# D-ForenRIA: A Distributed Tool to Reconstruct User Sessions for Rich Internet Applications

Salman Hooshmand[1], Muhammad Faheem[1],
Gregor V. Bochmann[1], Guy-Vincent Jourdan[1], Russell Couturier[2], Iosif-Viorel Onut[3]

[1]EECS, University of Ottawa, Ottawa, Ontario, Canada
[2]Chief Technology Officer Forensics,IBM Security, USA
[3]Principal R&D Strategist, IBM Centre for Advanced Studies, Canada
{shooshma, mfaheem}@uottawa.ca, {bochmann, gvj}@eecs.uottawa.ca
russ.couturier@us.ibm.com, vioonut@ca.ibm.com

## ABSTRACT

Rich Internet Applications (RIAs) which use JavaScript and Ajax have become the norm for modern Web applications. However with RIA, the reconstruction of user-interactions from recorded HTTP logs is a new and challenging problem. We present *D-ForenRIA* a distributed tool for session-reconstruction for RIAs. *D-ForenRIA* provides detailed information about user actions including DOM elements involved and user-inputs provided. *D-ForenRIA* incorporates novel techniques to order candidate user-interactions based on DOM features and knowledge acquired during session reconstruction. In addition, using several browsers concurrently makes the system scalable for real-world use. The results of our evaluation on several RIAs show that *D-ForenRIA* can efficiently reconstruct use-sessions in practice.

## Keywords

User-Interactions Reconstruction, Rich Internet Applications, Traffic Replay, HTTP Traces

## 1. INTRODUCTION

*"Rich Internet Applications"* (RIAs [13]) use JavaScript and Ajax [15] to create smooth and responsive browser-based Web applications, providing end-users with an experience similar to "desktop" (i.e., non-Web) applications. RIAs have become the norm for modern Web applications. For example, Google has adopted RIA technologies to develop most of its major products (Gmail, Google Groups, Google Maps, etc.) In fact, we have evaluated the top 100 Web sites from alexa.com and found 87 of them use Ajax to communicate with the server-side scripts[1].

During a user session, each user-browser interaction with a RIA generates a series of HTTP requests. The server responds to these requests and the browser changes the state of the application by processing these responses. The set of exchanged messages during a session is called *"User-Trace"* or *"User-Log"*.

[1]http://ssrg.eecs.uottawa.ca/sr/alexaajax.htm

These traces can be captured (entirely or partially) on the Web server (or on the network using a proxy) for further analysis. A proxy serves as an intermediary between the client and the Internet and thus can also be used to capture traffic. These HTTP logs are considered important tools for Web developer and administrators. These logs are essential in Web analysis and can be used to reconstruct the user interactions from a given session. We call this *"Session Reconstruction"*.

Session Reconstruction has several applications, including in Web usage mining or in forensic analysis. For instance, user logs contain important information about usage patterns and the behavior of the users. Therefore, *Session Reconstruction* can be used for Web usage mining to analyze users' behavior on a particular Web site. In Web usage mining, the extracted behavior of users can be used to improve the application and better understand the user's needs. In forensic analysis, if the Web site administrator has a recorded traffic of a malicious user session, then *Session Reconstruction* can be used to find out how the user interacted with the application to conduct the intrusion.

The previous techniques for session reconstruction using HTTP traces focus on *traditional* Web applications (i.e., inter-linked static HTML pages accessible through hyperlinks). In these applications, users usually navigate between pages by following links and the linked page is explicitly mentioned inside attributes of the **href** element. An assumption often made is that each user action navigates the application to a new page with a new URL. In reality, however, RIAs do not meet this assumption and requests are often generated dynamically by JavaScript execution. Hence, when the Web application is a modern RIA, "Session Reconstruction" is not trivial. Such Web applications are based on "asynchronous" communications which allow the client to ask for specific data rather than a whole web page. Thus an HTML page can be partially updated without changing the URL. Therefore, determining the user interactions from a set of asynchronous calls would be extremely difficult and time consuming. To deal with this challenge, a technique which efficiently and automatically finds user actions during a session is required.

Manual reconstruction is extremely difficult, time consuming, and application dependent (i.e., a set of patterns may vary for different RIAs). Such solution is not realistic for large scale applications. We need a tool that can perform an *automated* and *complete* reconstructions of user interactions. Existing solutions require to either instrument the user's browser, or the Web application itself [5, 6], or are based on predefined patterns of actions [21]. Such solutions are inadequate in the real-world contexts. For example, the administrator of the Web application learns that a hacker found and exploited a vulnerability a few months back. For forensic analysis the only available input is the server-generated logs. This task is often performed off-line without access to the initial Web server due to security concerns of forensic analysis [11].

In order to address above challenges, we propose *D-ForenRIA*, a tool that helps the administrator to recover the details of the user-interactions after the fact using only the available logs. *D-ForenRIA* is an improvement over a previous version of the tool, *ForenRIA* [7]. *D-ForenRIA* uses a collection of browsers working concurrently to reconstruct the session more efficiently, and the reconstruction techniques have been improved. We demonstrated an early version of *D-ForenRIA* [17] without any form of cost estimation, and the evaluation leads to very good scalability. The reader is invited to see [7] for more details on the old system (i.e. ForenRIA). In addition, a companion site has been setup at http://ssrg.site.uottawa.ca/sr/demo.html where videos and further information is being made available.

The main contributions of this paper are:

- We propose *D-ForenRIA*, a distributed system which automatically reconstructs the user-interactions for RIAs using only previously recorded user-session logs as input.
- We propose learning mechanisms during reconstruction which are based on the history of generated requests during event executions.
- We empirically evaluate the efficiency of our approach on five RIAs. The results show that our method can efficiently reconstruct user interactions from previously recorded HTTP traffic.

The rest of this paper is organized as follows: In Section 2, we first model RIAs as a finite state machine (FSM) and then define the session reconstruction problem. In Section 3, we present the tool, *D-ForenRIA*, with detailed description of its general architecture and session reconstruction approach. We present some evaluation results that highlights the effectiveness of *D-ForenRIA* in Section 4. Then we briefly discuss the state of the art in Section 5. Finally, we present our concluding remarks and future directions in Section 6.

## 2. PROBLEM DEFINITION

Our input is a set of previously recorded HTTP traffic coming from the user session. The output we want to produce is the set of user-browser actions including user-input data, DOM of the pages as seen by the user during the session, and screenshots of each recovered state.

The log is composed of a list of $< req_i, res_i >$ pairs where $req_i$ is the $i^{th}$ request and $res_i$ is the corresponding response. More formally, we can describe the problem using finite state machine (FSM) notation. An FSM is described by a quintuple $(I, O, S, s_0, \delta, \lambda)$ where:

1. $I$ is the set of possible user-browser interactions with the RIA (Inputs to the FSM).
2. $O$ is the set of possible generated HTTP requests and responses.
3. $S$ is the set of all states of the RIA reached by the user during the session.
4. $s_0$ is the initial state of the RIA.
5. $\delta : S \times I \to S$ is the transition function where $\delta(s_i, a_j)$ refers to the next state of the application after execution of $a_j$ in $s_i$.
6. $\lambda : S \times I \to O$ is the output function, where $\lambda(s_i, a_j)$ refers to the generated requests and responses by executing action $a_j$ at state $s_i$.

$\lambda$ and $\delta$ functions can be extended to accept a sequence of inputs (here a sequence of actions). For example $\delta(s_0, a_1 \dots a_n)$ refers to the state reached by the application after applying the sequence of actions $a_1 \dots a_n$ from the initial state $s_0$. After executing each action $a_i$ at state $s_{i-1}$, the application reach the next state $s_i$ (i.e., $\delta(s_{i-1}, a_i) = s_i$).

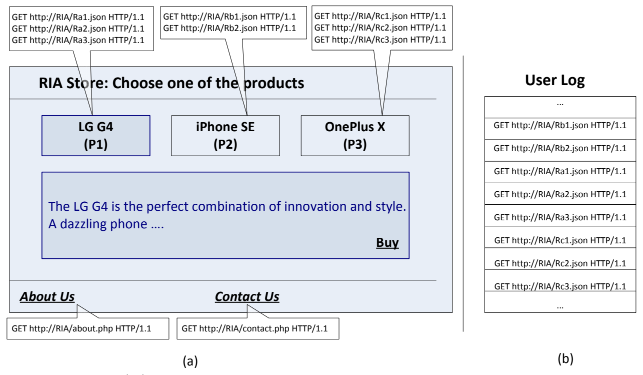
```
GET http://RIA/Ra1.json HTTP/1.1
GET http://RIA/Ra2.json HTTP/1.1
GET http://RIA/Ra3.json HTTP/1.1
```
```
GET http://RIA/Rb1.json HTTP/1.1
GET http://RIA/Rb2.json HTTP/1.1
```
```
GET http://RIA/Rc1.json HTTP/1.1
GET http://RIA/Rc2.json HTTP/1.1
GET http://RIA/Rc3.json HTTP/1.1
```
```
GET http://RIA/about.php HTTP/1.1
```
```
GET http://RIA/contact.php HTTP/1.1
```
User Log
```
...
GET http://RIA/Rb1.json HTTP/1.1
GET http://RIA/Rb2.json HTTP/1.1
GET http://RIA/Ra1.json HTTP/1.1
GET http://RIA/Ra2.json HTTP/1.1
GET http://RIA/Ra3.json HTTP/1.1
GET http://RIA/Rc1.json HTTP/1.1
GET http://RIA/Rc2.json HTTP/1.1
GET http://RIA/Rc3.json HTTP/1.1
...
```
RIA Store: Choose one of the products
LG G4 (P1)
iPhone SE (P2)
OnePlus X (P3)
The LG G4 is the perfect combination of innovation and style. A dazzling phone ....
Buy
About Us
Contact Us
(a)
(b)

Figure 1: (a) A simple RIA and generated requests after clicking on DOM elements. (b) Portion of the user's log

Having defined the RIA as an FSM we can present the problem using the following inputs and outputs: here as the input we have the initial state $s_0$ of the RIA and the set of previously generated requests and responses $T$, and our goal is to find the sequence of actions $A$ where:

$$\lambda(s_0, A) = T \tag{1}$$

Following a series of action in $A$, we can infer all states of the RIA during the given user-session.

*Example*: Figure 1 shows the initial state of a simple RIA application. This is a portion of an E-Commerce Website where the user can click on each product and see its description. Figure 2 presents a portion of this RIA's FSM. This FSM shows four states S = $\{s_0, ..., s_3\}$ which refer to states after clicking on the first, second, third products, and "about-us" respectively[2]. For example, the initial state of the application is $s_0$ and in this state by clicking on P2, two request/response pairs $\{Rb_1, Rb_2\}$ are generated and the RIA reaches state $s_1$.

Suppose that we have the user-log of Figure 1 (b), $T = \{Rb_1, Rb_2, Ra_1, Ra_2, Ra_3, Rc_1, Rc_2, Rc_3\}$. Given this input the desired sequence of actions is $A = \{P_2, P_1, P_3\}$ which satisfies equation 1.

[2]Other states are not shown to keep the diagram simple

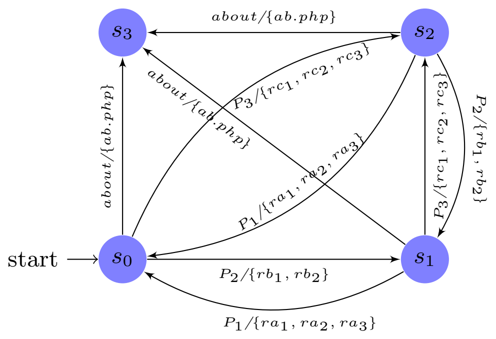

Figure 2: Portion of the FSM of the RIA in Figure 1. Inputs represent clicking on an element, output represents generated HTTP requests/responses

**Working Conditions** It is assumed that *D-ForenRIA* has the log for a *single* user. Extracting such a single-user log from server logs is a well-studied problem (see e.g., [22]) which is out of the scope of this paper. In our context, the traffic has already been recorded and no additional instrumentation of the user's browser is possible. In addition, our reconstruction is done off-line with no access to the original RIA. Furthermore, *D-ForenRIA*, currently can not support situations where an action does not generate any HTTP traffic.

## 3. PROPOSED SOLUTION

**Architecture of the System:** Figure 3 presents the general architecture of *D-ForenRIA*, with the four main components: SR-Proxy, SR-Browsers, Trace, and Output. *D-ForenRIA* has been implemented as a distributed system, where a number of SR-Browsers (Session Reconstruction Browsers) interact with the *SR-Proxy* (Session Reconstruction Proxy) to concurrently reconstruct the user session.

The *SR-Proxy* performs the role of the original Web server, and responds to the stream of requests sent by each *SR-Browser*. *D-ForenRIA* is based on a distributed set of *SR-Browsers* that can dynamically be added or removed during the reconstruction process. Each SR-Browser is composed of a real Web browser (e.g., Firefox) and a controller. The controller is responsible for executing the message sent by the *SR-Proxy* on the Web browser (e.g., which action should be executed on the current DOM). The previously recorded *HTTP trace* is given as an input to the *SR-Proxy*. SR-Browsers are responsible for doing what SR-Proxy asks them to do (e.g. executing actions using its browser) whereas SR-Proxy responds to the generated requests and verifies the correctness of the actions. A distributed implementation ensures concurrent execution of several actions at each RIA's state. The *SR-Proxy* keeps track of previously tried actions and uses this knowledge to choose the next candidate actions. *D-ForenRIA* recovers automatically and efficiently all the required output of the session reconstruction process. The *Output* includes the precise sequence of *User actions* (e.g., clicks, selections), the *User inputs* provided during the session, *DOMs* of each visited page, and screenshots of the pages seen by the user.

**Interactions between SR-Browser and SR-Proxy**: We now present the communication chain between a *SR-Browser* and the *SR-Proxy*. Session reconstruction can be seen as a loop of interactions, where the *SR-Proxy* repeatedly assigns the next candidate action to the *SR-Browser* (see Figure 4). We call this repetitive process *iteration*. We refer to Figure 4 as an illustration of the sequence of messages exchanged between the main components with items of the following numbers refer to the numbers in the figure.

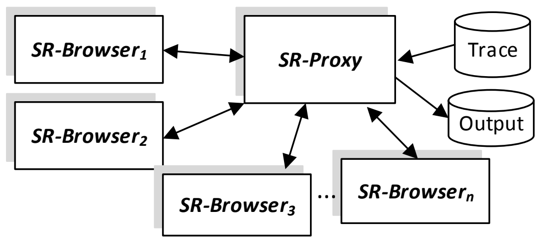

Figure 3: Architecture of *D-ForenRIA*

1. At each iteration, the *SR-Browser* sends a *"Next"* message asking *SR-Proxy* the action to do next.

2. The *SR-Proxy* asks the first *SR-Browser* reaching the current state to send the list of all possible actions on the current DOM (using the *"Extract (clue)"* message). At this step, the *SR-Proxy* also sends the clue (as explained in Section 3.2) regarding the next expected HTTP requests.

3. The *SR-Browser* extracts a set of candidate actions and tags them using DOM based features (e.g., tags the elements that are hidden or have no event handler attached). The *SR-Browser* sends these *annotated actions* to *SR-Proxy.*

4. The *SR-Proxy* computes the score of annotated actions using a scoring function (see Equation 2). These scores are used to sort the list of annotated actions from the most promising to the least promising action as explained in Section 3.2.

5. After this, and while working on that same state, the *SR-Proxy* assigns a new candidate action to each *SR-Browser* that sends a *"Next"* message, along with all the required instructions to reach that state (using an *"Execute (actionlist)"* message).

6. As each *SR-Browser* executes known or new actions, they generate a stream of HTTP requests. The proxy responds to the generated requests using the recorded log ("HTTP Request" / "HTTP Response" loop).

This outer loop continues until all user actions are recovered.

### 3.1 Extraction of Candidate Actions

In *D-ForenRIA*, after a state is discovered, *SR-Proxy* assigns to one of the browsers the task of extracting the candidate user-browser actions on the DOM. These actions are then assigned one-by-one to SR-Browsers by *SR-Proxy* until the correct action is found. Examples of actions include clicking on an element, scrolling down a list, filling a field and submitting forms (which includes filling fields and clicking on a submit button).

**Event-handlers and Actions**: To find candidate actions, *D-ForenRIA* needs to find "event-handlers" of DOM elements. Event-handlers are functions which define what should be executed when an event is fired. For example, in Figure 6, *FetchData(2)* is the event-handler for *onclick* event of P2.

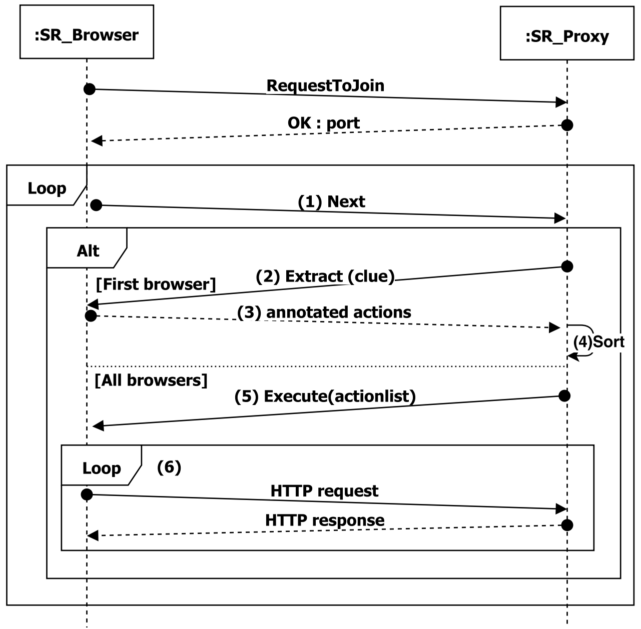

Figure 4: Sequence diagram of messages between a SR-Browser and the SR-Proxy

The existence of this event-handler means that there is a candidate action "*Click P2*" on the current DOM.

**Detection of Event-Handlers**: Event-handlers can be assigned statically to a DOM element, or dynamically during execution of a JavaScript code. To detect each type, we use the following techniques:

1. *Statically assigned event-handlers*: to find this type of handlers, it is enough to traverse the DOM and check the existence of attributes related to event-handlers (e.g. *onclick*, *onscroll*,...).

2. *Dynamically assigned handlers*: in JavaScript, dynamically assigned handlers are set using the *addEventListener* function[2]. So, to find this type of event-handlers we need a way to keep track of calls to this function. *D-ForenRIA*, overrides the built-in *addEventListener* function (Figure 5) such that each call of this function notifies *D-ForenRIA* about the call (Line 3) and then calls the original *addEventListener* function (Line 4). This technique is called hijacking [10] and can be realized by injecting the code in Figure 5 in the current DOM.

   *The Importance of Bubbling*: DOM elements can also be nested inside each other and the parent node can be responsible for events triggered on child nodes via a mechanism called "Bubbling"[4]. In this case, there is a one-to-many relationship between a detected handler and possible actions and by finding an event-handler we do not always know the actual action which triggers that event. In some RIAs, for example, the *Body* element is responsible for all click events on the page. However, in practice this event-handler is only responsible for a subset of the elements inside the body. In this case, it is difficult to find the elements which trigger the event and are handled by the parent's event handler. To alleviate this issue, *D-ForenRIA* tries elements starting from the bottom of the DOM tree assuming that leaf elements are more likely to be elements triggering the event.

```
1   var addEventListenerOrig = Element.prototype.
        addEventListener;
2   var EventListener=function(type,listener) {
3      notifyDynamicHandler(this, type, listener);
4      addEventListenerOrig.call(this, type,listener);

5   };
6   Element.prototype.addEventListener= EventListener;
```

Figure 5: Hijacking the built-in JavaScript AddEventListener function to detect dynamically assigned handlers.

Finally, the SR-Browser which has been asked to generate the pool of actions extracts all possible actions based on extracted event-handlers. In addition, for each action it also calculates some *meta-data* describing features of elements involved in that action. This *meta-data* is used to order actions by the SR-Proxy side (Section 3.2.1).

## 3.2 Efficient Ordering of Candidate Actions

Web pages usually have hundreds of elements. So blindly trying every action on these elements to find the right one is impractical (see Section 4). In *D-ForenRIA*, we have several techniques to order candidate actions. Our methods are categorized into two sets.

### *3.2.1 Element-Based Ordering*

In this technique we evaluate HTML elements based on their properties. Our goal is to decrease the priority of actions with which users cannot interact. For example, elements which are invisible or elements without event handler (Section 3.1) and elements with tags with which users usually do not interact (e.g. *script*, *link*) have the lowest priority.

This technique calculates the "*meta-data*" of each action. This meta-data is the summary of the important features of the elements involved in the action (e.g. whether elements are visible or not, whether any handler is attached to them). When SR-Browser sends the actions to the *SR-Proxy* (message (3) of Figure 4), each action is tagged by this meta-data.

### *3.2.2 Signature-Based Ordering*

In this category, a *SR-Browser* and the *SR-Proxy* collaborate to find the next most promising action. For instance, when a *SR-Browser* reaches the correct state by executing the correct action, it sends the "*Next*" message to the *SR-Proxy*, which passes the information regarding the next expected HTTP request as a response along with the "*Extract*" message (message 2 in Figure 4). We call this information the *clue*.

The *SR-Browser* exploits the received *clue* to possibly find a right action on the current DOM. For example, if the *clue* suggests that the next possible action should be the navigation to page $p$, then all the *href* tags on DOM that point to page $p$ are assigned higher score and clicking on these actions should have the maximum priority. In addition, if there are other *hrefs* which request page $q$ ($q \neq p$), then clicking on these elements should have the minimum priority.

The *SR-Proxy* also uses the knowledge acquired from the traffic generated by previously tested actions (which we call the "signature" of the actions) to prioritize the pool of actions.

The signature-based scoring function (see Equation 2) calculates *score(s,a,r)* (such that $score(s, a, r) \in \{0, 0.5, 1\}$) us-

```
<meta name="SSRG" content="Sample RIA">
<body onload = "attachHandler()">
  <div style="visibility:hidden">SSRG 2016 </div>
  <span>RIA Store:Choose one of the products</span>
  <hr>
  <span id='p1' >LG G4</span>
  <span id='p2' >iPhone SE</span>
  <span id='p3' onclick="FetchData(2)">OnePlus X</
     span>
  <div id="container">--</div>
  <hr>
  <a href="about.php">About Us</a>
  <a href="contactus.php">Contact Us</a>
</body>
```

Figure 6: A simple DOM instance

ing clue $r$ for all actions on the current state $s$. The most promising actions will be assigned higher scores.

If we define $PA$ as the set of previously executed actions, we can calculate the score for candidate actions of a new state using the following formula:

$$score(s_i, a, r) = \begin{cases} 0 & (a \in PA) \wedge \forall j \ \neg prefix(\lambda(s_j, a), r) \\ 0.5 & (a \notin PA) \\ 1 & otherwise \end{cases} \tag{2}$$

where $s_i$ is the current state and $s_j$ is one of the previous states and $prefix(A, B)$ determines whether the sequence of requests in $A$ are at the beginning of the sequence of requests in $B$.

If the signature of action $a$ is not a prefix of the remaining trace $r$, we should decrease the priority of $a$. On the other hand, if a signature of $a$ matches the next unconsumed trace, $a$ should be considered a promising action.

*D-ForenRIA* combines both element-based and signature-based scoring to rank the promising actions (a.k.a., pool of actions) at each state.

### 3.2.3 Example

To illustrate how *D-ForenRIA* orders actions on a page, we use the simple RIA given in Figure 1 (a). The DOM of this RIA and corresponding JavaScript code are shown in Figures 6 and 7 respectively.

```
      var reqs =
      [ ["ra1.json", "ra2.json", "ra3.json"],
      ["rb1.json","rb2.json"],
      ["rc1.json","rc2.json","rc3.json"]];
      //Attch handlers
      function attachHandler(){
        $("#p1").on("click",
        function(){ FetchData(0); });
        document.getElementById("p2").onclick =
        function(){ FetchData(1); }
      }
      //Fetching data
      function FetchData(id){
        $('#container').empty();
        for (res of reqs[id]) {
          $.get( res, function( data ,status ) {
            $('#container').append(data);
          }, 'text');
        } }
```

Figure 7: A simple JavaScript code snippet

Considering the "Element-Based" ordering, *D-ForenRIA* minimizes the priority of non promising elements. In the DOM of Figure 6, the *meta, hr* and *body* elements have non-promising tags and therefore are given the lowest priority. Two *div* tags are also not promising because of being hidden and having no handler attached, respectively.

To apply the "Signature-Based" ordering, the system should calculate the score of any action based on its $\lambda$ function. At the initial state, the priority for two *hrefs* is minimum since their initiating requests (*about.php* and *contactus.php*) do not match the next expected requests. The score for the remaining actions are *0.5* since the system has not tried any action yet. Assume that actions are tried in the order $p_1$, $p_2$, $p_3$. *D-ForenRIA* will try *p1* and *p2* to discover the correct list of actions. In addition it learns $\lambda(s_0, p_1)$ and $\lambda(s_0, p_2)$. At the next state, *p1* gets the score of *1* since its signature $Ra_1, Ra_2, Ra_3$ matches the remaining of traces ($< Ra_1, Ra_2, Ra_3, Rc_1, ... >$), $p_2$ gets score of *0* since its $\lambda$ does not match and *p3* gets *0.5* since we have not tried this action yet. So, the correct action $p_1$ is detected immediately since it has the maximum score of *1*. At the third state, $p_3$ gets a score of *0.5* while $p_1$ and $p_2$ are assigned score *0*. At this state also the correct action is detected immediately. To sum up, the 3 actions are found after trying 4 actions on the DOM. Blindly trying all possible actions could require executing $8 + 7 + 9 = 24$ actions[3] on the page. Since there are 14 actions in the current DOM on average $14 \div 2 = 7$ actions should be tried before finding the correct action by trial and error.

## 3.3 Checking the Stable Condition (SR-Browser)

The *SR-Browser* usually needs to execute a set of actions as decided by the *SR-Proxy* in response to a "*Next*" message. After triggering each action, a SR-Browser should wait until that action is completed and the application reaches what we call a "stable condition".

To check the stable condition, a SR-Browser checks two things:

- *Receiving All Responses*: *D-ForenRIA* uses two techniques to be sure that the response for all generated requests have been received. First, SR-Browser waits for the *window.onload* event to be triggered. This event is being triggered when all resources have been received by the browser. However, this event is not triggered when a function requests a resource using Ajax.

  To keep track of Ajax requests, *D-ForenRIA* overrides *XMLHttpRequest*'s *send* and *onreadystatechange* functions. The first function is being called automatically when a request is being made and the second function can be used to detect when a response is fully received by the browser.

- *Applicability of the Action on DOM*: When there is no more pending requests, the system waits for the elements involving in the action to appear on the page. This check is required to let the browser consume all previously received resources and render the new DOM.

## 3.4 Timeout-based Ajax calls

RIAs sometimes fetch data from the servers periodically (e.g., current exchange rate or live sports scores).

There are different methods to fetch data from the server. One approach, which is called *polling*, periodically sends HTTP requests to the server using Ajax calls. There is

[3]There are 14 actions in the DOM of Figure 6, and if *D-ForenRIA* traverses the DOM tree in postorder way, clicking on products $p_1$, $p_2$ and $p_3$ are 8th, 7th and 9th actions in the pool of candidate actions.

usually a timer set with *setTimeout/setInterval* functions to make some Ajax calls when the timer fires. To keep track of such calls, *D-ForenRIA* takes a two-step approach:

1. *Timer Detection*: It detects all registered timers by overwriting the *setTimeout/setInterval* functions. The SR-Browser then executes these functions to let the SR-Proxy know about the signature of the timer.
2. *Timer Triggering*: Since *D-ForenRIA* knows the signature of timers, when it detects that the next expected HTTP request matches the signature of any timeout based function, it asks a SR-Browser to trigger that function (According to formula 2).

However there are two other approaches to implement periodic updates: *Long-Polling* which is based on keeping a connection between client and server open, and *WebSockets* which creates a bidirectional non-HTTP channel between the client and server. Currently, *D-ForenRIA* supports polling but not Web-Sockets and Long-Polling.

### 3.5 Handling Randomness (SR-Proxy)

Randomness occurs when the execution of the same action generates a different set of requests and responses. This can happen at the client-side as well as at the server-side. However, *D-ForenRIA* is based on a proxy (that act as a server) which replays the same traffic, therefore our system would not face the server-side randomness.

If the client includes randomly generated values in the requests, then these requests will differ from the recorded traffic. For example, random parameters are one of the methods to disable proxy caching in modern RIAs. Handling the client-side randomness is challenging. In *D-ForenRIA*, the *SR-Proxy* uses a deterministic method (i.e., the same action is selected each time) to detect a partial match of the generated requests from the recorded traffic. If the partial match satisfies what we call the similarity criteria ,that is, everything is similar except the values of parameters in the resources, *SR-Proxy* asks the browser to re-execute the same action. If the *SR-Proxy* observes a change in the same set of parameters, then it concludes that the action contains randomly generated values. The system does not consider these random parts of the requests for checking the correctness of the action.

### 3.6 Other features of the system

In addition to a distributed architecture, *D-ForenRIA* utilizes other techniques to make the session-reconstruction practical and efficient. The reader is referred to [7] for detailed explanation of the basic version of *D-ForenRIA*:

- *Finding User-Inputs*: *D-ForenRIA* can also detect the form submission actions to continue the session reconstruction. The *SR-Proxy* glance over the log for the occurrence of $< name, value >$ pairs and sends this information as a *clue* to *SR-Browser.* The *SRBrowser* tries to match $< name, value >$ pairs inside the *clue* with input elements on the current DOM and then tries different actions to submit the form. However, *D-ForenRIA* currently does not support other non-HTML-form user-inputs.
- *Loading the Last Known Good State (Reset)*: When a SR-Browser executes an incorrect action it should reload its previous state. In this case, *D-ForenRIA* asks the SR-Browser to "*reset*" to the initial state and execute all previously detected actions.
- *Verifying the Correctness of an Action*: *SR-Proxy* considers an action correct if the entire set of generated requests/responses eventually form a gap-less block in the recorded traces and this gap-less block comes right after the previously matched block.

## 4. EXPERIMENTS

To assess the effectiveness of the proposed session reconstruction system we have conducted several experiments. Our research questions can be presented as follows.

- *RQ1.* Is *D-ForenRIA* able to reconstruct user-session efficiently?
- *RQ2*: Does distributed reconstruction have a positive influence on the performance and is there any limit on the number of browsers that can be added to reduce the execution time?
- *RQ3*: How effective are different techniques of ordering candidate actions on a given state?
- *RQ4*: What are the User-Log storage requirements in *D-ForenRIA*?

Our experimental data along with the videos are available for download[4].

### 4.1 Subject Applications

In this paper, we limit our test-cases to RIAs. We used sites with different technologies and from different domains. The reason to focus on RIAs is that other tools can already perform user-interaction reconstruction on non-Ajax Web applications (e.g. [19]). Table 1 presents characteristics of our test-cases.

The first site, C1, in our case study is a web-based open-source file manager, written in JavaScript using jQuery and jQuery UI. Our second case, C2, is an Ajaxified version of IBM's Altoro-mutual website. This is a demo banking website used by IBM for demonstration purposes. Our team has made this website fully Ajax-based where all user actions trigger Ajax requests to dynamically fetch pages. C3 is a fully Ajax-based periodic table and C5 is a Website developed by our team which represents a typical personal homepage. The more advanced website, C4, is a Web-based goal setting and performance management application built using Google Web toolkit) which has numerous clickables at each state.

### 4.2 Experimental Setup

Experiments are performed on Linux-based computers with an Intel® Core™2 CPU at 3GHz and 3GB of RAM on a 100Mbps LAN. To implement the *D-ForenRIA* SR-Browsers, we used Selenium over actual browsers. *D-ForenRIA*'s SR-Proxy is implemented as a Java application. For each test application, we recorded the full HTTP traffic of user interaction with the application using *Fiddler*[5].

To address RQ1, we captured a user-session for each of the subject applications and ran *D-ForenRIA* to reconstruct the session using the given traffic. We report "cost" and "time" of the reconstruction as measures for efficiency.

[4]http://ssrg.site.uottawa.ca/sr/demo.html
[5]http://www.telerik.com/fiddler

Table 1: Subject applications and characteristics of the recorded user-sessions

| ID | Name | #Requests | #Actions | URL |
|---|---|---|---|---|
| C1 | Elfinder | 175 | 150 | https://github.com/Studio-42/elFinder |
| C2 | AltoroMutual | 204 | 50 | http://www.altoromutual.com/ |
| C3 | PeriodicTable | 94 | 45 | http://ssrg.site.uottawa.ca/apr5/success1/ |
| C4 | Engage | 164 | 25 | http://engage.calibreapps.com/ |
| C5 | TestRIA | 74 | 31 | http://ssrg.eecs.uottawa.ca/testbeds.html |

The "cost" counts how many events SR-Browsers have to execute before successfully reconstructing all interactions. The following formula calculates the cost of session reconstruction:

$$n_e + \sum_{n=1}^{n_r} c(r_i) \qquad (3)$$

where $n_e$ is the number of actions on the user's session and there are $n_r$ resets (see Section 3.6) during reconstruction and the $i^{th}$ reset, $r_i$, has cost of $c(r_i)$. If we define $l(r_i)$ as the number of actions discovered before the $i^{th}$ reset, we have $c(r_i) = l(r_i)$ in our implementation of loading last know good state (see Section 3.6)

As a point of comparison, the results are also provided for the "basic solution" defined as follows:

***The basic solution***: Any system aiming at reconstructing user-interactions for RIAs needs to at least be able to handle user inputs recovery, client-side randomness, sequence checks and be able to restore a previous state, otherwise reconstruction may not be possible. In our experiments, we call such a system the "basic solution". It performs an exhaustive search for the elements of the DOM to find the next action and it does not use the proposed techniques in Section 3.2. To the best of our knowledge at the time of writing, no other published solution has the characteristics of such a basic solution, and thus no other solution can help reconstructing RIA sessions, even inefficiently.

***The no-reset time***: If our session reconstruction algorithm can find all user-browser interactions without trying incorrect actions it does not need any reset. We also report the inferred time for this "no-reset" algorithm by measuring the total time required by *D-ForenRIA* to reconstruct the session minus the time spent during reloading the last known good state. This provides an "optimal" time for our tool.

To address RQ2, for each given user-session log, we ran *D-ForenRIA* with 1,2,4 and 8 browsers and report cost/time of the reconstruction to measure scalability of the system. To address RQ3, we ran *D-ForenRIA* using a single browser and measure how effective is applying each of the element/signature ordering at each DOM. Finally, to answer RQ4 we report storage requirements for each action in the compressed format and the effect of pruning multimedia resources from traces.

## 4.3 Experimental Results

**Efficiency of D-ForenRIA (RQ1):** Table 2 presents the time and cost of reconstruction of full sessions using *D-ForenRIA*, and the basic solution. In this experiment we use just a single SR-Browser. We report time measurement for several browsers in the next section.

Table 2: Time and Cost of reconstruction using *D-ForenRIA*, and the basic solution.

| ID | D-ForenRIA | | Basic Solution | |
|---|---|---|---|---|
| | #Events | Time (H:m:s) | #Events | Time (H:m:s) |
| C1 | 2,882 | 0:11:01 | 102,933 | 09:51:26 |
| C2 | 52 | 0:02:25 | 34,505 | 04:31:57 |
| C3 | 1,325 | 0:04:22 | 308,548 | 19:28:48 |
| C4 | 3,506 | 0:19:47 | 21,518 | 02:12:01 |
| C5 | 394 | 0:02:29 | 14,847 | 00:48:29 |

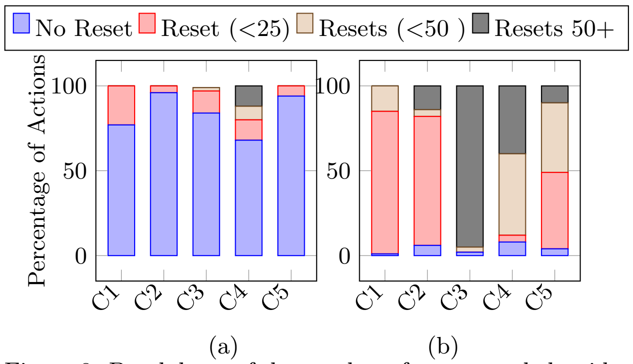

Figure 8: Breakdown of the number of resets needed to identify a user-browser interaction in *D-ForenRIA* (a) and in the basic solution (b).

*D-ForenRIA* outperforms the basic solution in all cases. On average it takes 44 events to be executed to find a user-browser interaction while the basic solution needs 5964 events. Regarding the execution time, *D-ForenRIA* (even using a single browser) is an order of magnitudes faster than the basic solution. On average *D-ForenRIA* needs 13 seconds to detect an action while the basic solution needs around 8 minutes to detect an action.

*Number of Resets per Action*: Figure 8 present a breakdown of the number of resets needed to detect a user browser action in the test cases in *D-ForenRIA* and the basic solution. For *D-ForenRIA*, in all cases the majority of actions are identified without any reset (The worst case happens in $C_4$ where 32% of actions need at least one reset to be found and 12% of these actions need more than 50 resets). On average in our test-cases 84% of actions are found immediately at the current state based on the ordering done by the SR-Browser and SR-Proxy (Section 3.2). On the other hand, for the basic methods (Figure 8 (b)), 96% of actions need at least 25 resets. This figure also shows that the ba-

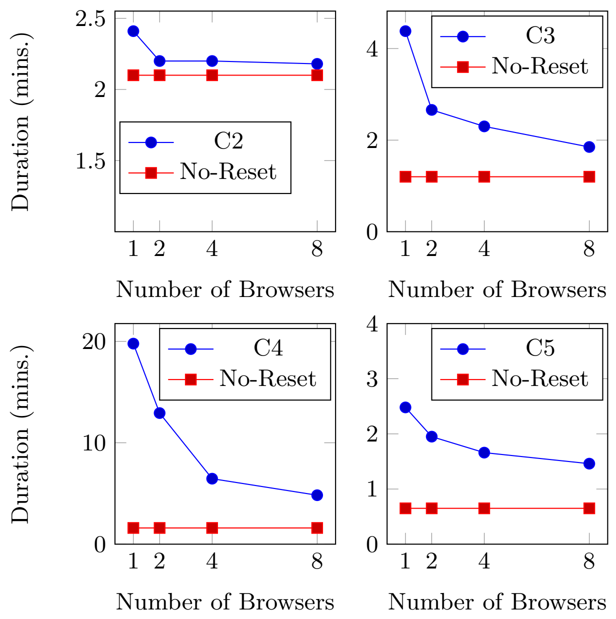

Figure 9: Scalability of *D-ForenRIA* in different RIAs compared to the no-reset time.

sic solution tries more than 50 actions to find 32% of actions.

**Performance of the Distributed Architecture (RQ2)**: Figure 9 presents the execution time of the system when we add more browsers to reconstruct the sessions. The results are reported for 1,2,4 and 8 browsers. Since *D-ForenRIA* is concurrently trying different actions on each DOM we expected that adding more browser would speedup the process. In fact, if the algorithm needs $nr_i$ resets to find the $i^{th}$ correct action, using $nr_i$ browsers should decrease the execution time. The results we obtained verified this argument. The best speedup happens in C3 and C4 where we have the largest number of resets (See Figure 8). For C5 adding more browsers is not as effective as C4 and C3 since many actions are found correctly without the need to try different actions (Ordering of actions detect the correct action as the most promising one (Section 3.2)). However, adding more browsers is not always beneficial; For example in C2, we observed no improvement of the execution time after adding more browsers (from 2 to 4). This is because in this application many actions are found immediately by *D-ForenRIA*. In this case, adding more browsers to try different candidate actions at each DOM is not beneficial.

**Efficiency of Candidate Actions Ordering Techniques (RQ3)**: As we discussed in Section 3.2, SR-Proxy and SR-Browsers in *D-ForenRIA* collaborate to find the most promising candidate actions on the current DOM. *D-ForenRIA* uses several techniques which we categorized as “Element-based” and ”Signature-based”. To understand the effectiveness of theses techniques we measured the characteristic of each DOM during reconstruction. For each DOM, we were interested to know the number of elements, number of visible elements, elements with handler, leaf elements, the size of the DOM and also number of signatures that can be applied on the DOM. Table 3 presents the average of these measurements for all DOMS of the test cases. On average, over all applications, there were 267 elements on the DOM, 86% of them are visible, 17% have handler, 54% are leaves

Table 3: Characteristics of DOM elements and ratio of elements with signatures at each DOM

| ID | #Elements | #Visible | #Handlers | #DOM Leaves | DOM Size(KB) | Signs. applied(%) |
|---|---|---|---|---|---|---|
| $C_1$ | 79 | 64 | 14 | 43 | 6.2 | 4 |
| $C_2$ | 108 | 106 | 31 | 51 | 8.1 | 12 |
| $C_3$ | 648 | 417 | 126 | 500 | 23.4 | 8 |
| $C_4$ | 268 | 228 | 1 | 117 | 22.7 | 20 |
| $C_5$ | 44 | 43 | 10 | 22 | 9 | 23 |

Table 4: Log size features for test cases

| ID | Reqs./ Action | Log Size / Action (KB) | Pruned Log Size/ Action (KB) |
|---|---|---|---|
| $C_1$ | 1.16 | 1.58 | 1.2 |
| $C_2$ | 1.36 | 1.41 | 0.41 |
| $C_3$ | 1.05 | 1.12 | 1.12 |
| $C_4$ | 6.56 | 11.47 | 9.96 |
| $C_5$ | 2.38 | 3.38 | 3.38 |

and *D-ForenRIA* has the signature of 16% elements on the current DOM.

Filtering based on visibility is effective in all cases and reduces candidate actions by 15%. Ordering based on event handlers is quite effective in all cases except in C4. If we exclude C4, 77% of elements don’t have any handler. In RIAs like C4 where there is a single handler to handle all events on the DOM, it is very challenging to find elements with actual event handlers. As we suggested in Section 3.2, *D-ForenRIA* gives high priority to leaf elements of the DOM. However, there is still a considerable ratio of leaf nodes, 54% on the DOMs. To sum up, in websites similar to C4, it was insufficient to just apply “Element-based” ordering, however “Signature-based” was effective in all cases and we could apply it on 16% of elements on each DOM.

**User-Log Storage Requirements (RQ4)**: One of the features of the input user-log for *D-ForenRIA* is that it should contain both HTTP request and response. It includes the body of requests and responses. One may be concerned about the size of the user-log. To investigate the storage requirements of “Full” HTTP traffic in RIAs we measured some features of HTTP logs in our test cases (Table 4). e

As expected the number of requests for each action is quite low. In our experiment the actions with the most number of requests are usually the first page of the application and the average number of requests per action is less than 3 requests. This low number of requests are expected because of Ajax calls for partial updates of the DOM which are common in RIAs. To measure the storage requirements, we calculated the compressed required space[6] to store the “full” HTTP request-responses of each action. The required size for each action varies from as low as 1.12 KBs to the high of 11.74 KBs for $C_4$ and the average is 3.79 KBs. We also considered pruning the log from multimedia resources (i.e. Images and videos). With pruning, the average required space dropped

[6]The compression was done using 7z algorithm

by 15% and reached about 3.2 KBs. These numbers seem promising because of the trend in decreasing costs of storage devices. In addition, the Web-master can purge the recorded traffic after session reconstruction or if it does not seem interesting for further analysis.

## 4.4 Discussion:

***Recording HTTP traffic***: The HTTP requests exchanged between a browser and the server can be logged in different places in the network; They can be logged on the server, in the proxy-server or even on the client-side. However, recording using proxy or on the client-side requires additional configuration/installation of recording software which is not desired. We believe that the best scenario to use *D-ForenRIA* is recording the traffic on the server side[7]. HTTP servers (Like Apache[3] or IIS[1]) can be configured to record the full traffic which is the input of session reconstruction. It is notable that to be able to use *D-ForenRIA*, there is no need to change the Web application or to instrument any code on the client side. In addition, no extra information is collected on the server side which minimizes privacy concerns.

*SSL and recording HTTP traffic*: In *D-ForenRIA* it is assumed that the traffic is in plain format and no decryption is needed on the input. Although SSL[14] provides a secure connection "between" a client and the server it does not encrypt logs on the server. Therefore SSL is not a barrier for recording the traffic on the server.
However, a real issue with SSL-enabled sites exists during the session reconstruction process. SR-Browsers still want to communicate with a real server, however, *D-ForenRIA* does not have access to the actual server during reconstruction. To solve this problem, *D-ForenRIA*'s SR-Proxy acts as a man-in-the-middle[8].

***Importance of using Selenium***: *D-ForenRIA* uses Selenium to implement our SR-Browsers. Selenium enables us to use different actual browsers (for example Chrome or Firefox) during reconstruction. It is important for *D-ForenRIA* to use the browser that the user has used while visiting the website. For example, in our applications, C4 could only be reconstructed using FireFox since the website generates slightly different resources based on the current user's browser. However, *D-ForenRIA* can detect the user's browser in the headers of HTTP requests and select the right type of browser automatically.

***Threats to validity***. A threat to the validity of our experiments is the generalization of the results to other test cases. To mitigate this issue, we used test cases from different domains and test cases built using different technologies; although we do need more test cases to make our results more generalizable.

The other threat to the validity of our experiment is related to the cost/time function (Equation 3) that we used to measure efficiency of *D-ForenRIA*. However, we believe that these metrics capture the effectiveness of the system properly.

## 5. RELATED WORKS

Formerly, *user-session reconstruction* meant being able to find which pages a user had visited during a session, and differentiating users in server logs. This task is often considered as a preprocessing task for Web Usage Mining [23, 12]. In this paper, we assume that individual user-session have already been identified using one of these techniques. To the best of our knowledge at the time of writing, few works [9, 16, 19, 20] have been done in this area, notably there is only one effort [7] to report for RIAs.

WebPatrol [9] introduced the concept of automated collection and replay of Web-based malware scenarios (WMS). The malware infection trails are collected by signature-based, low-interaction "honey clients" and its scenario replay component reconstructs the original infection trail at any time from WMS depository. However, WebPatrol is not meant to reconstruct generic RIA sessions and therefore it cannot be adopted for our purpose. Resurf [16] proposed a graph-based method, which creates *click-through-stream* of user's session based on a referral relationship between user-requests. Recently, a session reconstruction [20] approach has been introduced to reconstruct *multi-tabbed* user session, which is considered non-trivial since HTTP logs do not contain information regarding opened tabs. The method links browser logs, HTTP logs and traces in lower layers of the network to extract information regarding opened tabs. The working condition of this approach requires traffic recording on the user's machine, which is a limitation in our context as we aim to reconstruct user session from given HTTP logs. Such assumption is also not convenient because of dependency on end-users and privacy concerns.

The tools most related to our proposed approach are presented in ClickMiner [19] and ForenRIA [7]. ClickMiner [19] reconstruct user session from HTTP traces recorded by a passive proxy. Their proposed approach focuses on actions that change the URL of the application. However, in RIAs, many actions do not alter the URL but rather update the DOM of the page [18]. In addition, although there is some level of support for JavaScript, it is lacking the specific capabilities (e.g., handling user-inputs, client-side randomness, restoring the previous state, sequence check) that are required to handle RIAs. Our previous work ForenRIA [7] proposes a forensics tool to perform automated and complete reconstruction of user session in RIAs. However, ForenRIA is implemented as a single-thread system where a single client (i.e., browser) is responsible to execute all the possible actions on a given page, which is not scalable. In addition, in ForenRIA, the client-side randomness was handled by running the two concurrent instances of the *SR-Browsers*. However, such implementation occupies more resources (i.e., it runs 2 times the number of required browsers), which is not ideal. In this paper, we have replaced the single-thread system with a distributed architecture for scalability and better performance. Several improvements are also suggested to handle client-side randomness, time-based Ajax calls, and detection of stable condition.

## 6. CONCLUSIONS

In this paper, we proposed *D-ForenRIA*, a distributed tool to recover user-browser interactions from a given HTTP trace in RIAs. Session-reconstruction in RIAs is a challenging task since simply looking at data flowing between the browser and the server does not provide the necessary information to reconstruct user-interactions. In contrast to previous session-reconstruction methods, *D-ForenRIA* only needs previously captured network traffic and there is no need for manipulation of the web application or installation

[7]It is notable that the SR-Proxy in *D-ForenRIA* is just used during the reconstruction and not for recording the traffic.

of any softwares on the client-side.

Experiments on different websites show promising improvement of performance and scalability, however there are different directions for improvements: first, the system must be tested on larger sets of RIAs. In addition, we need better algorithms to detect the most promising candidate actions where signature-based ordering is inapplicable. Another research area is to extend the ability of *D-ForenRIA* to detect user input-data which are submitted without standard HTTP forms.

There are other issues that remain open. For example, it is still unclear how to effectively handle actions which do not generate HTTP traffic or all of the requests that are cached by the browser. How to handle this problem in a non-naive way is also an open question.

**Acknowledgements** This work is supported by Center for Advanced Studies, IBM® Canada Lab and the Natural Sciences and Engineering Research Council of Canada (NSERC).

## 7. REFERENCES

[1] Advanced Logging for IIS - Custom Logging. Available at: https://msdn.microsoft.com/en-us/library/ms227673.aspx. Accessed July 5, 2016, 2009.

[2] addEventListener API, Mozilla Developers Network. Available at: https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener. Accessed June 28, 2016, 2016.

[3] Apache HTTP Server Version 2.2: Apache Module moddumpio. Available at: http://httpd.apache.org/docs/2.2/mod/moddumpio.html. Accessed May 28, 2016, 2016.

[4] Bubbling and capturing in JavaScript. Available at: http://javascript.info/tutorial/bubbling-and-capturing. Accessed June 28, 2016, 2016.

[5] S. Andrica and G. Candea. Warr: A tool for high-fidelity web application record and replay. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 403–410. IEEE, 2011.

[6] R. Atterer and A. Schmidt. Tracking the interaction of users with ajax applications for usability testing. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1347–1350. ACM, 2007.

[7] S. Baghbanzadeh, S. Hooshmand, G. Bochmann, G.-V. Jourdan, S. M. Mirtaheri, M. Faheem, and I. V. Onut. ForenRIA: The reconstruction of user-interactions from http traces for rich internet applications. In *Proceedings of the Twelfth Annual IFIP WG 11.9 International Conference on Digital Forensics*, 2016.

[8] F. Callegati, W. Cerroni, and M. Ramilli. Man-in-the-middle attack to the https protocol. *IEEE Security and Privacy*, 7(1):78–81, 2009.

[9] K. Z. Chen, G. Gu, J. Zhuge, J. Nazario, and X. Han. WebPatrol: Automated collection and replay of web-based malware scenarios. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011.

[10] B. Chess, Y. T. O'Neil, and J. West. Javascript hijacking, 2007.

[11] M. Cohen. Pyflag–an advanced network forensic framework. *Digital investigation*, 5:S112–S120, 2008.

[12] R. F. Dell, P. E. Román, and J. D. Velásquez. Web user session reconstruction with back button browsing. In *Knowledge-Based and Intelligent Information and Engineering Systems*, 2009.

[13] P. Fraternali, G. Rossi, and F. Sánchez-Figueroa. Rich internet applications. *Internet Computing, IEEE*, 14(3):9–12, 2010.

[14] A. Freier, P. Karlton, and P. Kocher. Secure socket layer. *IETF draft, November*, 1996.

[15] J. J. Garrett. Ajax: A New Approach to Web Applications. Available at: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/. Accessed May 28, 2015, 2005.

[16] T. K. M. F. Guowu Xie, Marios Iliofotou and Y. Jin. ReSurf: Reconstructing web-surfing activity from network traffic. 2013.

[17] S. Hooshmand, A. Mahmud, G. Bochmann, M. Faheem, G.-V. Jourdan, R. Couturier, and I. V. Onut. D-ForenRIA: Distributed reconstruction of user-interactions for rich internet applications. In *Proc. WWW*, Montreal, Canada, 2016.

[18] A. Mesbah. Software analysis for the web: Achievements and prospects. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER) – FoSE Track (invited)*, 2016.

[19] C. Neasbitt, R. Perdisci, K. Li, and T. Nelms. ClickMiner: Towards forensic reconstruction of user-browser interactions from network traces. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1244–1255. ACM, 2014.

[20] S. Raghavan and S. Raghavan. Reconstructing tabbed browser sessions using metadata associations for multi-threaded browser implementation. 2016.

[21] F. Schneider, A. Feldmann, B. Krishnamurthy, and W. Willinger. Understanding online social network usage from a network perspective. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 35–48. ACM, 2009.

[22] M. Spiliopoulou, B. Mobasher, B. Berendt, and M. Nakagawa. A framework for the evaluation of session reconstruction heuristics in web-usage analysis. *Informs journal on computing*, 15(2):171–190, 2003.

[23] J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan. Web usage mining: Discovery and applications of usage patterns from Web data. *SIGKDD Explorations Newsletter*, 1(2):12–23, 2000.