Dynamically Detecting DOM-related Atomicity Violations in JavaScript with Asynchronous Call

Dezhi Wang, Lei Xu*, Baowen Xu
State Key Lab. for Novel Software Technology
Dep. of Computer Science and Technology
Nanjing University, 210023, China
mg1433058@smail.nju.edu.cn, {xlei, bwxu}@nju.edu.cn

Weifeng Zhang
School of Computer
Nanjing University of Posts and Telecommunication
Nanjing, China
zhangwf@njupt.edu.cn

***Abstract*—In Web applications, atomicity violations caused by AJAX (Asynchronous JavaScript and XML) generate non-determinism and inconsistency. This paper introduces a dynamical detecting approach for atomicity violations in AJAX. Implemented based on Jalangi, an existing instrumentation framework, our technique monitors the execution of a Web application and tracks the Function Callback Flow to find atomic regions by taint analysis. Next, we build a dynamic event model, so as to precisely record the related DOM elements in call and callback steps with low overhead. Then we develop an event-based algorithm to acquire the results, which are classified into different severity levels (benign or harmful). Finally, we conduct an empirical experiment on a subset of Alexa top-ranked websites. Our tool detects 175 DOM related atomicity violations in which 48.6% of violations are identified as harmful after manual inspection.**

Keywords—AJAX; atomicity violation detection; dynamic analysis; function callback flow; event-handler

I. INTRODUCTION

JavaScript asynchronous invocation (AJAX) in browser side not only makes a single-threaded Web application can satisfy the throughput of the multithreaded program, but also ensures the front-end users maintain the initiative of the next operation while the previous event is processing. Currently, the wide usage of AJAX is based on the processing of an asynchronous call from the client or the server that requests for services: completing the data exchange through the background and the database [2]. Despite the effectives of AJAX, its careless usage can bring potential incorrectness issues to the whole Web application [5].

When a request action occurs in two atomic operations that should be continuous, other operations perform a request or response inside the atomic region and the corresponding relationship between operating sequences appears uncertain. This situation called as atomicity violation produces a race condition outside, which generates an explicit impact to the user or system [4, 7]. Considering the potential damages of atomicity violations, users should be forbidden to use the same memory locations across different AJAX calls.

Recent studies [3] find that DOM-related faults account for 88% of Incorrect Method Parameter faults and the majority (65%) of JavaScript faults are DOM-related, and over half faults are atomicity violations. Therefore, in this paper, we focus on the DOM-related atomicity violations and we also make the classification on the detected results based on the DOM APIs used [9].

In order to detect atomicity violations in AJAX-based Web applications, we firstly track the Function Callback Flow (FCF, constructed from the flow of related callback functions) to find atomic regions by taint analysis. Next, we build a dynamic event model, Aevent, so as to precisely record the related DOM elements in call and callback steps with low overhead. Then based on the definition of atomicity violation, we develop an event-based algorithm to acquire the results which are classified into different severity levels (benign or harmful, and harmful means last for a while and have visible or serious effects on the Web application). Finally we implement our experiment on the subset of Alexa top-ranked websites (over 300 websites).

In summary, we make the following contributions.

- We propose an approach for dynamically detecting AJAX-related atomicity violation in front-end Web applications
- We design the model of Function Callback Flow (FCF) to present the atomic regions during the execution of Web applications.
- We implement Aevent to record related information which can reduce the overhead and show the stage of current execution. We tested more than 300 websites downloaded from Alexa top-ranked websites for atomicity violations with our tool AVChecker.

The rest of this paper is organized as follows: Section 2 presents the preliminary about AJAX and atomicity violations, together with the challenges. Section 3 gives an overview of our method. Section 4 introduces the implementation details. Section 5 evaluates our approach empirically. Section 6 presents the case study. Section 7 discusses the related work. Section 8 concludes this paper.

II. MOTIVATING EXAMPLE

In this section, we briefly introduce the working mechanism of AJAX, and then we show a concrete example of atomicity violations and the challenges to detect them.

* Corresponding author

978-1-5090-4517-4/16 $31.00 © 2016 IEEE
DOI 10.1109/SATE.2016.14

A. Example of AJAX Process

Figure 1 indicates the code snippet for an AJAX operation. XMLHttpRequest (XHR) object is the core, whose properties, methods and event handlers are used during the AJAX call.

```
01 <html>
02 <head>
03    <meta charset='utf-8' />
04    <title>demo</title>
05    <script >
06    function ajax1(){
07       var xmlhttp=null;
08       var data =    document.getElementById('data')
09       if (window.XMLHttpRequest){
10          xmlhttp=new XMLHttpRequest();
11       }else if (window.ActiveXObject){
12          xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
13       }
14       xmlhttp.onreadystatechange = function (){
15          if (xmlhttp.readyState==4 && xmlhttp.status==200){
16             document.getElementById('show').innerHTML = 'ajax01'
17          }
18       }
19       xmlhttp.open("GET",...,true);
20       xmlhttp.send(data);
21    }
22    function ajax2(){
23       document.getElementById('show').innerHTML = 'ajax02'
24    }
25       </script>
26 </head>
27 <body>
28    <input type='text' id='data' name ='data'>
29    <input type='button' id='bt1' onclick='ajax1()'>
30    <a id='a1' href='javascript:ajax2()'>
31    <div id='show'></div>
32 </body>
33 </html>
```

Figure 1. Code snippet for an AJAX operation

In Figure 1, there are two buttons (bt1 and bt2), and each button is bound to a JavaScript function (ajax1 and ajax2). The whole process of an AJAX realization is divided into the following steps: Start, Call, Callback, End.

Start: Clicking the button (*id* = '*bt1*') in the HTML page to trigger the event bound to the *onclick* event handler.

Call: Calling the related JavaScript function (*ajax1*). During the execution of function body, *ajax1* news an XHR object to realize the AJAX. XHR object is hanged up after sending an HTTP post to server-side and waits for the response from server-side.

Callback: When the XHR object receives the message from the server-side, it checks the values of *readyState* and *status*. If conditions are satisfied, then it calls the callback function of *onreadystatechange* and executes the related function.

End: After the response result displays in the HTML page, an AJAX comes to an end.

B. Atomicity Violation

Since JavaScript applications always work in a single thread, atomicity violations occur due to the conflictions of two JavaScript functions, not two threads. And the response from server-side will arrived at the client-side in an uncertain time for the delay of network. Therefore, the execution of the callback function might have conflicts with others, even worse, if the callback function and others have operations on the same atomic region, an atomicity violation occurs.

Atomicity violations often cause the unordered atomic region. If the region is a DOM element, it may cause a wrong appearance or data, even worse, a program crash [10]. Figure 2 shows the example of an atomicity violation: *ajax2*'s callback executes before *ajax1*'s callback, while the supposed result is firstly executing *ajax1*'s callback, which may lead to the runtime exceptions.

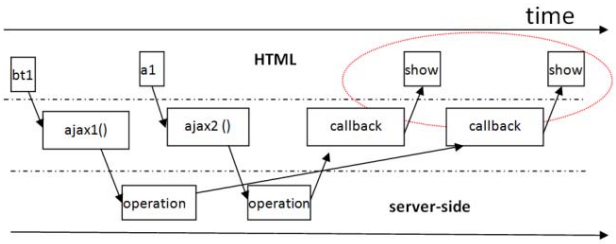

Figure 2. An example of an atomicity violation

C. Challenges

In order to detect the atomicity violations in AJAX Web applications, we need to acquire the atomic region and the sequences of JavaScript functions, and then ensure that the asynchronous function call be executed in a right order within the atomic region. Since the concurrency bugs in Web applications happen between JavaScript functions, especially those functions called in the asynchronous way, traditional detection methods for multi-thread programs cannot be directly used, and there are several challenges.

CH1: Callback function identification

JavaScript functions executed during the callback stage are the related callback functions. We cannot get the whole callback stage only rely on a callback function, for a callback function may have relations with other JavaScript functions during the callback stage. So, we need to record the related functions to track the execution of the callback function. However, since functions can be passed as an argument between other functions in JavaScript, we cannot use Control Flow Graph (CFG) to acquire the whole chain of callback function execution.

CH2: AJAX-related atomicity violation recording

Executing a Web application contains the operations on HTML pages, CSS and the database in server-side, and the information is trivial and repeatable. Even more, if the application imports some JavaScript libraries, it will cause a big overhead to record the traces. Besides, AJAX-related atomicity violations are often caused by network delay, thus we need extra efforts to record the states of XHRs, which makes the detection even more complicated than before.

III. Technique Overview

We develop a dynamic analysis tool AVChecker to detect atomicity violations in AJAX-based Web application based on the Jalangi framework [8]. Using AVChecker, when users make operations on Web application, they will immediately get a result about whether the current operations contain a real or potential atomicity violation.

As proposed in CH1, it is difficult to identify whether the whole functions are executed in callback stages or not. Therefore, we give the definition and example of Function Callback Flow based on taint analysis.

A. Function Callback Flow

Firstly, we make a taint on the callback function body of an AJAX before running the program, then we record the executed functions by Jalangi. When we meet a tainted function, it means this function is a callback function. Furthermore, we also taint such functions that have relations with this tainted function, so as to acquire the whole callback execution chain, namely, the Function Callback Flow (FCF).

Function Callback Flow (FCF) is a set, which collects the JavaScript functions and JavaScript variables in order during the whole chain of a callback function call.

We use a graph G <*Nodes*, *Edges*> to denote FCF, in which *Nodes* represent the related functions, and *Edges* denote the relations of functions.

Nodes = <*TID*, *F*>, in which *TID* is the unique number of a function and *F* is the function body. *Nodes* in FCF are the functions executed during the callback stage, and they could be JavaScript variables or anonymous functions.

Edges = <*TIDs*, *R*>, in which *TIDs* is a pair of *TID*s that represents a pair of functions and *R* denotes the relation between two functions during the execution. There are three types of relations, when considering the JavaScript features.

(1)*Function call*: If a function is called directly during the callback function body, it is a function call.

(2)*Method use*: Sometimes a JavaScript object will be created during a function, and some attributes of the created object are also functions. When the function-type attribute of object is used, the related function will be executed, and it is called as method use.

(3)*Argument pass*: Some functions will be passed to another function as an argument and will be called during the execution of the function that the argument is passed to, and it is called as argument pass.

B. Event-based Model

In general, the detection methods for atomicity violations check the read-write operations in JavaScript functions. But the execution of JavaScript events is concerned with trivial operations in CSS and HTML elements. And it induces extra overheads to record traces with extra JavaScript libraries.

To resolve CH2, we put forward a dynamic event model, Aevent, so as to give each event an attribute to show which AJAX step the current state is in.

Aevent is a JavaScript object, and it has three attributes *handlerSource*, *cbHandlerSource* and *XHR*.

(1)*handlerSource* is an array for DOM elements which are operated by JavaScript function during the call stage;

(2)*cbHandlerSource* is an array to represent the related DOM elements manipulated during the callback stage;

(3)*XHR* represents the related XMLHttpRequest attribute, and it records attributes, methods and event handlers of XHR objects. If the attributes change, XHR updates itself to show the stage of current execution.

C. Atomic Region Classification

We define five types of atomic regions about the DOM elements in HTML page, and we classify the first four types as harmful according to the affection on both client-side and server-side [9, 10], since these bugs may lead to unsafe situations (primacy leak, finance loss, etc.) or display the disordered visible appearances.

(1) Global-related: JavaScript can manipulate the global object of an HTML page, such as using the attribute *location* of *window* to change the current URL path.

(2) Cookie-related: The browser can save variables in cookie during runtime, and JavaScript can manipulate the related variables in cookie by API *document.cookie*.

(3) Form-related: Server-side programs need to acquire the information from HTML page, such as *username* and *password* during an AJAX-based login process.

(4) Node-related: JavaScript can use related DOM APIs to create or remove DOM elements, such as *createElement* or *removeChild*.

(5) Common: Such APIs as *getElementById* have no influence to the GUI elements or critical variables, and the violations generated from them are considered as benign.

IV. IMPLEMENTATION

We realize our approach based on tool Jalangi and the details of AVChecker are illustrated as follows.

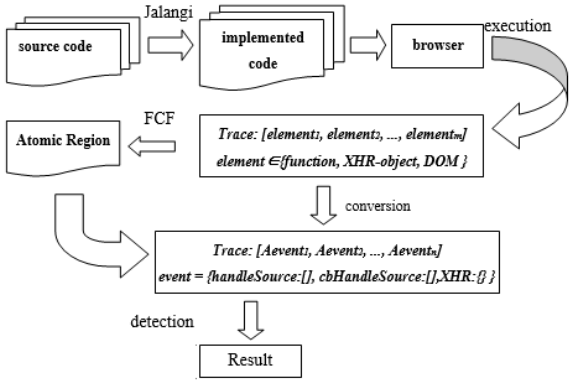

Figure 3. System Overview

Figure 3 shows the system overview. AVChecker instruments source codes of Web applications by Jalangi, and then collects the execution information in the browser; next it uses the FCF to acquire the atomic regions. AVChecker also makes a conversion on the collected traces and then it detects atomicity violations with Aevent.

A. Trace Collection

Since JavaScript is a dynamic language, and some codes will be generated during the runtime (such as *eval*), static analysis is not enough to handle this situation. So we choose a dynamic way to analyze the program and to meet the CH1.

When users make operations on Web applications, Jalangi can provide a trace array for testers through inserting the related JavaScript codes. In this way, we can record the user operations exactly during the execution. The collected trace array is such an array that each element is an operation of a JavaScript statement, and it can be a function, XHR-object or DOM element

Function: Some of the JavaScript functions are denominative and others are anonymous. Since JavaScript function executed without interrupted, the attribute type of

Function represents the entrance or exit of the function. By comparing the current function with the callback of recorded XHR object, we can determine whether a function is a callback function. Data structure of Function is shown as:

Function: {type: STRING, name: STRING, body: STRING}

XHR-object: XMLHttpRequest object is the key object in AJAX. During the whole execution, we need to record the related attributes and methods of an XHR. Data structure of XHR is shown as:

XHR-object: {_id: NUMBER, attribute: OBJECT, value: STRING}

DOMElement: Since we focus on the atomic region located in HTML page, we need to record the operations of API used by JavaScript to manipulate the DOM elements. We record the arguments passed to these APIs. Data structure of DOMElement is shown as:

DOMElement: {_id: NUMBER, api: STRING, args: ARRAY}

B. Atomic Region Identification

Firstly, we look through the source code of the Web application and taint all the callback functions of the event listener, for the further callback stage identification;

Secondly, we search the tainted JavaScript functions to locate the entrance of a related callback and use FCF to identify the whole callback execution respectively;

Thirdly, we classify all DOM elements in the trace into two arrays: FCF-inner, FCF-outer. If a DOM element is operated during a chain of FCF, we set it as FCF-inner; otherwise, we set it as FCF-outer;

Finally, we achieve atomic regions by two rules:

Rule 1: If both FCF-inner and FCF-outer have the same DOM element, which represents current function operating a DOM element will also be operated by a callback function in the future, then this DOM element is an atomic region;

Rule 2: If a DOM element shows in the FCF-inner more than once, which means more than one callback function has operations on this DOM element, then this DOM element is an atomic region.

C. Trace Conversion

Algorithm 1 is presented for trace conversion: traverse the original trace into Aevent Trace, based on element type.

XHR: if the current element is an initialization of XHR object, function *createNewAevent(Ei)* is used to create an Aevent model with current XHR object (Line 4-6), and then update Aevent when the attributes of this XHR object are met (Line 7); if the current element is an event handler of XHR object, the bound JavaScript function is pushed into a callback function list *cbList* (Line 8-9).

DOMElement: if the current element is a DOMElement, a temporal list *tmp* is recorded (Line 10-11).

Function: if the current element is a JavaScript function and there is an Aevent model, the collected DOMELements list is put into the handlerSource of current Aevent and Aevent is pushed into the *AT*(Aevent Trace), after that the current Aevent is deleted (in Line 13-16); if there is no Aevent model, *isCBFunction(cbList, Ei)* is used to check if the current function is the callback function of recorded XHR objects, if true, *getAeventByXHR(AT, Ei)* is used to acquire the related Aevent model from AT, and the cbHandlerSource of Aevent is set as the collected DOMELements list, after that the modified Aevent model is updated into the *AT* and *tmp* is deleted (in Line 17-21).

Algorithm 1 : Trace Conversion

Input *: Trace Array T = [E1,E2,...,Em]*
Output: *Aevent Trace Array AT = [A1,A2,...,An]*

```
1.  AT← [] ,cbList =[],tmp=[]
2.  foreach Ei ∈T do
3.    switch(Ei.type)
4.      case XHR
5.          If XHR. initialize then
6.              Aevent ← createNewAevent(Ei)
7.          updateAevent(Aevent, Ei)
8.          If XHR.eventhandler then
9.              cbList.push(Ei)
10.     case DOMElement
11.         tmp.push(Ei)
12.     case Function
13.         If Aevent then
14.             Aevent.handleSource ← tmp
15.             AT.push(Aevent)
16.             delete Aevent
17.         Else If isCBFunction(cbList, Ei) then
18.             Aevent← getAeventByXHR(AT, Ei)
19.             Aevent.cbHandleSource ← tmp
20.             update(AT, Aevent)
21.         delete tmp
22. return AT
```

D. Atomicity Violation Detection

In Algorithm 2, we take the *AT* (Aevent trace) and *AR* (atomic region) as input, and make the following operations for each event in T.

Algorithm 2 : Atomicity Violation detection

Input *: Aevent Trace Array AT = [A1,A2,...,Am] ,AR(Atomic Region)*
Output: *Result (R, S)*

```
1.  Result← {R:'',S:''}
2.  foreach Ai ∈AT do
3.    foreach Aj ∈AT ,j<i do
4.      If checkTarget(Ai, Aj) then
5.          Result.R ← Warning
6.          Result.S ← getType(Ai, Aj)
7.      tmp1 ←  (Aj.handlerSource ∪ Ai.cbHhandlerSource) ∩ AR
8.      tmp2 ←  (Aj. cbHhandlerSource ∪ Ai.cbHhandlerSource) ∩ AR
9.      If tmp1 or tmp2  then
10.         If isSafeState(Ai, Aj) then
11.             Result.R ← Warning
12.             Result.S ←getType(Ai, Aj)
13.         Else
14.             Result.R ← Error
15.             Result.S ←getType(Ai, Aj)
16.     Else
17.         Result.R ← Safe
18. return Result
```

The i^{th} element in AT(*Ai*) is compared with elements before it one by one (Line 2-3). Next, if two events are triggered by the same DOM element (Line 4), then the two events may have atomicity violations, and necessary information is recorded (Line 5-8). Then *isSafeState*(*Ai*, *Aj*) is used to check the current state: if it is safe, it contains potential atomicity violations (Line 10-11); otherwise, it contains a real atomicity violation (Line 13-14); else it contains no atomicity violation (Line 16-17). *getType* (Line 6, 12, 15) is used to take two events' operation type of DOM elements.

Table 1. Program characteristics and detection results

Web sites	Program Characteristics				Trace				Conversion					Result						
	HTM	JavsScript			XHR	DOM	FUN	Total	Event	Rate	FCF-	FCF-	Atomic	Warning	Error	Type				
	Size *	Num	Size	Libs					Trace		outer	inner	Region			global	cookie	form	node	common
www.hao123.com	2.3	14	0.26	6	9	13173	3994	17313	51	0.3%	209	3	2	3	2	0	2	0	0	3
www.milliyet.com.tr	6.45	25	1.15	10	39	2394	1611	4044	16	0.4%	81	26	12	3	4	1	2	0	1	3
www.welt.de	7.72	53	3.85	6	296	19449	4156	23901	311	1.3%	83	30	3	4	3	0	1	0	1	5
www.radikal.com.tr	3.71	25	1.49	4	85	6056	2343	8484	648	7.6%	105	122	11	6	5	1	2	3	1	4
edition.cnn.com	6.05	26	2.57	9	181	4150	2512	6843	22	0.3%	37	3	1	1	1	0	0	0	0	2
www.gazetta.it	2.84	37	1.79	6	110	5738	836	6684	27	0.4%	65	25	12	15	9	2	3	2	3	14
wireless.att.com	6.75	72	5.77	17	105	2644	1236	3985	37	0.9%	113	44	2	2	2	0	1	0	0	3
www.aljazeera.net	3.2	32	1.92	5	197	1738	922	2857	25	0.9%	30	28	7	2	7	1	2	0	1	5
www.huffingtonpost.com	3.61	30	1.24	6	137	10117	7047	17301	266	1.5%	65	8	2	2	2	0	1	0	0	3
www.bankofamerica.com	0.82	6	0.5	0	88	1782	241	2111	9	0.4%	56	507	12	11	5	2	2	4	2	6
www.chase.com	1.46	17	0.52	7	55	955	271	1281	3	0.2%	38	15	3	4	2	1	1	0	0	4
www.softonic.com	2.58	27	1.32	5	77	441	1010	1528	20	1.3%	22	80	16	13	4	2	3	2	2	8
www.etsy.com	1.93	9	0.8	4	25	1267	3152	4444	19	0.4%	89	10	4	3	4	1	1	0	0	5
www.espncricinfo.com	1.67	8	0.78	3	157	4197	2239	6593	17	0.3%	70	16	2	4	2	0	0	2	0	4
www.zol.com.cn	2.12	18	0.65	3	19	2273	867	3159	50	1.6%	72	84	13	8	8	1	3	4	2	6
www.iqiyi.com	5.36	24	2.24	5	267	23854	16368	40489	42	0.1%	855	18	2	3	1	0	1	0	0	3
libero.it	1.24	10	0.335	0	35	9670	2185	11890	1021	8.6%	190	2	1	2	1	0	1	0	0	2
www.rambler.ru	1.42	31	0.98	10	60	1124	1057	2241	77	3.4%	50	286	8	4	1	0	1	0	1	3
www.adclickxpress.is	0.68	17	0.5	14	145	2068	105	2318	9	0.4%	11	9	2	1	2	1	1	0	0	1
extratorrent.cc	1.02	8	0.2	7	40	5047	8050	13137	4	0.0%	5	66	3	2	1	0	1	0	0	2
www.t-online.de	3.02	37	1.07	8	92	12113	16814	29019	86	0.3%	82	26	6	7	4	2	4	0	2	3
www.terra.com.tr	1.49	10	0.24	5	174	9712	2697	12583	37	0.3%	56	13	2	3	2	1	2	0	1	1
Total	67.44	536	30.18	140	2393	1E+05	79713	2E+05	2797	1.3%	2384	1421	126	103	72	16	35	17	17	90

V. EXPERIMENT

We implement our experiment in a browser using a proxy and make the on-the-fly instrumentation based on Jalangi2 to require the trace, and then detect atomicity violations with AVChecker. The machine used for running all experiments has a 3.4GHz quad-core CPU and 4GB RAM, and OS is Linux ubuntu-14.04.1-amd64.

We tested over 300 websites from the Alexa's top-ranked websites. Considering the factor of network delay, we test a web site for 10 times with the same operation and record the detected results. Among them, over 100 web sites are confirmed to have atomicity violations, and we select 22 websites that have atomic violations stably. Table1 shows the basic information about these websites and the results detected by AVChecker, listed as the benchmark.

A. Experiment results

In Table 1, Columns 2-5 show the scale from the view of HTML and JavaScript respectively. Columns 3-4 list the number and the size of JavaScript files, besides, column 5 (Libs) records the used JavaScript libraries. Columns 6-9 show the elements in the collected trace: XHR, DOM and FUN. Column 10 shows the size of the converted traces; and Column 11 represents the scale of converted trace when comparing to original trace, the conversion reduces the size into 1.3%, which greatly cut down the recording overhead. Columns 12-13 show the number of FCF-inner and FCF-outer and Column 14 denotes the atomic regions during the current execution. Columns 15-21 represent the results of our detection. Columns 15-16 record the warning and error results while Columns 17-21 show the types of the detected results, including global, cookie, form, and node (considered as harmful bugs), and the sum of them is 85.

JS libraries (jQuery or zepto) or Front-end frameworks (React or Angular) are widely used, and the analysis will introduce a high overhead. However, AVChecker uses FCF to reduce the overhead, and by recording the entrance and exit of a JavaScript function and emitting the detail of calling APIs from external JS libraries, it reduces the size of origin collected trace to 1.3% on average while keep the important information for further detection.

B. Limitations

During the implementation of our analysis, we acquire the detection results based on the dynamic execution of Web applications. Since a single execution cannot cover all JavaScript functions in one Web application, the detection may miss some cases in a certain degree. Namely, due to the dynamic natural of our approach, we cannot guarantee the soundness of our analysis. There are other non-deterministic factors that may affect the coverage of all the related bugs (e.g., network delay, and random operations in programs or user interactions).

```
<!-- script.js -->
01 simpleUpdate = function () {
02   setChatStatus('updating...');
03   xml_httpPost('/chat_update.php','',updateChat);
04 },
05 setChatStatus = function (str) {
06   getEl('chat_timer').innerHTML = str;
07 },
08 chatStatusUpdate = function () {
09   if (sec === 0) {
10         setChatStatus("<img src='...'onclick='simpleUpdate();'/>");
11         return false
12   }
13   setChatStatus(sec + 'sec');
14   chattimer = setTimeout(chatStatusUpdate, 1000);
15 },
16 updateChat = function (str) {
17   chatStatusUpdate();
18 },
```

Figure 4. Code snippet from site *extratorrent.cc*

VI. CASE STUDY

In **extratorrent.cc** (http://extratorrent.cc/), we detected atomicity violations by AVChecker. The key part of the code is shown in Figure 4. DOM tags are simplified to contain only relevant attributes. The img tag with *chat_timer* id is

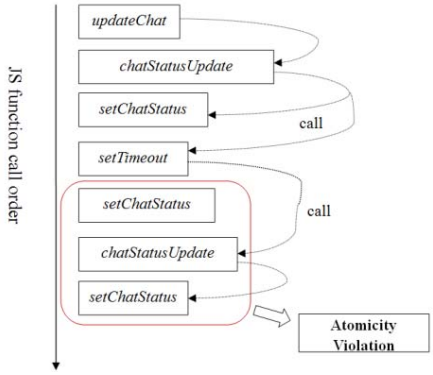

Figure 5 Function call stack of Atomicity Violation

registered as onclick event handler *simpleUpdate*.

In Figure 5, when *setTimeout* calls *chatStatusUpdate*, a function call to *setChatStatus* is executed unexpectedly due to the delay of network. This causes an atomicity violation on the *div* element with id *chat_timer*. The state-of-the-art tool EventRacer fails to detect this concurrency bug, for the triggering condition is changing the value of the innerHTML attribute of the DOM element inside the atomic region.

VII. RELATED WORK

A. Data Race Detection

Paper [6] proposed tool EventRacer and the notion of race coverage. It can be used to quickly expose ad hoc synchronizations, and then it presents a dynamic analysis algorithm that efficiently computes races in event-driven programs by keeping the width of its vector clocks much smaller than the standard approach. Paper [5] proposed a new view of benign and harmful data races in JavaScript Web applications, and argued that harmful races should be the primary focus of analysis tools, and it presented a lightweight exploration algorithm for finding data races in runtime traces of JavaScript programs, using the static analysis on source code of Web application to guarantee the coverage. These methods will cause high overhead, and overlooking the relation between elements and mechanism of event execution will produce false positives.

B. Atomicity Violation Detection

Papers [1] constructed a framework focused on the target atomic regions to detect atomicity violations during the execution, although they might lose the ability to catch specific interleavings which leaded to atomicity violations since they only focused on atomic regions, they still could be sufficient effective to detect atomicity violations through pattern matching or pointer supervision. Papers [10] demonstrated the atomicity violations caused by the asynchronism and proposed a static program analysis to detect such bugs in Web applications. But these methods need extra computation and cause high overhead.

VIII. CONCLUSION AND FUTURE WORK

We presented a dynamic detection technique for atomicity violation in XHR-based Web applications. Our method uses Jalangi to collect event-based traces, and constructs the analysis models of Function Callback Flow and Aevent. We have confirmed the advantages of our method from the Alexa's top-ranked websites.

In the future, we will extend our approach to cover more mechanism including ES6. Then predictive analysis for atomicity violations and concurrency bug replay and fixing are also in our research agenda.

ACKNOWLEDGMENT

This work is partially supported by the National Natural Science Foundation of China (Grant Nos. 61272080, 91418202, 61403187). All support is gratefully acknowledged.

REFERENCES

[1] C.Flanagan, and S. N.Freund, "Atomizer: a dynamic atomicity checker for multithreaded programs", Proceedings of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 04), Venice, 2004, 175-190.

[2] S. O. Frolin, and K.P.I. Mesbah, "AutoFLox: An Automatic Fault Localizer for Client-Side JavaScript", Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST 12), Amsterdam, 2012, 31-40.

[3] S. O. Frolin, K. Pattabiraman, and B. Zorn, "JavaScript Errors in the Wild: An Empirical Study", Proceedings of the 22nd IEEE International Symposium on Software Reliability Engineering (ISSRE 11), 2011,100-109.

[4] A. Mesbah, and A. Deursen, "Invariant-based Automatic Testing of AJAX User Interfaces", Proceedings of the 31st International Conference on Software Engineering (ICSE 09), Vancouver, 2009, 210-220.

[5] E. Mutlu, S. Tasiran, and B. Livshits, "Detecting JavaScript Races that Matter", Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE 15), New York, USA, 2015, 381-392.

[6] B. Petrov, M. T.Vechev, and M. Sridharan, "Race Detection for Web Applications", Proceedings of the 33th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 12), Beijing, China, 2012, 251-262.

[7] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An Analysis of the Dynamic Behavior of JavaScript Programs", Proceedings of the 34th annual ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI 10),Toronto, 2010, 1-12.

[8] K. Sen, S. Kalasapur, and T. Brutch, "Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript", Proceedings of the 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 13), Saint Petersburg, 2013, 488-498.

[9] A.Taly, U. Erlingsson, and J. Mitchell, "Automated Analysis of Security-Critical JavaScript APIs", Proceedings of the 32nd international conference on IEEE Symposium on Security and Privacy (SP 11), Oakland, 2011,363-378.

[10] Y. H. Zheng, T. Bao, and X. Y. Zhang, "Statically Locating Web Application Bugs Caused by Asynchronous Calls," Proceedings of the 20th International World Wide Web Conference (WWW 11), Hyderabad, 2011, 805-814.