 0	105 / 0	1+	517
Tixean chat	10 / 3	1024 / 597	1	596
Rogozhka chat	33 / 0	5554 / 0	1+	951
Phormer	6 / 4	6390 / 6188	13	955
Quizzy	8 / 0	396 / 0	0*	455
Ajallerix	16 / 4	6605 / 663	7	1327
Ajax File Browser	203 / 9	63882 / 3492	24	1893

* Quizzy is a plug-in app and doesn't provide a runnable server page. Manual setup required.
+ '.html' Static server page.

Table 3 presents the results of the client page extraction. The second column shows the number of PHP files and the number of *relevant* PHP files. Not all PHP files are relevant because many of them simply retrieve data from the server and return them as plaintext. In other words, they don't emit HTML pages or JS code. There is a possibility that JS code can be sent in plaintext and extracted via *eval*() on the client side. However, it's not a common practice for the sake of security.

We define a server side PHP file to be relevant as long as it can emit (part of) a client HTML page. The formal definition is as follows.

(1) If a server page p has <*html*> tags, which means its output can be a valid client page, p is relevant.
(2) If a page p_1 is (transitively) included in another page p_2 that has <html> tags and p_1 has JS strings, p_1 is relevant.

The third column lists the lines of code of *.php* and *.inc* files and the lines of relevant code. Page extraction is only applied to the relevant pages. Column '#page gen' shows the number of valid client pages generated (passing the HTML syntax check). The last column shows the total LOC of the extracted JS, which are subject to our static analysis.

From the data, we can make the following observations. Although there may be many server pages, the relevant ones are few. Furthermore, in the relevant pages, the part with JS strings (our technique would manipulate path conditions to reach those places) are often not guarded by conditionals. In other words, no matter what control flow paths of the server script are taken at runtime, the same JS is likely to be emitted to the resulting client page. This is supported by the small number of extracted client pages presented.

Table 4: Analysis Result.

	Time (seconds)	Type-1 Total	Type-1 FP	Type-2 Total	Type-2 FP
Xhtml chat	3.62	1	0	1	0
AjaxLogin	5.49	1	0	0	0
Tixean chat	5.64	1	0	1	0
Rogozhka chat	7.23	1	0	1	0
Phormer	7.58	1	1	1	1
Quizzy	5.62	3	0	1	0
Ajallerix	9.56	4	2	1	0
Ajax file browser	12.53	7	1	1	0
www.msn.com	4.79	0	0	0	0
Total	—	19	4	7	1

Table 4 presents the final results. *Time* is the average analyzing time for one client page. *Type-1* and *Type-2* stand

for the inconsistency problem and the atomicity violation, respectively. False positives(FP) are also collected. Observe that our technique is quite effective and identifies a set of real bugs in these applications. We manually confirm those bugs.

5.1 Case Study

We have presented two motivating cases for Ajallerix in Section 2. Next, we will present two other cases to demonstrate the effectiveness of our technique.

Inconsistency in Ajax File Browser. The following is a simplified code snippet from Ajax File Browser.

```
@ _js/afb.js

 5  var currentShare = 0;

53  function loadShare(id,path) {
      // cbLoadShare() is ajax response handler
63    getUrl(xmlFile+'?getShare&share='+id+'&path='+escape(path),
            id, cbLoadShare, function(){...} );
70  }

74  function cbLoadShare(xhr) {
86    currentShare = xhr.argument;
91  }

129 function download(path) {
140   window.location = 'index.php?download=' + path + '&share='
                      + currentShare;
145 }
```

Event handler *loadShare*() will be triggered when a shared folder is selected. Event handler *download*() will be triggered upon the download button being clicked. Both handlers use Ajax to send requests. Function *cbLoadShare*() is the response handler for the request made in *loadShare*(). A global variable *currentShare* is defined at line 86 in *cbLoadShare()*, and used at line 140 in a request sent in *download()*.

To trigger the bug, assume folder one is presently shown, the user then selects folder two. Due to network delay, the response may take time. While seeing no response, the user clicks the download button, expecting to download a file in folder one. Due to the non-determinism, *cbLoadShare*() responding for the selection request may get executed before *download*(). In this case, a request with inconsistent parameters is sent to server at line 140: the *path* is a filename in folder one but the value of *currentShare* stands for folder two. A wrong file may be downloaded.

Atomicity Violation in XhtmlChat. Another example is from a chatting plug-in XhtmlChat.

```
@ scripts.js
15 var lastID = -1;

26 function receiveChatText() {
28   httpReceiveChat.open("GET",GetChaturl + '?lastID=' + lastID
                          + ..., true);
29   httpReceiveChat.onreadystatechange = handlehHttpReceiveChat;
32 }

35 function handlehHttpReceiveChat() {
36   if (httpReceiveChat.readyState == 4) {
37     results = httpReceiveChat.responseText.split('---');
39     for(i=0;i < (results.length-1);i=i+3) {
         //inserts the new messages into the page
40       insertNewContent(results[i+1],results[i+2]);
41     }
42     lastID = results[results.length-4];
44     setTimeout('receiveChatText();',4000);
45   }
46 }
```

As shown in the above code snippet, *receiveChatText*() is triggered after sending a message or when a timer set at line 44 fires. It sends out a request at line 28 to the server to retrieve messages that happen after the timestamp *lastID*. In the response handler *handlehHttpReceiveChat*(), the updated messages are inserted to the current page at line 40. A global variable *lastID* is defined in the at line 42 and used in an Ajax request at line 28.

Consider the following execution sequence: (1) an update request R_1 with *lastID*=t is sent; (2) the user sends a message so that another update request R_2 also with *lastID*=t is sent before the response of R_1 has arrived; (3) response to R_1 arrives and *lastID* is defined to $t + x$, meaning the messages in between t and $t + x$ are retrieved and inserted to the current page; (4) response to R_2 arrives and *lastID* is set to $t + x + y$ and the messages in between t and $t + x + y$ are appended. Note that the messages in between t and $t+x$ are duplicated due to the atomicity violation.

5.2 False Positives

Currently, we manually examine reported bugs to confirm they are real ones. We have found some false positives (FP). They may be caused by the following two scenarios.

The first case is that the event handlers that are needed to trigger the problems may never be present in the same client-side web page in reality. However, they may be present in the same page extracted by our server side technique. Recall our method explores all relevant paths including those infeasible ones as our current technique can not reason about the feasibility. Exercising an infeasible path leads to an impossible page. Bugs identified on the page may be FPs. The problem can be mitigated using the symbolic execution techniques in [6, 11] to exclude infeasible paths.

The second case is that although all the event handlers are present on the same page, the failure inducing interleaving is not feasible. For example, an Ajax request is only used to load a page. The response handler is used to initialize that page. In other words, they are supposed to be executed only once, before any other events can be triggered. So interleaving is impossible. In our future work, we plan to analyze the happens-before relations enforced in the code and use them to prune the infeasible interleavings. In practice, this case is what we have observed. All FPs presented in Table.4 are caused by this reason.

6. FIXING THE BUGS

We can avoid the Ajax problems in two ways.

Disabling Relevant Controls. If the use of a shared variable in a request and the corresponding response handler is not avoidable, we can enforce the atomicity of the request and the response in the user layer by preventing events that may lead to accesses to the same variable from happening in between the request and the response. Naively, we can block the user by setting Ajax in the synchronous mode. However, that degrades the user experience. To avoid completely freezing the page, the programmer can disable only the controls that can fire offending requests.

This method can avoid the inconsistency problem. However, atomicity violations are still possible. Please note user controls are not the only way to send requests to server. Others such as the interval timer events (e.g. those generated by *setInterval*()) can too.

Using Conditional Variables. In concurrent programming, besides locks and wait/notify, conditional variables can be used to play the role of synchronization. We can

leverage conditional variables too. A conditional variable can be used to indicate whether the response is handled. If not, another request shares the same variable is not allowed. It has been used in practice by some complex JS programs, although not consistently.

7. RELATED WORK

Web application testing. Server side script testing is increasingly studied lately. Artzi et al. [6] propose a technique called Apollo for finding bugs in PHP scripts. They leverage symbolic execution to generate test inputs so that control flow paths can be covered. The tool can detect PHP faults that generate invalid client pages. Hanfold et al. [11] propose a static symbolic execution technique to identify interface. By analyzing server-side Java code, it can extract parameter combinations sent by clients and their possible values to generate test input suite. Although they may need to manipulate client side controls to provide inputs, the target problem is to locate bugs in server-side scripts. In comparison, we are analyzing problems in client side JS code.

Also, there are many web application testing tools, such as Seleninum, WebKing, and Sahi, that work in a record-and-replay fashion. Testing starts with recording the manual interactions between the tester and the controls on subject interface. Then these techniques either faithfully replay the interactions or heuristically generate combinations of these interactions to test both the client and the server. The fault detection capability depends on the quality of the recorder interactions. They require lots of manual effort. Furthermore, these techniques are not good at the problems we are addressing because they do not consider non-determinism.

Marchetto et al. [13] propose a state-based approach to test Ajax applications. They collect traces of JS execution and construct state machines from the traces. Test cases are generated from the state machines to expose problems in JS code. They look into problems caused by event orders. Non-trivial manual efforts are required in their model construction and refinement. Then Mesbah et al. [14] crawl Ajax applications, simulate user events and infer model automatically. In comparison, our technique is static and focusing specifically on problems caused by asynchronous calls.

JavaScript Analysis. Recent works [21, 18] survey the use of JS, including dynamic features and their impact on security concerns. Several works apply static analysis to identify the vulnerabilities including Drive-by Download and Cross Site Scripting(XSS). Chugh [8] applies information flow analysis to avoid information leak. Wassermann and Su [20] apply taint analysis and string analysis to perform input check in order to detect the XSS vulnerabilities. Guha et al. [10] use static analysis to extract a model of expected user behavior to detect Ajax intrusion. Our analysis is also static but aims at a different problem.

Data Race Detection and Atomicity Violation. There are many existing works on data race detection [7, 19, 16] and atomicity violation [17, 9, 12]. Most of them are designed for threads or processes. The execution model is different from that of JS. They often leverage synchronization primitives. Hence, they are not directly applicable.

8. CONCLUSION

We propose a static analysis that detects non-deterministic problems caused by asynchronous calls through Ajax. The problems we are addressing are data inconsistency that may cause permanent damage on the server side and atomicity violations that may cause runtime exceptions. The analysis is automatic. It handles individual client side pages and server side scripts. Our results show that it is very effective in identifying real bugs in real world applications with reasonable cost.

9. ACKNOWLEDGMENTS

We would like to thank the reviewers for their substantial efforts. This research is supported, in part, by the National Science Foundation (NSF) under grants 0834529 and 0845870. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

10. REFERENCES

[1] T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/.
[2] the open source PHP compiler (phc). http://www.phpcompiler.org/.
[3] JavaScipt usage statistics. http://trends.builtwith.com/javascript .
[4] Ajallerix, a web image gallery. http://developer.novell.com/wiki/index.php/Ajallerix.
[5] A. Aho, M. Lam, R. Sethi, and J. Ullman. Compilers: principles, techniques, and tools (2nd Ed.). *Pearson Education, Inc, 2006*.
[6] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar and M. Ernst. Finding Bugs in Dynamic Web Applications. In *ISSTA'08*.
[7] R. Callahan, J. Choi. Hybrid dynamic data race detection. In *PPoPP'03*.
[8] R. Chugh, J. Meister, R. Jhala, S. Lerner. Staged information flow for javascript. In *PLDI'09*.
[9] C. Flanagan and S. Freund. Atomizer: A Dynamic Atomicity Checker For Multithreaded Programs. In *POPL'04* .
[10] A. Guha, S. Krishnamurthi and T. Jim. Using Static Analysis for Ajax Intrusion Detection. In *WWW'09*.
[11] W. Halfond, S. Anand, and A. Orso. Precise Interface Identification to Improve Testing and Analysis of Web Applications. In *ISSTA'09*.
[12] S. Lu, J. Tucek, F. Qin, Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII*.
[13] A. Marchetto, P. Tonella and F. Ricca. State-Based Testing of Ajax Web Applications. In *ICST'08*.
[14] A. Mesbah and A. Deursen. Invariant-based automatic testing of AJAX user interfaces. In *ICSE'09*.
[15] Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *WWW'05*.
[16] M. Naik, A. Aiken and J. Whaley. Effective static race detection for Java. In *PLDI'06* .
[17] C. Park, K.Sen. Randomized active atomicity violation detection in concurrent programs. In *SIGSOFT '08/FSE-16*.
[18] G. Richards, S. Lebresne, B. Burg, J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI'10*.
[19] S. Savage, M. Burrows, G. Nelso, P. Sobalvarro and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. In *ACM Trans. Comput. Syst. 15(4): 391-411 (1997)* .
[20] G. Wassermann and Z. Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *ICSE'08*
[21] C. Yue and H. Wang. Characterizing insecure javascript practices on the web. In *WWW'09*.