Journal of Internet Services and Applications

RESEARCH Open Access

Recovering user-interactions of Rich Internet Applications through replaying of HTTP traces

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

Abstract

In this paper, we study the "Session Reconstruction" problem which is the reconstruction of user interactions from recorded request/response logs of a session. The reconstruction is especially useful when the only available information about the session is its HTTP trace, as could be the case during a forensic analysis of an attack on a website. Solutions to the reconstruction problem do exist for "traditional" Web applications. However, these solutions cannot handle modern "Rich Internet Applications" (RIAS). Our solution is implemented in the context of RIAs in a tool called D-ForenRIA. Our tool is made of a proxy and a set of browsers. Browsers are responsible for trying candidate actions on each DOM, and the proxy, which contains the observed HTTP trace, is responsible for responding to browsers' requests and validating attempted actions on each DOM. D-ForenRIA has a distributed architecture, a learning mechanism to guide the session reconstruction process efficiently, and can handle complex user-inputs, client-side randomness, and to some extents actions that do not generate any HTTP traffic. In addition, concurrent reconstruction makes the system scalable for real-world use. The results of our evaluation on several RIAs show that D-ForenRIA can efficiently reconstruct user-sessions in practice.

Keywords: User-interactions reconstruction, Rich Internet Applications, Traffic replay, HTTP traces

1 Introduction

Over the last decade the increasing use of new Web technologies such as "Asynchronous JavaScript and XML" (AJAX) [1], and "Document Object Model" (DOM) manipulation [2] have provided more responsive and smoother Web applications, sometimes called "Rich Internet Applications"(RIAs) [3] or "Single-page Applications" [4]. RIAs have become the norm for modern Web applications [5]. For example, Google has adopted RIA technologies to develop most of its major products (Gmail, Google Groups, Google Maps, etc.) In fact, an evaluation of the top 100 Web sites from alexa.com shows that 87 of them use AJAX to communicate with the server-side scripts[1].

Despite the benefits of RIAs, this shift in Web development technologies has created a lot of challenges; many previous methods for analyzing traditional Web applications are not useful for RIAs anymore [6]. This is mainly due to the fact that AJAX fundamentally changes the concept of a web-page, which was the basis of a Web application. Among these challenges is the ability to analyze HTTP traffic of users' sessions.

When a user visits a website, user-browser interactions generate a set of HTTP requests and responses that are usually logged by the Web server. These logs can be used for example for "Session Reconstruction". We define the *Session Reconstruction Problem* as using *only* the logs to recover of an entire user-session, including the sequence of user-browser interactions during a user's visit, a reconstruction of the pages as they were presented to the user, the input values provided by the user and the elements of the page in which these inputs were provided. The reconstruction must not rely on prior instrumentation of the application, and must be performed completely offline (that is, without accessing the original application).

*Correspondence: gjourdan@uottawa.ca
[1] University of Ottawa, 800 King Edward Avenue, K1N 6N5 Ottawa, Canada
Full list of author information is available at the end of the article

Session reconstruction is specially important when the only information available from a previous session is the HTTP logs as could be the case during a forensic analysis of an attack on a website [7]. In addition, a session reconstruction tool may be used to understand users' navigation patterns in the website (in Web usage mining [8]) or to derive test-cases for the Web application [9].

The previous session reconstruction methods deal with traditional Web applications and cannot handle RIAs. Traditional web applications are composed of a set of web pages and users navigate the website by following hyperlinks. An assumption often made is that each user action navigates the application to a new page with a new URL. This URL is mentioned in the *href* property of the links (HTML anchors); therefore by simply considering links on each page and the next expected traffic, user actions of a traditional Web application can be extracted.

On the other hand, RIAs are event-based applications. Any HTML element, and not only links can potentially respond to actions of a user by an event-handler. These event-handlers can be assigned statically (using the attributes of an element), or dynamically by calling event-registration JavaScript functions. Detection of statically assigned handlers can be done by just scanning the DOM, however the algorithm needs a more advanced mechanism to keep track of dynamically assigned events. In addition, a session reconstruction tool for RIAs cannot tell which requests are going to be generated after triggering an action by simply looking at the target element of an action. The reason is that many of the requests are generated by script code running in the browser, and the response typically only contains a small amount of data which is used by the receiving script to partially update the current DOM (see Fig. 1 for an example). Therefore, extraction of candidate user actions at each state, and finding the source of a given HTTP request is challenging in RIAs. Consequently, when the Web application is a modern RIA, session reconstruction is not easy.

Although it is usually possible to recover user-client interactions by instrumenting the client or the application code, such instrumentation is not always desirable or feasible (e.g., in the analysis of a previously happened security incident).

In this paper, we present an approach to reconstruct sessions of RIAs using HTTP traces as only input. Our method uses a proxy, which contains the trace, and a set of browsers to reconstruct the session; The proxy plays the roles of the server and browsers are responsible for trying candidate actions that change the client-state of the RIA. The browsers also capture some information about the client-state of the application after performing an action, such as the screen-shot and the DOM of each state. The approach has been implemented in a tool called *D-ForenRIA*[2]. To the best of our knowledge, *D-ForenRIA* is the first tool that can efficiently reconstruct user-browser interactions of a RIA from a given HTTP log. A companion site has been set up at http://ssrg.site.uottawa.ca/sr/demo.html where videos and further information is being made available.

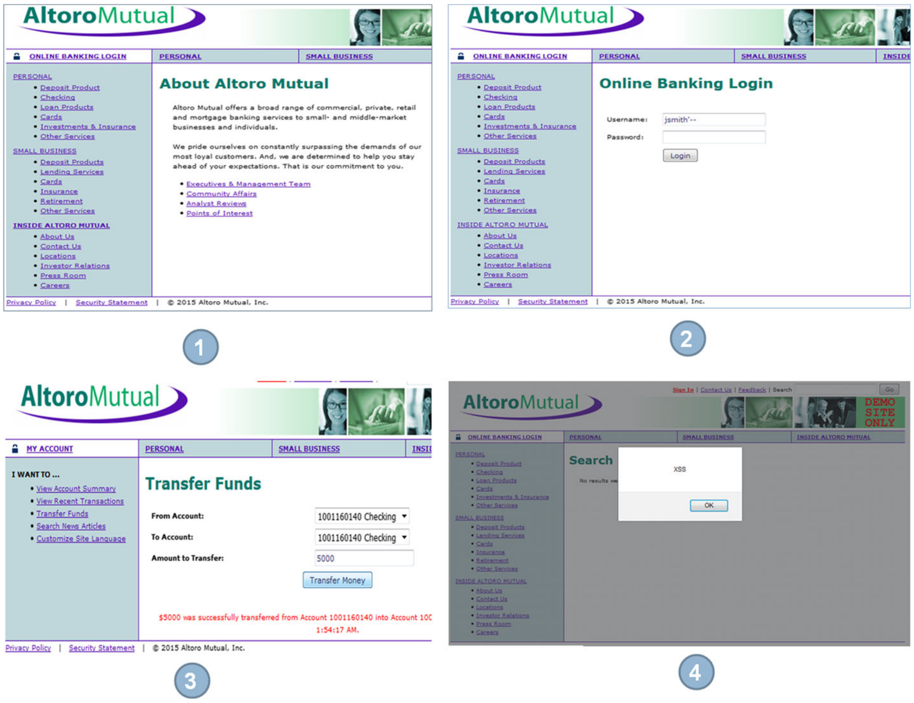
D-ForenRIA is an improvement over a previous version of the tool [10]. The extensions are based on a new distributed architecture, adding the ability of detection of complex user-input actions and actions that do not generate any HTTP traffic, and proposing a more efficient way of ordering candidate actions. We also report our experiments on six different RIAs.

1.1 Demonstration scenario for D-ForenRIA

We have implemented our proposed session reconstruction algorithm in a tool called *D-ForenRIA*. In order to illustrate the capabilities of *D-ForenRIA*, we have created a sample attack scenario, using a vulnerable banking application created by IBM for demonstration and test purpose (Fig. 2). In our case study, the attacker visits the vulnerable web site (part 1 in Fig. 2), uses an SQL-injection vulnerability to gain access to private information (part 2 in Fig. 2) and transfers some money to her own account (part 3 in Fig. 2). She also uncovers a cross-site scripting vulnerability that she can exploit later against another user (part 4 in Fig. 2). This session creates the trace depicted in Fig. 3. *D-ForenRIA* can help to recover what the attacker has done during the session from this input log.

```
[{ "tid": 3,
"action": "DataProvider",
"method": "getNodeTypes",
"type": "rpc",
"result": [ {
    "nodeType": "6",
    "cls": "typo3-pagetree-topPanel-button",
    "html": "<span class=\"t3js-icon icon icon-size-small ...",
    "title": "Backend User Section",
    "tooltip": "Backend User Section"
}], "debug": "" } ]
```

Fig. 1 The body of a typical HTTP response in a RIA (Adapted from TYPO3 RIA [42])



About Altoro Mutual

Altoro Mutual offers a broad range of commercial, private, retail and mortgage banking services to small- and middle-market businesses and individuals.

We pride ourselves on constantly surpassing the demands of our most loyal customers. And, we are determined to help you stay ahead of your expectations. That is our commitment to you.

Online Banking Login

Username: jsmith'--

Password:

Transfer Funds

From Account: 1001160140 Checking

To Account: 1001160140 Checking

Amount to Transfer: 5000

$5000 was successfully transferred from Account 1001160140 into Account 10C 1:54:17 AM.

Search

No results we

XSS

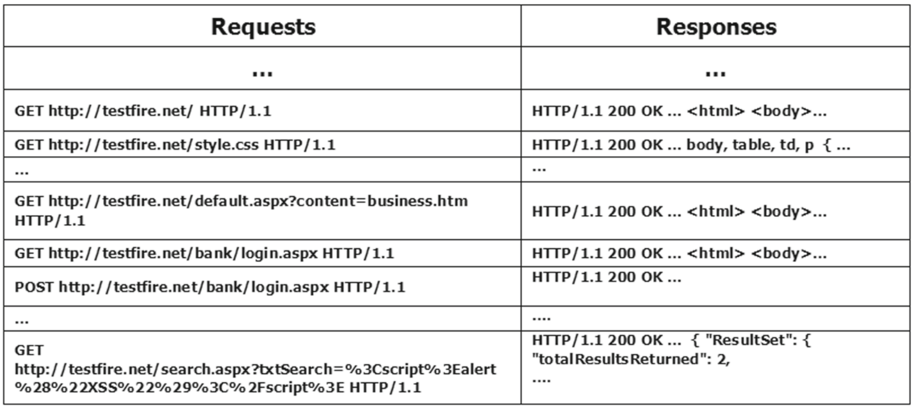
© 2015 Altoro Mutual, Inc.

Fig. 2 Screenshots of an attack scenario

Figure 4 presents the user-interface of *D-ForenRIA*[3]. To reconstruct this session, the user provides the trace as an input to the *SR-Proxy* (region 1 in Fig. 4). The users can configure different settings such as the output folder (region 2 in Fig. 4) and observe some statistics about the progress of the reconstruction (region 3 in Fig. 4).

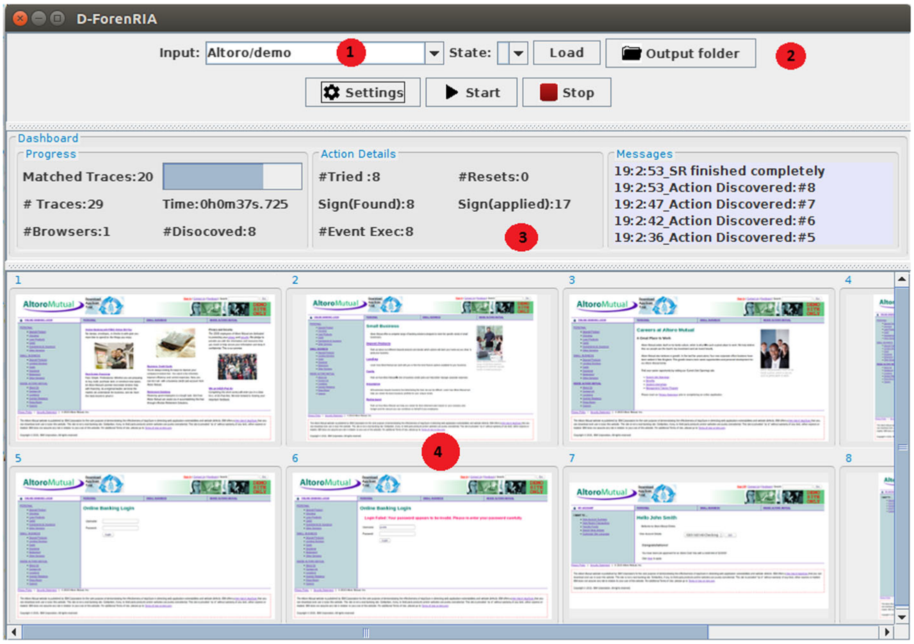
Given the full traces of this incident (Fig. 2), *D-ForenRIA* reconstructs the attack in a couple of seconds. The output includes screenshots of all pages seen by the hacker (region 4 in Fig. 4). To see the details of a recovered user action, a click on one of the thumbnails opens a new window. For example, Fig. 5 presents the details of the step taken for unauthorized login including the inputs hacker exploited for SQL-injection attack. The full reconstructed DOM can also be accessed from that screen.

A forensic analysis of the attack would have been quite straightforward using *D-ForenRIA*, including the discovery of the cross-site scripting vulnerability. Comparatively, doing the same analysis without our tool would have taken much longer, and the cross-site scripting vulnerability would probably have been missed. A demonstration of this



Requests	Responses
...	...
GET http://testfire.net/ HTTP/1.1	HTTP/1.1 200 OK ... <html> <body>...
GET http://testfire.net/style.css HTTP/1.1	HTTP/1.1 200 OK ... body, table, td, p { ...
...	...
GET http://testfire.net/default.aspx?content=business.htm HTTP/1.1	HTTP/1.1 200 OK ... <html> <body>...
GET http://testfire.net/bank/login.aspx HTTP/1.1	HTTP/1.1 200 OK ... <html> <body>...
POST http://testfire.net/bank/login.aspx HTTP/1.1	HTTP/1.1 200 OK ...
...	....
GET http://testfire.net/search.aspx?txtSearch=%3Cscript%3Ealert%28%22XSS%22%29%3C%2Fscript%3E HTTP/1.1	HTTP/1.1 200 OK ... { "ResultSet": { "totalResultsReturned": 2, ...

Fig. 3 Portion of the HTTP log for an attack scenario

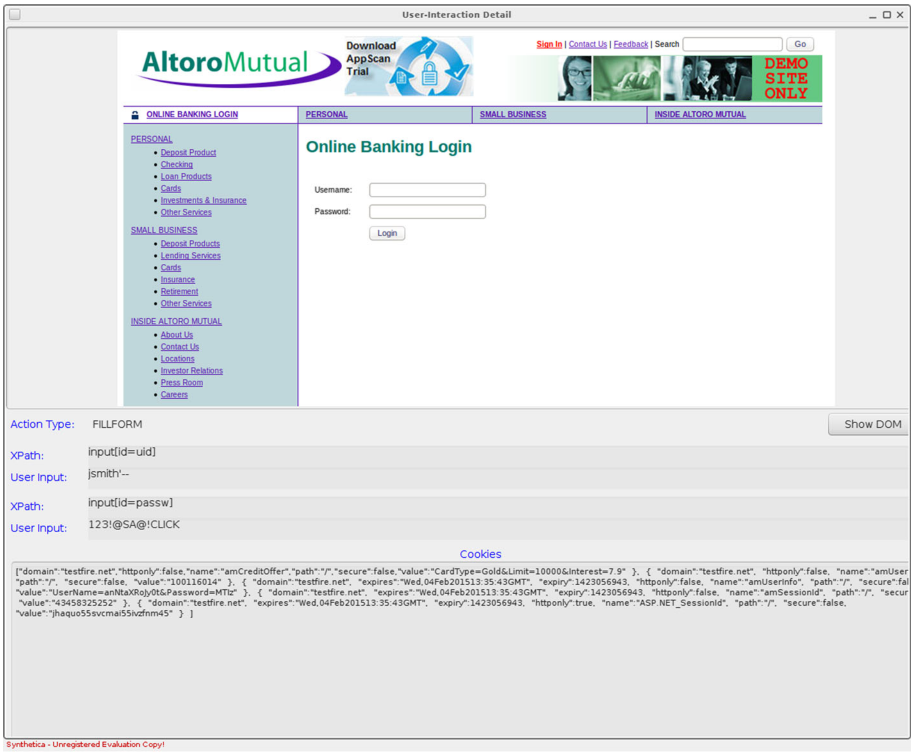


Matched Traces:20
Traces:29 Time:0h0m37s.725
#Browsers:1 #Disocoved:8

#Tried :8 #Resets:0
Sign(Found):8 Sign(applied):17
#Event Exec:8

```
19:2:53_SR finished completely
19:2:53_Action Discovered:#8
19:2:47_Action Discovered:#7
19:2:42_Action Discovered:#6
19:2:36_Action Discovered:#5
```

Fig. 4 Screenshot of the main window of *D-ForenRIA*



Action Type: FILLFORM

XPath: input[id=uid]

User Input: jsmith'--

XPath: input[id=passw]

User Input: 123!@SA@!CLICK

Cookies

["domain":"testfire.net","httponly":false,"name":"amCreditOffer","path":"/","secure":false,"value":"CardType=Gold&Limit=10000&Interest=7.9" }, { "domain":"testfire.net", "httponly":false, "name":"amUser
"path":"/", "secure":false, "value":"100116014" }, { "domain":"testfire.net", "expires":"Wed,04Feb201513:35:43GMT", "expiry":1423056943, "httponly":false, "name":"amUserInfo", "path":"/", "secure":fal
"value":"UserName=anNtaXRoJy0t&Password=MTIz" }, { "domain":"testfire.net", "expires":"Wed,04Feb201513:35:43GMT", "expiry":1423056943, "httponly":false, "name":"amSessionId", "path":"/", "secur
"value":"43458325252" }, { "domain":"testfire.net", "expires":"Wed,04Feb201513:35:43GMT", "expiry":1423056943, "httponly":true, "name":"ASP.NET_SessionId", "path":"/", "secure":false,
"value":"jhaquo55svcmai55ivzfnm45" }]

Fig. 5 Details of an SQL-Injection attack in *D-ForenRIA*

case study can be found on http://ssrg.site.uottawa.ca/sr/demo.html. In the remaining sections of this paper we are going to explain the details of *D-ForenRIA* and different techniques that are used.

1.2 Organization

The rest of this paper is organized as follows: In Section 2, we first model the session reconstruction problem in the context of client/server applications, and then in Section 3 we discuss the session reconstruction problem in RIAs and present a detailed description of our session reconstruction approach. We present some evaluation results that highlight the effectiveness of *D-ForenRIA* in Section 4. Then we discuss the state of the art in Section 5. Finally, we present our concluding remarks and future directions in Section 6.

2 Session reconstruction problem

The session reconstruction problem can be defined for any application based on a "client/server" architecture (such as a Web application or many mobile applications). In this section, we first explain the general case of the session reconstruction problem for client/server applications, and present a solution for this general case. Then, in Section 3 we explain our particular approach to reconstruct sessions of RIAs.

Figure 6 represents the context of a session reconstruction tool. A user interacts with a client and each interaction may generate one or more requests to the server (Fig. 6 part a). The server in turn processes these requests and sends some responses back to the client. The set of exchanged request/responses can be logged by the server or by other tools on the network for further analysis. The goal of session reconstruction is to find the sequence of user-client interactions of a session using the log of that session (Fig. 6 part b).

If we assume that there have been n user-interactions during a session, the trace of the session can be presented as $< rs_1, rs_2, \ldots, rs_n >$, where rs_i is the sequence of requests/responses that have been exchanged after performing the i^{th} interaction. rs_i can also be empty in the case that the interaction does not generate any requests/response. The goal of the session reconstruction tool is to find one (or more) sequence of interactions, that generate a sequence of requests/responses $< rs_1', rs_2' \ldots, rs_n' >$ that match the given trace. We are using a "Match" function and we are looking for a sequence of interactions such that $\forall_{1 \leq i \leq n}$ Match(rs_i, rs'_i). The *Match* function may be "strict", that is, two sequences match if they are equal. However, in practice, the *Match* function needs to be more flexible and ignore certain differences in the observed requests, rs'_i, and the expected traffic rs_i. For each user interaction, the session reconstruction tool also extracts the type of the action, and all the required information to perform that action.

We also assume that the application can be modeled by a finite state machine. We define the FSM as a tuple (S, s_0, I, δ) where:

- S is a set of states. Each state corresponds to a client-side state of the application. In RIAs, the states represent DOM instances.
- s_0 is the initial state of the application, when the session starts.
- I is the set of possible user-browser interactions with the application.
- $\delta : S \times I \rightarrow S$ is the transition function, where $\delta(s_i, a_j)$ refers to the next state of the application after the execution of a_j in state s_i.

It should be noted that the aim of the session reconstruction is not to extract the full state machine of the

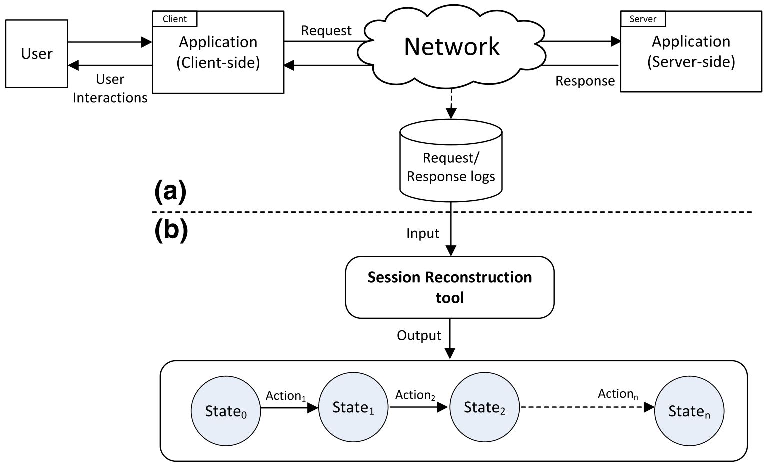


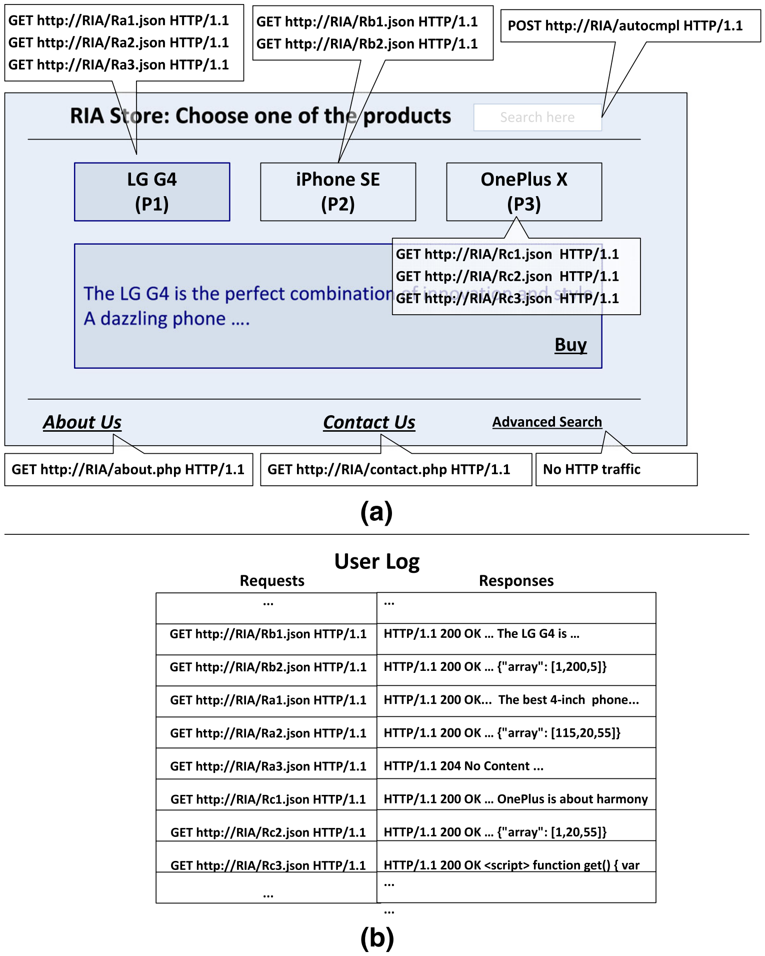
Fig. 6 a client/server architecture , **b** Session reconstruction tool (input and output) application. Therefore, the reconstruction problem does not suffer from the “state explosion” problem that is seen e.g. in the crawling problem [6]. Indeed, reconstructing a user session is to find a simple path in the state machine of the application, a path which corresponds to the user interactions during a session. In fact, the extraction of the full state machine would be impossible for a session reconstruction tool, since that the tool is off-line during the reconstruction and just operates based on a given set of recorded trace. A state that was not reached by the user during the session cannot possibly be reconstructed.

In the case of RIAs, the client is a Web browser which communicates using HTTP with a server. From the given HTTP log, we want to produce the set of client-side states and user-browser interactions during the session. Each client-state is defined by the DOM built by the browser when on that state. The DOM represents the structure of an HTML page including the elements of the page and their attributes. The DOM is also used to generate a screenshot of the page as seen by the user. In addition, the algorithm finds the XPath of the elements clicked, and the values that the user submits during the session.

Consider the example of Fig. 7. It shows a simple page of a RIA that displays a description of different products using AJAX (left), and a sequence of generated requests/responses during a session (right). Given the user-log of the session, $< Rb_1, Rb_2, Ra_1, Ra_2, Ra_3, Rc_1, Rc_2, Rc_3 >$, a working sequence of actions is $< Click(P_2), Click(P_1), Click(P_3) >$. This sequence of actions generates the given sequence of requests/responses.

Assumptions: To ensure the effectiveness of our session reconstruction method, the following assumptions are made about the target application, the input trace, and user-input actions.

- *Determinism*: It is assumed that the application is deterministic *from the viewpoint of the user.* It means that given a state and a given requests/responses sequence, the next client-state is uniquely defined[4]. If



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

POST http://RIA/autocmpl 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

No HTTP traffic

User Log

Requests	Responses
...	...
GET http://RIA/Rb1.json HTTP/1.1	HTTP/1.1 200 OK ... The LG G4 is ...
GET http://RIA/Rb2.json HTTP/1.1	HTTP/1.1 200 OK ... {"array": [1,200,5]}
GET http://RIA/Ra1.json HTTP/1.1	HTTP/1.1 200 OK... The best 4-inch phone...
GET http://RIA/Ra2.json HTTP/1.1	HTTP/1.1 200 OK ... {"array": [115,20,55]}
GET http://RIA/Ra3.json HTTP/1.1	HTTP/1.1 204 No Content ...
GET http://RIA/Rc1.json HTTP/1.1	HTTP/1.1 200 OK ... OnePlus is about harmony
GET http://RIA/Rc2.json HTTP/1.1	HTTP/1.1 200 OK ... {"array": [1,20,55]}
GET http://RIA/Rc3.json HTTP/1.1	HTTP/1.1 200 OK <script> function get() { var
...	... ...

Fig. 7 **a** A simple page and generated requests after clicking on each element. **b** Portion of the user’s log

the application is not deterministic from the viewpoint of the user, then when the session reconstruction tool tries an action that was actually performed by the user, it could reach a state that is not the state reached by the user. In this case, the tool will not be able to reconstruct the remaining actions. However, the application may still have randomness in the generated requests. This means that the execution of the same action from a given state, may generate a different sequences of requests and responses.

- *Input traces*: It is assumed that the session reconstruction tool has the log for a *single* user. Since the log on the server usually contains the trace for different users, the session reconstruction tool relies on other methods (such as [11]) to extract the traffic for a single user.
  In addition, the tool can read the unencrypted requests/responses and can decrypt the logs if it is encrypted. This assumption is necessary since the tool needs to compare the generated request after performing an action with the input log. We also assume that the input log is "complete"; this means that there is no need to have the traffic from the previous session to reconstruct the current session, and the log is recorded from the start of the session.
- *Access to the application during reconstruction*: It is assumed that the reconstruction is done off-line. This means that during the reconstruction process there is no access to the server, and the tool only exploits previously collected HTTP traces. This assumption ensures that the tool remains effective even when the server is not available (for example, because of an attack or a bug in the application). Moreover, replaying the session off-line without accessing the server provides a sandboxed environment which is especially desirable during forensic analysis.
- *User-Input Actions*: Regarding the actions that include input values from users, we have made the following assumptions; first, it is assumed that the input values passed into the generated requests are not encoded in a non-standard way; otherwise, the session reconstruction tool cannot recover the *actual* values entered by the user; the second assumption is regarding the *domain* of user-input values. It is assumed that the tool can produce acceptable values for a user-input action, using some preset libraries of possible inputs. This is necessary to be able to automatically input values that will not be blocked by client-side validation. Note that this does not mean that the tool should somehow guess the correct user inputs values. These values will be found in the log. Instead the tool should be able to provide *some* inputs that will be usable to continue with the session.

2.1 A general session reconstruction approach

Here we present an algorithm for the session reconstruction of applications which are based on the client/server model. The algorithm uses three components that participate in the reconstruction of a session: The *Client*, that can list/execute possible actions on the current state; A *Robot* that simulates user-client interactions, and a *Proxy* that replaces the actual server and responds to the requests sent by the client.

Algorithm 1 A general session reconstruction algorithm

Input: An input user log R, switch to find first/all solution(s) *findAll*
Output: The sequence of user-interactions that generate the given log, *sol*

```
 1: Procedure Main()
 2:     $sols \leftarrow \{\}$
 3:     $proxy \leftarrow$ InitProxy($R$)
 4:     $client \leftarrow$ InitClient($proxy$)
 5:     $robot \leftarrow$ InitRobot($client$)
 6:     $S_0 \leftarrow client$.GetState()
 7:     SR($S_0$, $R$, {})

 8: Procedure SR(State $S_n$, Trace $R$, ActionPath $ap$)
 9: if $R = \{\}$ then
10:     $sols$.add($ap$)
11:     if not $findAll$ then
12:         terminate()
13:     end if
14: else
15:     $A \leftarrow client$.ExtractCandidateActions($S_n$)
16:     for $a \in A$ do
17:         $R_s \leftarrow robot.Evaluate(a, S_n)$
18:         if $proxy$.Match($R_s$, begin($|R_s|$, $R$)) then
19:             $S_{n+1} \leftarrow client.GetState()$
20:             SR($S_{n+1}, R - R_s, ap + a$)
21:         end if
22:     end for
23: end if
```

The algorithm that is used to extract user-interactions, is shown in Algorithm 1. The *main* procedure takes care of initialization. The recursive session reconstruction procedure, SR, starts at line 8. In this approach, the algorithm extracts all possible candidate actions of the current state of the application S_n (line 15), and tries them one by one (line 17). Trying an action in S_n may change the state of the application to another state, S_{n+1}.

If the execution of action a generates a sequence of requests R_s which *Match* the requests/responses in the beginning of R (line 18), a is considered a possible correct action at the current state [5]. Since the requests can be generated in different orders, the order of elements in R_s does not matter for the *Match* function (line 18). The action is also accepted if it does not generate any requests (Such as clicking on the "Advanced Search" link in Fig. 7).

The algorithm then appends a to the currently found action sequence, and continues to find the rest of the actions in the remaining trace (line 20)[6]. The algorithm stops when all requests in the input trace are matched. The session reconstruction algorithm starts from the initial state of the algorithm (line 7). The output contains all solutions for the problem. Each solution includes a set of user-client actions (that matches the input trace). At each state, there may be several actions which are correct (line 18). In this case, the algorithm finds several correct solutions for the problem. However, in practice we can make the algorithm faster by adding a switch (parameter *findAll*) to stop the algorithm after finding the first solution (line 10).

During the execution of the algorithm, the *Client*, the *Robot*, and the *Proxy* collaborate to perform several tasks; the *Client* lists the possible actions on the current state (line 15), the *Robot* triggers the action on the current state (line 17) and the *Proxy*, responds to the requests generated during the executing the action by the client (line 17).

2.2 An improved session reconstruction algorithm

Although the general algorithm (Algorithm 1) is easy and straightforward, it should address different issues to be practical.

2.2.1 Signature-based ordering of candidate actions

Algorithm 1 blindly tries every action at each state to find the correct one (lines 16-17). However, in practice there may be a large number of candidate actions at a given state, so the algorithm needs a smarter way to order candidate actions from the most promising to the least promising. We propose to use a "Signature-based" ordering of candidate actions as follows.

In this technique, the algorithm uses the "Signature" of each action to sort the pool of candidate actions. The signature of an action is the traffic which has been generated when it was performed previously possibly from another state of the application (For example, in Fig. 7 the signature of clicking on P_2 is $\{Rb_1, Rb_2\}$). The signature of an action may also be discovered without the need to execute the action; for example, the signature of clicking on "contact-us" in Fig. 7 can be discovered by just looking at its *href* property in the DOM (Fig. 14). It is notable that the session reconstruction algorithm does not have the signature of all actions; the signature for an action is extracted once an action is evaluated for the first time.

To apply the signature-based ordering, the session reconstruction tool should be able to identify different instances of the same action at different states. We need to find an *id* such that this id remains the same in different states; therefore, in each state the session reconstruction tool calculates the id for each possible action and uses this id to find the signature of the action from previous states.

The signature-based ordering, assigns a priority-value $\in [0,1]$ to all candidate actions on the current state. This value is assigned based on how well the signature of the action, matches the next expected traffic; To this end, the *Match* function should return a value $\in [0,1]$, where a higher value means a higher match (e.g., 1 represents a full match and, 0 mean no match at all). When the signature of the action is not available (when the algorithm has not yet executed the action), the priority value of θ $(0 \leq \theta \leq 1)$ is assigned to the action. As a consequence, the algorithm first tries actions in a decreasing order of match until the match value reaches the threshold θ. Then, the algorithm tries actions that do not have any signature, and finally the least promising actions are tried (actions that have lower match than the threshold). If two actions have the same match value (e.g. they both fully match the next expected trace), the action with the longest sequence of requests, will have a higher priority.

Example: Consider the simple example in Fig. 7a and the given trace of Fig. 7b. In this example, we assume that clicking on a product displays some information about the product, but does not add any new possible user actions to the page. The threshold value, θ, of 0.5 is used when we do not have a signature of an action.

To apply the "signature-based" ordering, the algorithm assigns priority-values to candidate actions. At the initial state, the priority for the two *href* elements is minimum since their initiating requests (*about.php* and *contactus.php*) do not match the next expected requests, $< Rb_1, Rb_2, ... >$.

The priority-value for the remaining actions is *0.5* because the algorithm has not tried any action yet. Assume that actions are tried in the order P_1, P_2, P_3. The algorithm will try clicking on P_1 and P_2 to discover the first interaction (i.e. $Click(P_2)$). In addition, it learns the signature of clicking on P_1 and P_2. At the next state, clicking on P_1 gets the priority of *1* since its signature $< Ra_1, Ra_2, Ra_3 >$ matches the remaining of traces , $< Ra_1, Ra_2, Ra_3, Rc_1, \ldots >$, clicking on P_2 gets priority of *0* since its signature does not match and *P3* gets *0.5* since we have not tried this action yet. So, the correct action $Click(P_1)$ is selected immediately. At the third state, also $Click(P_3)$ gets a priority of *0.5* while $Click(P_1)$ and $Click(P_2)$ are assigned priority of *0*. At this state the correct action is selected immediately. To sum up, the 3 actions are found after trying 4 actions on the current state.

2.2.2 Concurrent evaluation of candidate actions:

Algorithm 1 is sequential and single-threaded. At each state the algorithm extracts the list of candidate actions (line 17), and executes them one by one using the client (the **for** loop in lines 16-22). The client needs to carry out several tasks to execute an action; it needs to initiate several requests, processes the responses, and update its state. These tasks can take a long time for the client to finish.

Therefore the total runtime can often be decreased by using several clients. After extraction of candidate actions (line 17), the algorithm assigns each action to a client in a new thread. The algorithm does not need to wait for the client to finish the execution of the action, and assigns the next candidate action to the next client. In this approach, several actions can be evaluated concurrently, which potentially decreases the runtime of the algorithm.

2.2.3 Extracting action parameters

The approach needs to extract all candidate actions on each state. For each action, we need to extract all the information required to execute that action (we call such information the *parameters* of the action). For a *click* action, the only required parameter is the element that is the target of click. However, for actions that involve *user-inputs*, more parameters must be determined: First, the set of input elements, and second, the values that are assigned to these elements (value parameters). We assume that the *client* can provide us the list of input elements at each state. To detect value parameters of user-input actions, we propose the following approach:

1. At each state, the system performs each user-input action using an arbitrary set of values, x. These values are chosen from the domain of input elements in that user-input action. The system observes requests T after performing the user-input action.
2. If the next expected traffic is exactly the same as T but with different user-input values y instead of x, the system concludes that the user has performed the user-input action using y.

Example: The text-box on top of the example in the Fig. 7 is used to search in the sample site. To detect this user-input action, the algorithm fills the text-box using a predefined value (here “sampleText”) and compares the generated request with the next expected request. Since these two requests (Fig. 8a, b) are quite similar, and the only difference is the submitted value, the algorithm determines that the user has performed this action using a different value, and resubmits the action using the correct value, namely “IPhone”.

In this technique, the algorithm does not need to know anything about how the client-side formats the user input data, and it learns the format by trying an action and observing the generated traffic. However, this technique is only effective if the user-input data is passed as is; if there is any encoding of the submitted data, the actual data that has been used by the user cannot be extracted from the logs.

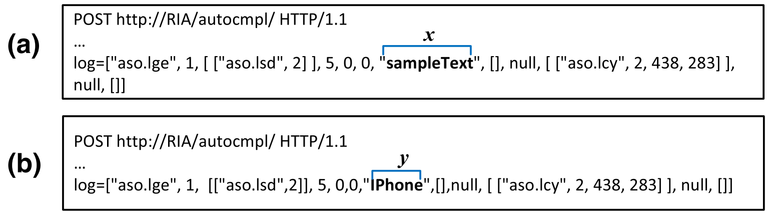
2.2.4 Handling randomness in requests

We have assumed that performing an action from the same state, may generate a sequence of requests/responses that includes some randomness. Both the client-side and the server-side of the application can contribute to this randomness. The client-side of the application can generate different *requests* after performing an action from the same state, and the server-side may respond with different *responses*.

The responses are served by the proxy by replaying a recorded trace. Therefore, there will be no randomness in the responses during the reconstruction. However, the session reconstruction algorithm still needs to handle randomness in the client-side generated requests.

If the execution of an action generates random requests, the algorithm cannot detect the correct action since executing the action generates requests which are different from the requests in input trace. The *Match* function (line 18 in Algorithm 1), needs to detect the existence of randomness and flexibly find the appropriate responses to the set of requests.

There are different forms and variations for the randomness in the generated requests; for example, the actual order of execution of a series of concurrent requests/responses can change. In this case, as we explained in Section 2.1, the *Match* function ignores the order of generated requests to find the matching responses. Requests in a client/server application usually contain some parameters (such as query-string parameters in a Web application); another form of randomness happens when the values of these parameters changes between two executions, for example when the value is dependent on the current time, or when the value is based



```
POST http://RIA/autocmpl/ HTTP/1.1
...
log=["aso.lge", 1, [ ["aso.lsd", 2] ], 5, 0, 0, "sampleText", [], null, [ ["aso.lcy", 2, 438, 283] ],
null, []]
```

```
POST http://RIA/autocmpl/ HTTP/1.1
...
log=["aso.lge", 1,  [["aso.lsd",2]], 5, 0,0,"IPhone",[],null, [ ["aso.lcy", 2, 438, 283] ], null, []]
```

Fig. 8 a The generated HTTP request after performing a user-input action using sample data. **b** the expected HTTP traffic in the trace. (*x*, *y* represent the sample and actual user-input values respectively) on a randomly generated value. To handle randomness of this type, we use the following approach:

We say that two requests match *partially* if they have the same URL, the same set of parameters, but some of these parameters have different values. For example, the requests

req.php?language=EN&user=john&time=12345 and
req.php?language=FR&user=john&time=56745

match *partially* because they are both to *req.php*, they have the same three parameters *language*, *user* and *time*, but the values of parameters *language* and *time* are different.

Suppose that once the system executes action a from state S_n (line 17 in Algorithm 1), it observes that the generated requests have a *partial* match with the next expected traffic. This difference between parameter values can have two causes:

1. Two different actions may send requests containing the same parameter but with different values. For example in a news RIA if we have two actions, one for fetching the latest technology news sending requests to *fetch.aspx?cat=tech* and another action for fetching the business news requesting *fetch.asp?cat=business*, in this case, the parameter *cat* differentiates between two types of actions. We call the parameter *cat* in the previous example a "constant" parameter since its value differentiates between different actions and the value will be the same for the same action from the same state.
2. Requests for the same action, contain a parameter that has a changing value after each execution. For example, in a news RIA there may be an action to fetch the latest news that sends a request to *latest.aspx?last=12_15_20*. In this request, the value of parameter *last* represents the time of the last update fetched by the client and is changed *by the client* every time a request is sent. The next request might look like *latest.aspx?last=12_16_20*. We call parameter *last* of the previous example a "non-constant" parameter since its value changes at each execution of the same action from the same state.

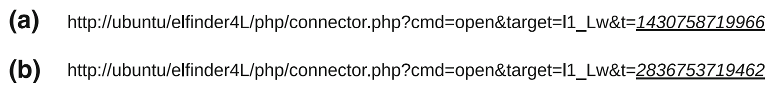
When the session reconstruction tool observes that the value of a parameter in the log differs from the value of the same parameter in the generated request after execution of an action, it should categorize the parameter as constant or non-constant. If the parameter is constant it means that the action is incorrect (that is, this is not the action taken by the user at that level of the reconstruction). For example, it would be the case if the user had clicked to fetch latest technology news, but the session reconstruction tool had tried fetching business news at this state. However, if the parameter is actually a non-constant one, its value should be ignored during comparison. For example, the value of the *last* parameter in the previous example is a non-constant, therefore if the tool triggers an action which generates *latest.aspx?last=12_16_20* and we expect *latest.aspx?last=12_15_20* in the log, we have actually selected a correct action since the value of the parameter *last* is non-constant and should be ignored.

A naive approach to handle randomness in requests, would be to compare the generated request with a request in the user-log based on a similarity function [7]; if the similarity between two requests is more than a threshold, the requests are considered a match. However, this approximate matching may mistakenly match two requests and jeopardize the reconstruction. Therefore, our solution does not depend on a threshold and automatically verifies the randomness in requests' parameters as follows:

If session reconstruction tool tries action a, in order to verify that a actually generates a request with non-constant parameter values the system performs a from S_n again; if the system observes a change in the value of the same parameters, it concludes that the value of those parameters are variable. The *Match* function does not consider these changing values for checking the correctness of the action. In the example above, the actual value of the parameter *last* will be ignored by the *Match* function for that request.

Example: In Web applications, the requests are sent for a resource (with a URL), and each request can contain some parameters and their corresponding values (in the query string of a *GET* request, or in the body if a *POST* request). Two HTTP requests can be considered a *partial* match if they are sent to the same resource, have the same set of parameters, but the values for some parameters are different.

Figure 9 depicts an example of a request which contains a parameter, *t*, which gets a different value each time requested by the browser. By comparing the request received in the first execution (part a in Fig. 9) with the



(a) http://ubuntu/elfinder4L/php/connector.php?cmd=open&target=l1_Lw&t=*1430758719966*

(b) http://ubuntu/elfinder4L/php/connector.php?cmd=open&target=l1_Lw&t=*2836753719462*

Fig. 9 a,**b**: Two HTTP requests that include a parameter with a changing value one received in the second execution (part b), the system detects that parameter t has changing values and ignores the value of t later during the reconstruction.

This technique works well if the randomness just happens in the values of request's parameters; however, there are some variation of randomness that cannot be easily handled by this approach; for example, when the number of requests that are generated after performing an action, or the number of parameters in a request are non-deterministic. Another important example of limitation is when the changes are slow, for example when the values are changed based on the date: it won't match the recorded traffic, but consecutive execution yields usually the same values.

2.2.5 *Handling non-user initiated requests:*

Requests that are exchanged between a server and a client can originate from different sources. In Algorithm 1, we have discussed user-initiated requests, however messages can also be sent without the user initiating a request first. These requests may originate from a *timer* on the client-side, or even from the server-side (such as Websockets[7] in RIAs). The general session reconstruction algorithm can be modified to handle these cases, as follows:

- *Timer initiated requests*: Timers can be detected based on the signature-based ordering of actions (Section 2.2.1); Timers are also one of the possible actions in the current state, so the algorithm first detects them on the current state (line 15 in algorithm 1) and evaluates each timer (line 17 in algorithm 1) to find the timer's signature. Later during the reconstruction, the algorithm triggers a timer when the signature of the timer matches the next expected traffic.
- *Server-initiated requests*: In client/server applications, sometimes the server needs to send some data to the client. In this case, the given trace to the proxy contains both requests that originate from the client-side of the application, and requests that are sent from the server. The *proxy* in our general algorithm has to be changed to detect these server-initiated requests; When the proxy observes that the next expected traffic is server-initiated, it just *sends* these requests to the client.

3 D-ForenRIA: a session reconstruction tool for RIAs

In Sections 2.1 and 2.2, we presented a general and improved algorithm for the session reconstruction problem. In this section, we propose a session reconstruction approach for RIAs. This approach realizes the improved session reconstruction algorithm in the context of RIAs and addresses several challenges mentioned in the previous section.

Our solution is implemented in a tool called *D-ForenRIA*. In this section, we first present the most important components of *D-ForenRIA*, then we describe the messages exchanged between these components, and finally explain the details of each component.

3.1 Architecture of D-ForenRIA

Figure 10 presents the architecture of *D-ForenRIA* with two main components: A "Session Reconstruction Proxy" (SR-Proxy) and a set of "Session Reconstruction Browsers" (SR-Browsers). SR-Browsers are responsible for loading the DOM and identifying/triggering actions (they combine the role of "robot" and the role of "client" presented in Section 2.1). The SR-Proxy performs the role of the original Web server, and responds to SR-Browsers' requests from the given input HTTP trace (that is, the role of "proxy" in Section 2.1). This ensures that during reconstruction the SR-Proxy just uses a previously recorded trace and has no access to the server. Based on our session reconstruction algorithm, we infer which user-browser interactions are performed during the session.

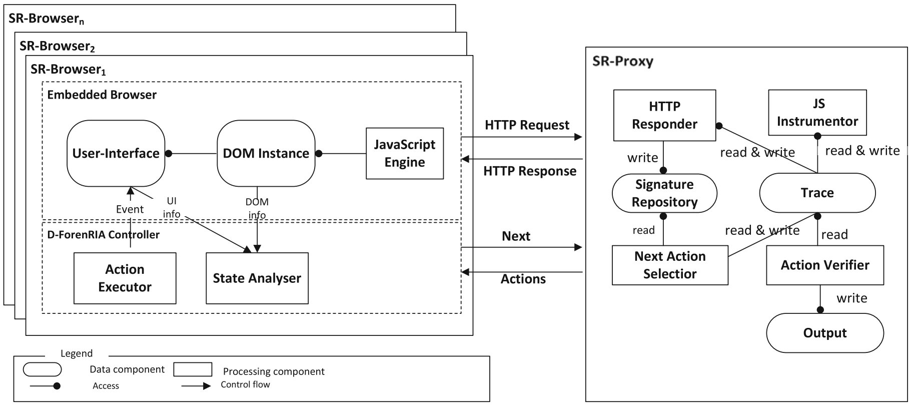


Fig. 10 Architecture of *D-ForenRIA*

D-ForenRIA works based on how Web applications work; during a session, the client does not know anything about the application, and user-interactions with the application generate a sequence of requests/responses that are accumulated in the browser. Therefore, during the replay, if the client is being fed with the same set of requests/responses by triggering the same user-interactions, we should be able to reconstruct the session.

In other words, we force the browser to issue the request related to the initial page in the trace; We feed the browser with the response for the initial page from the given trace. Through the natural rendering of this response, the browser will issue other requests (e.g. for images, JavaScript code) until the page loads completely. These requests are served by *D-ForenRIA*'s proxy from the given trace. During the reconstruction, the browser tries different actions and the SR-Proxy verifies the generated requests. When the requests generated after performing an action match the next unconsumed traffic, the SR-Proxy feeds the browser with the corresponding responses; this process continues until all HTTP traffic has been consumed.

3.1.1 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 Fig. 11). We call this repetitive process *iteration*. Figure 11 provides an illustration of the sequence of messages exchanged between the main components. The messages are exchanged in the following order:

1. At each iteration, the *SR-Browser* sends a *"Next"* message, asking the *SR-Proxy* the action to execute next.
2. The *SR-Proxy* asks the first *SR-Browser* reaching the current state to send the information about the state. This information includes list of all possible actions on the current DOM, and other information about the DOM such a screenshot of the rendered DOM.
3. The *SR-Browser* extracts the state information, and sends it to the *SR-Proxy*.
4. The *SR-Proxy* orders the list of candidate actions (using the signature-based ordering in Sections 2.2.1, 3.3).
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 *"ExecuteAction(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.

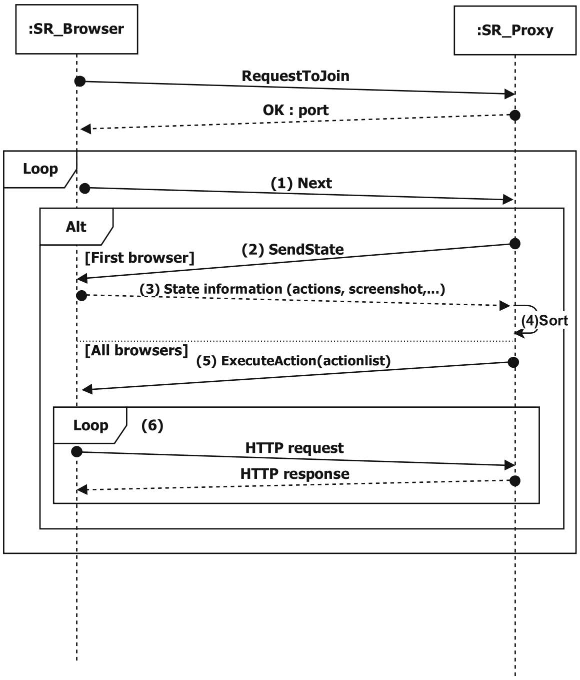


Fig. 11 Sequence diagram of messages between an SR-Browser and the SR-Proxy

3.1.2 SR-Browsers' and SR-Proxy's components

SR-Proxy and SR-Browsers have the following components which collaborate to reconstruct the session. In SR-Browsers, we have:

- The Embedded browser: A real Web browser (e.g., Firefox) which provides the ability to manipulate and access to the RIA's client states.
- The Controller: The controller is responsible for sending the "Next" messages to the SR-Proxy asking the action to be executed next.
  The controller also has the following components:
  - The Action Executor: Used to execute actions on the current state (e.g., clicks, form filling, timers). The execution is done by triggering an event of a DOM element; the corresponding event-handler is being executed and may use the JavaScript engine to complete the execution; The execution usually updates the user-interface of the embedded browser.
  - The State Analyzer: The analyzer is responsible for gathering information about the current DOM, such as the list of event handlers in the current DOM. In addition, it is used to extract information (such as the screenshots of the current DOM) when a new state is found. Furthermore, the analyzer checks whether the DOM has been updated completely after an action is being executed by the "Action Executor".

In the SR-Proxy, we have:

- The HTTP Responder: The HTTP responder replies to the stream of HTTP requests coming from embedded browsers. The previously recorded HTTP trace is given as an input to the SR-Proxy.
- The JS Instrumentor: The JS Instrumentor (JavaScript Instumentor) modifies the recorded responses before sending them back to the browser. This instrumentation is done to inject some JavaScript code to be executed on the embedded browsers to keep track of the event handlers in each state (Sections 3.2, 3.4).
- The Next Action Selector: This component keeps track of previously tried actions and uses this knowledge to choose which candidate action should be tried next.
- Signature Repository: The signature repository stores all detected signatures of actions once tried by an SR-Browser.
- The Action Verifier: This component confirms whether or not a performed action matches the expected traffic in the log. If it is the case, it updates the output. The output contains all the required outputs of the session reconstruction process. This 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.

D-ForenRIA is based on a set of SR-Browsers that can be dynamically added or removed during the reconstruction process. This architecture allows the concurrent execution of several actions at each RIA's state.

3.1.3 SR-Browsers' and SR-Proxy's algorithms

Based on this architecture, our simplified reconstruction algorithm executed by the SR-Browsers and SR-Proxy can be sketched as shown in Algorithms 2 and 3. We briefly overview the gist of the approach below, before providing details in the subsequent sections.

Algorithm 2 Sketch of the SR-Browser Algorithm

```
input: SR-Proxy's address
 1: HandShake(SR-Proxy)
 2: e ← AskforNext(SR-Proxy)
 3: while e ≠ finish do
 4:     if e is "ExecuteAction" request then
 5:         ActionExecutor.ExecuteAction(e);
 6:         StateAnalyzer.WaitForStableCondition();
 7:     else if e is "SendState" request then
 8:         s ← StateAnalyzer.GetStateInfo()
 9:         Send(SR-Proxy, s)
10:     end if
11:     e ← AskforNext(SR-Proxy)
12: end while
```

SR-Browser: Algorithm 2 specifies the steps executed by SR-Browsers. An SR-Browser first handshakes with the SR-Proxy. Then, there is a loop of interactions between the SR-Browser and the SR-Proxy; at each iteration, the SR-Browser asks the SR-Proxy what to do next (lines 2 and 11). The SR-Proxy can provide two answers. It either asks the SR-Browser to execute a set of actions, or it asks the SR-Browser to send back the state information. When the SR-Browser executes an action via the *Action Executor* (line 5) a stream of HTTP request/responses are exchanged between the *browser* and the SR-Proxy. The SR-Browser waits until all responses have been received, and makes sure that the DOM gets settled (line 6). In addition, when the SR-Browser discovers a correct action and a new state, the proxy requests the SR-Browser to send the state information to update the output (lines 14, 16). The state information includes the screenshot, the DOM, cookies of the current DOM and most importantly, the list of all candidate actions on the current DOM. The loop continues until all interactions are recovered.

Algorithm 3 Sketch of the SR-Proxy Algorithm

```
Input: set of logs Traces
Output: set of actions output
 1: Procedure MAIN()
 2:     output ← {}
 3:     while not finished do
 4:         SR-B ← HandShakeSRBrowser()
 5:         HandleSRBrowser(SR-B)
 6:     end while

 7: Procedure HandleSRBrowser(SR-B)
 8: while not finished do
 9:     req ← GetRequest(SR-B);
10:     if req is an "HTTP" message then
11:         HTTPResponsder.respondToHTTP(SR-B,req);
12:     else if req is a "Next" message then
13:         if ActionVerifier.Match(SR-B.lastRequests) then
14:             s ← SendState(SR-B)
15:             c ← SortCandidateActions(s, SignatureRepository)
16:             output.Add(SR-B.lastAssignedAction)
17:         else
18:             e ← NextActionSelector.ExtractNextCandidate
                Action(SR-B, Traces, c)
19:             ExecuteAction(SR-B, e)
20:         end if
21:         SignatureRepository.record(SR-B.lastAssignedAction,
            SR-B.lastRequests)
22:     end if
23: end while
```

SR-Proxy: The algorithm used by the SR-Proxy is shown in Algorithm 3. The *main* procedure (lines 1-6) waits for SR-Browsers to send a handshake. Since *D-ForenRIA* has a distributed architecture, SR-Browsers can join at any moment during the reconstruction process. After joining of a new SR-Browser, the SR-Proxy spawns a new thread to execute the *HandleSRBrowser* method (lines 10-26). This method assigns a new port to the newly arrived SR-Browser and responds to SR-Browser's messages. If an SR-Browser sends a normal HTTP request (line 10), the *httpresponder* attempts to find that request in the traces. Otherwise, it is a *next* message and the SR-Proxy needs to decide what actions the SR-Browser will do next. To do so, the SR-Proxy first verifies whether or not the last assigned action to this SR-Browser has generated requests/response that match the expected HTTP traffic (line 13). If it was the case, a new correct action and state have been recovered, and the SR-Proxy asks the SR-Browser to send its current state information (including the list of candidate actions) (line 14). The SR-Proxy then sorts these candidate actions from the most to the least promising based on the signature of the actions (line 15), and adds the newly discovered action to the output (line 16).

If the action executed by the SR-Browser did not generate the expected traffic (for example when the action were not triggered during the session), the *next action selector* chooses another action from the pool of candidates, and assigns it to the SR-Browser (lines 18-19). In all cases, the SR-Proxy also records the requests generated by an action in the *signature repository*. The information in the signature repository helps later when deciding if this action should be tried again (line 21). It is also notable that the SR-Proxy in *D-ForenRIA* is just used during the reconstruction and not for recording the traffic. The main steps of the session reconstruction procedure are explained below.

3.2 Extraction of candidate actions

In *D-ForenRIA*, after a state is discovered, the SR-Proxy assigns to the browser who executed the right action 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, and tried concurrently by SR-Browsers until the correct action is found.

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 Fig. 12, *FetchData(0)* is the event-handler for the *onclick*

```
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'); }
}
```

Fig. 12 A simple JavaScript code snippet

event of P_1. The existence of this event-handler means that there is a candidate action "*Click* P_1" on the current DOM.

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*:
   RIAs can also add event handlers dynamically, using some JavaScript code to assign handlers to HTML elements. JavaScript libraries (such as jQuery[8], Prototype and MooTools[9]) also provide their specific APIs to add event handlers[10]. One approach to find event listeners attached by these libraries is to parse the event information out of each JavaScript library. However, this approach requires calling the library's specific API to find the list of event handlers (such as calling the *retrieve("events")* method of an element in MooTools). This approach requires to know the API of the JavaScript libraries used in the reconstructed RIAs; since there are many such libraries used by RIAs this approach is not sustainable.
   A more effective approach for finding dynamic handlers is to account for the fact that no matter which libraries the RIA uses for assigning handlers, they all eventually call the JavaScript's *addEventListener* function [12] in the background. Therefore, by keeping track of *addEventListener* calls one can find every dynamically added event listeners. Since this approach is comprehensive, it is used in *D-ForenRIA*. As shown in Fig. 13, *D-ForenRIA* overrides the built-in *addEventListener* function 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 [13] and is realized by the *JavaScript instrumentor* that injects the code in the responses sent to SR-browsers from the SR-Proxy (shown in Fig. 13). Note that because our code is injected in a way to ensure that it will execute last, the hijacking will work even if the RIAs itself hijacks the same methods in the same way[11].

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" [14]. 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 element. In this case, it is hard to find the elements which trigger the event and are handled by the parent's event handler. To alleviate this issue, elements with an assigned event-handler are tried first. Then, *D-ForenRIA* tries elements without any event-handler starting from the bottom of the tree assuming that leaf elements are more likely to be elements triggering the event.

3.3 Efficient ordering of candidate actions (SR-Proxy)

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). *D-ForenRIA*, uses the signature-based ordering to order candidate actions at each state. As we discussed in Section 2.2, the signature based ordering is based on learning the signature of each action. In the case of RIAs, the signature of actions can be explicitly determined from the attributes of HTML elements that are involved in an action (such as the *href* attribute of a link), or determined once *D-ForenRIA* tries an action during the session reconstruction. *D-ForenRIA* assigns the signature-based scores to all elements on the current DOM. The SR-Proxy also remembers the signature of each action during the execution (line 21 in Algorithm 3).

In addition to signature-based ordering, *D-ForenRIA* also minimizes the priority of actions that involve elements with which users rarely interact; such as actions that involve elements that are invisible, have no event handler (Section 3.2), or have tags with which users usually do

```
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;
```

Fig. 13 Hijacking the built-in JavaScript AddEventListener function to detect dynamically assigned handlers

not interact (e.g. *script*, *head*). For example, in the DOM of Fig. 14, the *meta, hr* and *body* elements have low impact tags and therefore clicking on them are given the lowest priority. Three *div* tags are also assigned a lower priority value because one is hidden and the other two have no handler attached, respectively.

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 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 some timeout based function, it asks an SR-Browser to trigger that function.

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 *Web-Sockets* which creates a bidirectional non-HTTP channel between the client and server. Currently, *D-ForenRIA* supports polling but not Web-Sockets or Long-Polling. This approach, however, has an important limitation; the technique assumes that the generated requests after triggering of the timer remain the same. Therefore, if the timer handler generates changing requests, the technique becomes ineffective.

Example: Assume that our example in Fig. 7 also has a timer which registers itself using a setInterval call to fetch the latest news from the server (Fig. 15). This timer has an interval of one minute and needs at least a minute to be triggered. During the reconstruction, the timer needs to be triggered at the right time to match the trace. To address this problem, as we described here, *D-ForenRIA* detects timer callbacks and calls them at the right moment.

Figure 16 presents a session of a user with our example. This session lasts two minutes and includes {Click P_1, Timer Callback, Click P_2, Click P_3, Timer CallBack} events. When *D-ForenRIA* loads the application, it detects the existence of the timer and executes the callback function to find its signature. After detection of "Click P_1", it finds that the next expected traffic matches the signature of the timer and asks the SR-Browser to trigger the callback function of the timer. The callback is called later again, after the detection of the next two actions (Click P_2 and Click P_3) as well.

3.5 Detection of user inputs (SR-Proxy)

User inputs are an essential part of any user session. There are two steps in each user input interaction: First, the user enters some values in one or more HTML inputs, and second, the application sends these values as an HTTP request to the server. The standard way to send user inputs is using HTML forms. In HTML forms [15], each input element has a *name* attribute and a value. The set of all name-value pairs represents all inputs provided by the user. To detect user inputs submitted using HTTP forms, *D-ForenRIA* takes the following approach: First,

```
<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>
<div id="news"></div>
<input type="text" id="srch" onkeypress='autocmpl(event);'>
<span>Advanced Search</span>
</body>
```

Fig. 14 A simple DOM instance

```
function updatenews() {
$.get( 'latestnews.json',
function( data ,status ) {
$('#news').html(data); }, 'text');
}
setInterval(updatenews, 60000);
```

Fig. 15 A timer registered using setInerval to fetch the latest news

when an SR-Browser is being asked to extract actions, it detects all form elements on the current DOM as candidate actions (Section 3.2). The SR-Proxy then compares the next expected HTTP request with the candidate form submission actions. If the next HTTP request contains a set of name-value pairs, and the set of names matches *name* of the elements inside the form, SR-Proxy identifies the user input action and asks the SR-Browser to fill the form using the set of corresponding values found in the log.

In addition to *forms*, in RIAs any input element can be used to gather and submit the data to the server. Furthermore, input data are usually submitted in JSON format, and there is no information about the input elements inside the submitted request. Therefore, by simply looking at an HTTP request, it is no longer possible to detect whether the request belongs to a user-input action or not.

To detect user input actions that are not submitted using forms, *D-ForenRIA* uses the method proposed in Section 2.2. SR-Browser considers all input fields (i.e., *input*/*select* tags) that have an event-handler attached, and are nested inside a *form*/*div* element as candidate user-input actions. The SR-Browser then needs to decide which values to put in input fields. For some input element types (such as *radio*, *select*) it is easy to choose appropriate input values since the element's attributes contain the set of possible valid inputs (such as *option* tags inside a *select* element). For some types of input elements (such as a textbox intended to accept an email address), putting a value which does not match the accepted pattern may prevent submission of user input values to the server (because of the client-side validation scripts). In this case, we assume that *D-ForenRIA* has a dictionary of correct sample values for different input types. *D-ForenRIA*, also takes advantage of the new input elements introduced in HTML5 (elements such as *email*, *number* and *date*), to more easily assign correct values to input element.

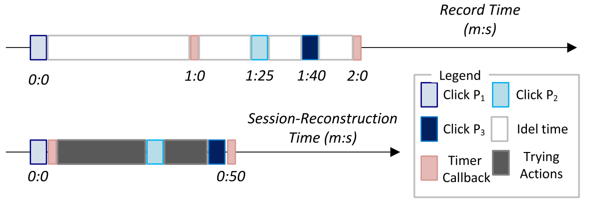


Fig. 16 A session with a timer: time flows from left to right along the axis (top) at Recording, and (bottom) at Session-Reconstruction time

3.6 Checking the stable condition (SR-Browser)

The SR-Browser usually needs to execute a series of actions as decided by the SR-Proxy in response to a "*Next*" message. After executing each action, an SR-Browser should wait until that action is completed and the application reaches what we call a "stable condition". In the stable condition, no more requests are going to be generated without new user interaction, and the DOM is fully updated. This condition must be met, otherwise the SR-Browser may try to execute the next action too early, an action that is not yet present on the DOM. To check the stable condition, an 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 the *XMLHttpRequest*'s *send* and *onreadystatechange* functions. The first function is called automatically when a request is being made and the second function can be used to detect when the browser fully receives a response.
- *Existence of the Action on DOM*: When there are no more pending requests, the system waits for the elements involved 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.

3.7 Loading the last known good state (SR-Browser and SR-Proxy)

When an SR-Browser performs an action on state *s*, and this execution does not generate the expected traffic, the SR-Browser needs to return back to some state (most probably *s*) as instructed by the SR-Proxy. *D-ForenRIA* uses a *reset* approach to transfer the client to a given state. To return back to the previous step, the SR-Proxy asks the SR-Browser to *reset* to the initial state and execute all previously detected actions [16].

As an alternative to the *reset* technique, there are approaches to *save*/*reload* the state of the browser. Saving/reloading the state, however, needs some information regarding the browser's internal implementation to get access to memory structures [17]. Therefore, save/load techniques are dependent on the browser's type, and there is no standard way to implement this idea. On the other hand, *D-ForenRIA*'s *reset* approach relies just on JavaScript execution and is supported by all browsers. However, one important limitation of the *reset* technique is that it can be time-consuming, particularly when there is a long sequence of previously detected actions.

3.8 Detection of actions that do not generate any HTTP request

When the reconstruction is done using only the previously recorded traffic as input, actions that do not generate any traffic can present a problem. In RIAs, actions may not generate any HTTP traffic because of caching or because the action just changes the DOM without requesting any resource from the server. To detect such actions, we use an auxiliary structure called "Action Graph". In this graph we define nodes and edges as follows:

- **Nodes**: Each node represents an action which is possible in some state of the RIA. Each node also contains some information about the set of HTTP requests/responses generated by the action.
- **Edges**: There is an edge between two nodes *a* and *b*, if there is a state from which *a* is possible and after performing a, *b* is available on the current state (probably a new state) of RIA.

Let node *C* be any currently enabled action; *D-ForenRIA* uses the following procedure to find the next action:

First, it checks all nodes of the graph to see if the signature of any action matches the next expected traffic. Suppose that we find such an action *D*. If *D* is present on the current DOM, we can immediately trigger that action. However, we may also accept *D* even if it is not present on the current DOM: This case happens when we find a path *CXD* from *C* to *D*, and we are sure that no action in *CX* generates any HTTP traffic. *D-ForenRIA* assumes that an action is not going to generate any traffic in two cases: first, if an action has not generated any traffic in a previous execution, and second, if all the generated requests in a previous execution contain HTTP headers that enable caching (such as *Cache-Control:public* headers [18]).

Example: Consider the example in Fig. 7. In this example, we have a node for each tried action during the reconstruction. Figure 17 presents the current state of the corresponding action graph. Suppose that action "Buy P_1" is enabled at the current state, and the next expected traffic is "*GET checkout.php*" which is the signature of clicking on the submit button when the user orders a product. Here the path: {Buy P_1, Submit} is valid since "Submit" matches the next expected request, and "Buy P_1" is known to not generate any traffic and it can be executed at the current state.

4 Experiments

To assess the effectiveness of the proposed session reconstruction system, we have conducted several experiments. These experiments are not exhaustive and are meant to evaluate the ability of *D-ForenRIA* to overcome the challenges detailed in the previous sections. As will be explained in Section 4.1, our test RIAs have been selected because they each present some of these challenges, and because they use a range of popular JavaScript libraries.

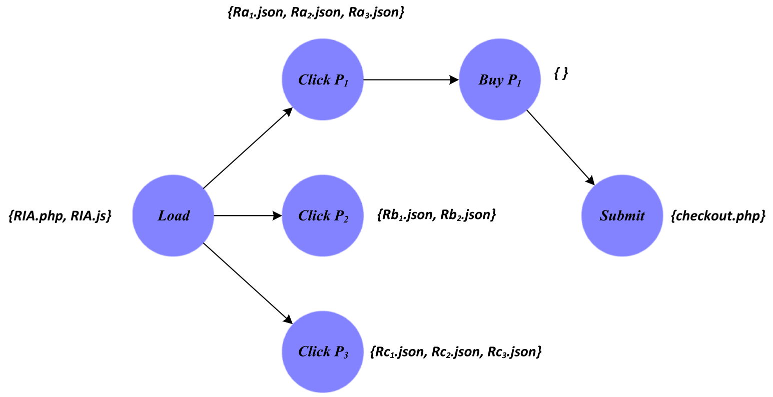

{Ra_1.json, Ra_2.json, Ra_3.json}
Click P_1
Buy P_1
{ }
{RIA.php, RIA.js}
Load
Click P_2
{Rb_1.json, Rb_2.json}
Submit
{checkout.php}
Click P_3
{Rc_1.json, Rc_2.json, Rc_3.json}

Fig. 17 Portion of the action graph of Fig. 7

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 should 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[12].

4.1 Test 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. [7]). 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. Finally, C6 is an open source project management tool, built using the Spring framework, and DWR (to handle AJAX). C6 has advanced user-input submission formats, and random values inside requests.

4.2 Experimental Setup

Experiments are performed on Linux-based computers with an Intel^{ő} Core™2 CPU at 3GHz and 3GB of RAM on a 100Mbps LAN. To implement the *D-ForenRIA* SR-Browsers, we used Selenium. *D-ForenRIA*'s SR-Proxy is implemented as a Java application. For each test application, we recorded the full HTTP traffic of user interactions with the application using *Fiddler*[13].

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.

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

$$n_e + \sum_{i=1}^{n_r} c(r_i) \tag{1}$$

where n_e is the number of actions in the user's session, and there are n_r resets (see Section 3.7) during reconstruction and the i^{th} reset, r_i, has cost of $c(r_i)$. The cost of reset r_i is determined by how much progress have been made during the reconstruction; If r_i, happens when the algorithm has detected m user-actions, the algorithm needs to execute m previously detected actions again to perform a reset (See Section 3.7), therefore $c(r_i) = m$.

We emphasize that the *cost* provides a more reliable measure of efficiency than the total *time* of the session reconstruction. It is due to the fact that the *time* depends on factors that are out of the control of the session reconstruction tool (such as the hardware configuration and the networks speed). On the other hand, the *cost* only depends on the decisions made by the session reconstruction algorithm.

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 the 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 3.3. To the best of our

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
C6	Tudu Lists	80	30	https://sourceforge.net/projects/tudu/

knowledge at the time of writing, no other published solution provides such a basic solution; thus there is no other solution that can reconstruct RIA sessions, even inefficiently.

The Min-Time: If our session reconstruction algorithm can find all user-browser interactions without trying incorrect actions its execution time becomes minimum. In this case, the algorithm does not need to do any *reset*. We 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 the cost and time of the reconstruction to measure the 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. 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 successful, full sessions reconstruction 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. In all cases, the reconstructions are complete and successful, meaning that the complete set of user actions are correctly recovered.

D-ForenRIA outperforms the basic solution in all cases. If we look at the number of events that must be executed to discover the next action (column "#Events/Action" in Table 2), on average across all experiments it takes *D-ForenRIA* the execution of 34 events to find the next single user-browser interaction while the basic solution needs 1720 events to find the same thing. Regarding the execution time, *D-ForenRIA* (even using a single browser) is orders of magnitudes faster than the basic solution. On average, over all experiments, *D-ForenRIA* needs 12.7 s to detect an action while the basic solution needs around 8 min to detect an action.

Number of Resets per Action: Figure 18 presents a breakdown of the number of resets needed to detect a single 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 C4 where 32% of the 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, 83% of the actions are found immediately at the current state based on the ordering done by the SR-Browser and SR-Proxy. On the other hand, for the basic method (Fig. 18b), 52% of the actions need at least 25 resets. This figure also shows that the basic solution tries more than 50 actions to find 32% of actions.

Performance of the Distributed Architecture (RQ2): Figure 19 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 browsers would speedup the process as long as the system required *resets*. Specifically, if the algorithm needs nr_i resets to find the i^{th} correct action, using up to nr_i browsers should decrease the execution time to find that action, while adding more browsers would not contribute any speedup. The results we obtained verified this argument. The best speedup happens in C3, C4 and C6 where we have the largest number of resets (See Fig. 18). 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 detects the correct action as the most promising one (Section 3.3)). Sometimes, adding more browsers is not beneficial; for example in C1 and C2, we observed no improvement in the execution time after adding more browsers (from 2 to 8). This is because in these application many actions are found immediately by *D-ForenRIA*. However, concurrent trying of actions is the key to scalability when the signature based ordering cannot find the correct action at states.

Table 2 Time and cost of sucessfull reconstruction using *D-ForenRIA*, the basic solution, and Min-Time

ID	D-ForenRIA			Basic solution			Min-Time
	#Events	#Events/Action	Time (H:m:s)	#Events	#Events/Action	Time (H:m:s)	Time (H:m:s)
C1	183	1	0:02:44	102933	686	09:51:26	00:02:21
C2	52	1	0:02:25	34505	690	04:31:57	00:02:06
C3	1325	30	0:04:22	308548	6856	19:28:48	00:01:12
C4	3506	140	0:19:47	21518	861	02:12:01	00:01:36
C5	319	10	0:02:29	14847	478	00:48:29	00:00:39
C6	631	21	0:11:24	22529	751	02:32:39	00:05:21

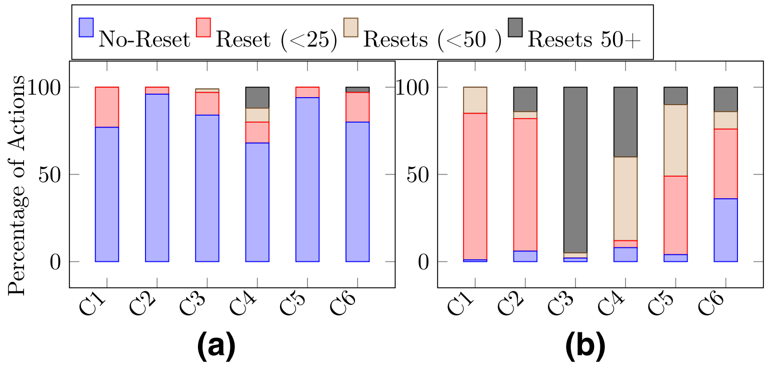


Fig. 18 Breakdown of the number of resets needed to identify a user-browser interaction in *D-ForenRIA* (**a**) and in the basic solution (**b**)

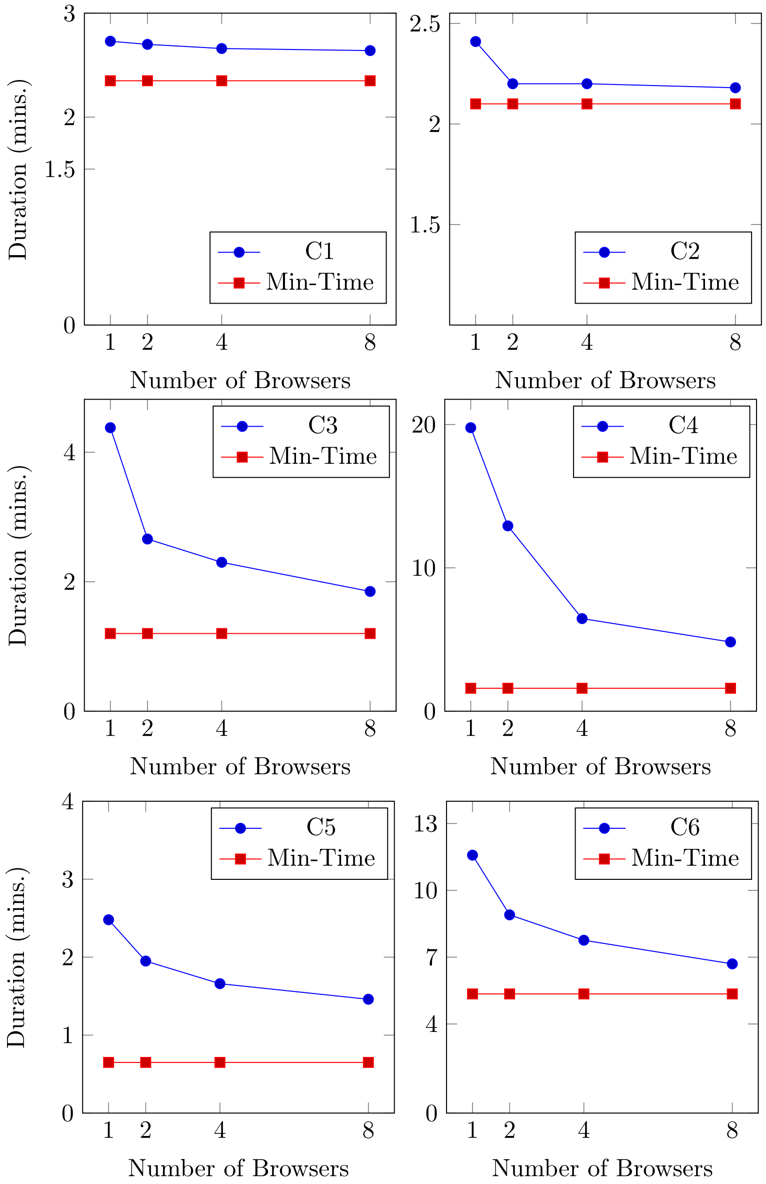


Fig. 19 Scalability of *D-ForenRIA* in different RIAs compared to the Min-Time

Efficiency of Candidate Actions Ordering Techniques (RQ3): As we discussed in Section 3.3, 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 looked at the number of elements, number of visible elements, number of elements with a handler, number of leaf elements, and also the 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.

In our experiments, ordering based on visibility helped us in finding correct actions sooner. We observed that in our test cases, all actions have been performed on visible elements on the page. On average, ordering based on visibility reduces the promising candidate actions by 18%. We also expected that the user-interactions happen with elements with an event-handler. The ordering based on event-handler was effective in all cases except C4. In all other RIAs, all the user-actions are performed on elements with an event handler; If we exclude C4, 76% 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, *D-ForenRIA* gives high priority to leaf elements of the DOM (Section 3.3). However, there is still a considerable ratio of leaf nodes, 55% on the DOMs. Giving higher priority to leaf nodes helped us to find correct actions in less time, since in C4 all the actions were performed on leaf nodes.

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; Although we could only apply the signature-based ordering on a rather small portion of the actions in each page (on average 14% of actions on each DOM, since most of the actions have not been executed and have not shown a signature), it could immediately detect the correct action on each DOM in 83% of cases (See Fig. 18).

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

ID	#Elements	Visible(%)	Handlers(%)	DOM leaves(%)	Signs. applied(%)
C1	79	81	18	55	4
C2	108	98	29	47	12
C3	648	64	19	77	8
C4	268	85	0.3	43	20
C5	44	97	23	50	23
C6	148	68	32	59	19
Average	243	82	20	55	14

HTTP-Log Storage Requirements (RQ4): One of the assumptions of the input user-log for *D-ForenRIA* is that it should contain both HTTP requests and responses. Since the input 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 we measured some features of HTTP logs in our test cases (Table 4).

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 is 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 to store the "Full" HTTP request-responses of each action[14]. The required size per action varies from as low as 1.12 KBs to the high of 11.74 KBs for C4 and the average is 3.68 KBs. We also considered pruning multimedia resources from the log (i.e,. images and videos). These resources affect the appearance of the website, and usually do not affect the ability of the website to respond to user-interactions. Therefore, pruning multimedia resources usually does not jeopardize the reconstruction process. With pruning, the average required space for a single action dropped by 14% and reached about 3.19 KBs.

4.4 Discussion:

Recording HTTP traffic: The HTTP requests exchanged between a browser and the server can be logged at different places in the network; they can be logged on the server, in the proxy-server or even on the client-side. However, recording on the client-side requires additional configuration/installation of recording software which is not desired. HTTP servers (such as Apache[15] or IIS[16]) can be configured to record the full traffic. To use *D-ForenRIA*, there is no need to change the Web application or to instrument any code on the client side[17].

In addition, the traffic can also be captured while it goes through the network using other tools such as proxies.

Table 4 Log size features for test cases

ID	Reqs./Action	Log Size/Action (KB)	Pruned Log Size/Action (KB)
C1	1.16	1.58	1.2
C2	1.36	1.41	0.41
C3	1.05	1.12	1.12
C4	6.56	11.47	9.96
C5	2.38	3.38	3.38
C6	2.66	3.16	3.12

Recording using proxies (or similar tools) is especially important in practice when a RIA sends requests to external servers since recording the traffic on external servers is inconvenient and usually not practical.

However, the recorded traffic using a proxy may be encrypted; the encrypted traffic may break our tool since the SR-Proxy cannot compare the generated requests with the log and verify the actions executed by SR-Browsers. In this case, the session reconstruction needs to take measures, such as a man-in-the-middle interception, to be able to read the unencrypted requests/responses.

SSL and recording HTTPS traffic: *D-ForenRIA* needs an access to the traffic in plain text format to perform the reconstruction. Nowadays, SSL/TLS [19] is widely used to encrypt the traffic between the client browser and server, making a network level direct access to the plain text of the traffic all but impossible. However, when the Web server receives an encrypted request, it must first decrypt it before processing it. Similarly, the response is first produced in plain text server-side before being encrypted and sent to the client. Therefore, HTTPS traffic is not actually a problem for session reconstruction if the logs from which the session is reconstructed are generated by the server hosting the application, which is in fact the normal situation: the reconstruction is carried out on behalf of the owner of the web server to investigate a security breach on that server. If access to the server logs are not possible, other methods are possible for HTTPS interception [20]; these methods are frequently used in industry to monitor an internal network, or by antivirus software for example, but they are more intrusive and require the installation of a certificate on the client. Nevertheless, such HTTPS interception system do provide perfectly adequate logging mechanism to allow session reconstruction on RIAs using HTTPS.

SSL-enabled sites do pose another issue for the session reconstruction process: the SR-Browsers communicate with the SR-Proxy as if the latter was the real server. Therefore, if the RIAs was using HTTPS, the reconstruction will use it as well, and the traffic should be encrypted based on the server's certificates which is not available to our tool. To solve this issue, *D-ForenRIA*'s SR-Proxy acts as man-in-the-middle [21]; we install our own certificate in SR-Browsers and then we just create and sign certificates with it.

User-input values encoded at client-side: We have assumed that the values used as sample data to detect user-input actions are going to be observed in the request generated after performing the action (Section 3.5). If the RIA applies some transformations on the input data before submitting it to the server, this can cause problems for the proposed technique. For example in C6, once a user selects a *true/false* value from a select element, the selected value is encoded as numerical values of *0/1*. To alleviate this problem, our technique can be improved in the following way.

For each user-input action, each user-input element is being tried with all possible values for that element (For example the form that contains a select with *true/false* values will be submitted twice, once with *true* selected and once with *false* as the selected value). This shows how the RIA maps different user input values to values inside the HTTP requests. However, this approach is only effective when the set of possible values for a user-input element is predefined (such as *select*, *checkbox* or *radio* input elements). In case of a free form element (such as a *text-box*), it is impossible to try all possible input values and find the mapping. Anecdotally, we have not seen many RIAs that encode textual user inputs. This is probably because RIAs usually run over HTTPS and thus do not need to worry about the data being intercepted. Therefore, what *D-ForenRIA* supports is based on a widely utilized practice in modern RIAs. It would be possible to enhance *D-ForenRIA*'s ability to handle non-standard value encodings by adding library-specific decoding code if a commonly used JavaScript library was used to encode user-input in a non standard way. This kind of enhancement is however beyond the scope of this research.

Variations in an action's signature: The signature-based ordering assumes that an action would generate the same set of requests at different states. When this is true, it is possible to predict the behavior of an action in the current state, based on the action's behavior in previous states. Although this assumption holds in majority of cases in our experimental results (Fig. 18), there are a few cases where this assumption is not satisfied. In our test applications, only C5 exhibits this behavior: the list of different items (such as products, photos,...) are presented in paginated catalogs and there are next and previous buttons to navigate between pages. The next (and the previous) button, generates different sets of requests in different states. Therefore, using the signature-based ordering is not effective in this case. Consequently, when *D-ForenRIA* observes this variation in the generated requests of an action, it disables using the signature information to order the action.

Importance of multi-browser support: Selenium [22] is a set of tools that enable a program to instantiate a Web browser and trigger different events of a Web page. *D-ForenRIA* uses Selenium to implement the SR-Browsers. One important feature of Selenium is that it supports the most popular browsers. This feature enables us to use different browsers (for example Chrome or Firefox) during reconstruction. It is important for *D-ForenRIA* to use the browser that the user used while visiting the website. For example, in our applications, C4 could only be reconstructed using Firefox since the traces are generated using Firefox, and the website generates different traffic based on the user's browser[18].

Hijacking JavaScript functions: To keep track of events on each state, *D-ForenRIA* hijacks several corresponding JavaScript methods. This includes overriding methods such as *setTimeout*, *setInterval*, etc. One may wonder whether this overriding interferes with the RIA if the RIA itself hijacks these methods. To mitigate this issues, *D-ForenRIA*'s hijacking mechanism is executed after possible hijacking codes executed, and it includes the RIA's code for the hijacked method. In other words, *D-ForenRIA* first executes the application's code for the JavaScript event (such as setTimeout) and then instruments its required code. This approach guarantees execution of the RIA's code before the code required by *D-ForenRIA*.

Non-Deterministic RIAs: Many sites (such as news websites, web-based mail clients etc.) are not deterministic, and the next state of the application when a request is sent will depend on some other evolving conditions. Since we assumed that the RIA under reconstruction is deterministic, one may argue that this assumption limits the applicability of the tool in the real world. For example, loading the home page of a news website each time probably results in different content. However, in the session reconstruction context, we replace the server with a proxy; the proxy always replies to a request using previously recorded traffic. Since the response is always the same (which corresponds to the response logged during the actual session) the application becomes deterministic for that action (e.g. in the case of a news website, loading the homepage from a prerecorded log always returns the same set of news). This type of *server-side non-determinism* is not a problem for the session reconstruction problem.

Generalization of our testing results: We discuss in the following whether one can assume that our results remain valid for user-session reconstruction in general, for any RIA. There are several issues that may limit such a generalization.

One issue is about supporting all technical features of RIAs; several features need to be added to *D-ForenRIA*, to make the tool applicable to more RIAs. Some of these features can be easily added to the tool; for example, *D-ForenRIA* currently does not support user actions that involve a right-click or scrolling a list. However, the current techniques in *D-ForenRIA* can be easily extended to handle these events. On the other hand, adding features such as handling the traffic generated by Web sockets/long polling, or coping with incomplete input trace that is recorded from the middle of a session, need more research and entirely new techniques.

A threat to the validity of our experiments is the generalization of the results to other test cases. To mitigate this issue, we used RIAs from different domains and test cases built using different JavaScript frameworks (Table 5). In addition, each of these applications is bringing new challenges for the session reconstruction tool. Table 6 presents the challenges we faced during reconstruction of the trace for each test case. We tackled these challenges one after the other starting from simpler cases such as C2, C3, C5 to more complex test cases C4 and C6. Although our session reconstruction approach is not an exhaustive one that handles every possible case, we have tried to solve a collection of difficult problems. However, there are more challenges to be addressed.

Another issue is regarding our input traces. We are using a single trace for each application which includes a limited number of user actions and is recorded by members of our research group. One may ask whether these traces are representative of the users of that application? We argue that the session reconstruction problem is similar to code testing, where people are trying to reach sufficient code coverage; each RIA consists of several different actions, but many of them trigger the same code. Here we are not trying to be representative because it is not meaningful in this case, what we are trying to do is to have a trace that covers the code of the application appropriately.

To measure the coverage of our traces, we measured the ratio of Javascript code executed during a session[19]. As Table 7 shows, the average code coverage is 82.7% with the minimum of 71.6% (for C5). In C2, all the possible actions have been performed during the session, and any other trace is just going to be a combination of these actions in a different order. In C1, C3 and C5, our traces do not include simple actions (such as actions that change the language of the RIA, or show a dialog). In C4 and C6 we have actions that encode user-input values, which are not supported by the current implementation of *D-ForenRIA*. Therefore, these actions are not included in our traces.

The characteristics of the RIA, can affect the performance of the signature based ordering. In the signature-based ordering, the assumption is that an action that is tried in a state will be present in other states of the RIA. If in a RIA performing an action generates an entirely new set of actions on the new state, the signature-based ordering will not be effective. However, based on the "partial updates" of the DOM in RIAs, usually performing an

Table 5 JavaScript frameworks of our test cases

Test case	C1	C2	C3	C4	C5	C6
JavaScript framework	jQuery/jQuery UI	Ajax	Ajax	GWT, Prototype	Ajax	DWR

Table 6 Challenges in our test cases

Challenge	Test case					
	C1	C2	C3	C4	C5	C6
Complex user-inputs	X	X		X		X
Changing values in Requests	X					X
Number of candidate actions			X	X		X
Bubbling				X		
Timers				X		X
Actions without HTTP traffic					X	X

action slightly changes the available actions on the page. Therefore, we believe that the signature-based ordering is effective in most RIAs.

5 Related works

Formerly, user-session reconstruction meant being able to find which pages a user had visited during a session from server logs. This task is often a preprocessing step for Web usage mining [23, 24]. In this paper, we assume that individual user-session have already been identified using one of these techniques [11, 25]. The previous session reconstruction approaches can be categorized as proactive and reactive methods [26].

Proactive methods record detailed information about actions during the actual session and usually do not deal with the recorded HTTP traffic. In proactive methods, the data about each user-browser action is collected during the actual session. Atterer [27], proposed to use a proxy which inspects HTTP responses from the server, and injects a specific user-tracking JavaScript code in the responses. Instead of using proxies, developers can also add user-tracking scripts to their code. The most popular example of these systems is Google Analytics [28]. The commercial products such as ClickTale [29] and ForeSee cxReplay [30] capture mouse and keyboard events in Web applications. The actions of the user can also be logged with browser add-ons. The Selenium IDE (implemented as a Firefox plug-in) [22] and iMacros for Chrome [31], or WaRR [32] record user actions and replay them later inside the browser. DynaRIA [33], ReAjax [34] also use an instrumented browser to capture all user interactions with an AJAX application. These tools provide several features to analyze the runtime behavior and structure of the application. In addition to using an instrumented browser, FireDetective [35] also instruments the server-side code of the application to capture a more accurate behavior of the application during a session. Rachna [36] reconstructs multi-tabbed user sessions using the recorded HTTP traffic, traces in lower layers of the network, and browser logs. Although a proactive session reconstruction approach can guarantee a complete reconstruction, they have deployment problems. These methods can also raise users' privacy concerns since they can track users' movements across the Web.

Table 7 Code coverage of our traces

Coverage	Test case					
	C1	C2	C3	C4	C5	C6
Trace Coverage (%)	89	100	75	88	71.6	73

In reactive methods, the input of the session reconstruction is the previously captured HTTP traffic of a session. The reactive session reconstruction for traditional Web applications is a well-studied problem [25]. In this paper, our focus was on reactive session reconstruction method for RIAs. To the best of our knowledge, few works [10, 37] have been done for session reconstruction in RIAs. Here we briefly mention some of the previous reactive session reconstruction techniques. None of the previous methods handle complex RIAs, and their focus is on Web 1.0 applications.

A graph-based method, RCI tries to reconstruct user-browser interactions [7]. RCI first builds the referral graph, and then prunes the graph by removing the automatically generated requests during rendering a page. Then nodes of the graph, which represent visited pages, are compared with DOM elements to find a triggering element for each request [7, 38]. ClickMiner [7] 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 [39]. In addition, although there is some level of support for JavaScript, ClickMiner is lacking the specific capabilities that are required to handle RIAs (e.g., handling user-inputs, client-side randomness, restoring the previous state, sequence check). Our previous work, ForenRIA [10], proposes a forensics tool to perform automated and complete reconstruction of a user session in RIAs. However, ForenRIA is less effective and scalable than *D-ForenRIA* since it is implemented as a single-threaded system where a single client (i.e., browser) is responsible for executing all the possible actions on a given page. ForenRIA also did not have the ability to support complex user input actions, timers, and actions that do not generate any traffic.

In reactive session reconstruction methods, caching can present a problem [7]. The reason is that caching prevents the registration of all requests by the server, and thus blurs the picture of user behavior [11]. Therefore, the reconstruction algorithm should attempt to infer cached requests. ClickMiner uses referral relationship graphs to detect missing pages which are not present in the recorded traffic. A related problem, *Path completion*, has been discussed in Web mining research [40, 41]. Path completion refers to extraction of page visits that are not recorded in an access log due to caching. All previously proposed path completion methods use relationships between pages to address this problem. For example, Spiliopoulou et. al. proposed a path completion method using the referer and the site-topology. Li. et. al. also use referer information to solve this problem [40]. However, in RIAs the Web application has very few pages and the referer information is usually missing; Therefore, previous path completion methods do not work with RIAs. As far as we know, our idea of using the action-graph (Section 3.8) is the first path completion technique for RIAs.

6 Conclusions

Session reconstruction is much more challenging in RIAs than in traditional Web applications. In RIAs, simply looking at the data flowing between the browser and the server does not provide the necessary information to reconstruct user-interactions. In this paper, we proposed *D-ForenRIA*, a distributed tool to recover user-browser interactions from a given HTTP trace in RIAs. The main contributions are:

- Providing a formal definition of the session reconstruction problem in the context of client/server applications
- A general solution for the session reconstruction problem that has been implemented in the context of RIAs in a tool called *D-ForenRIA*
- Addressing several challenges for session reconstruction in RIAs including the identification of candidate user-browser interactions, efficient ordering of candidate actions, distributed reconstruction, detection of complex user-input interactions and actions that do not generate any HTTP traffic
- Experiments on six different websites showing promising improvement of performance and scalability

However, there are still several 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. Moreover, the tool needs to be improved to handle cases where the log is incomplete, and when it is not recorded from the start of the session.

Exploring how *D-ForenRIA* can be used in a Web usage mining tool, or for the regression testing of RIAs by replaying the extracted user-actions, are other application domains for future work.

Endnotes

[1] The study performed in August 2016. Results are available at: http://ssrg.eecs.uottawa.ca/sr/alexaajax.html

[2] D-ForenRIA: Distributed Forensic session reconstruction for RIAs

[3] The user-interface was implemented by *Muhammad Faheem*.

[4] This does not prevent *server-side* non-determinism, where the response to a given request from a given client-state might be different at different time. This more common scenario is not a problem for session reconstruction since the responses are recorded and will this be deterministic in the replay.

[5] Here $|R_s|$ denotes the number of requests/responses in R_s, and *begin(n,b)* function returns the sequence of the first n elements of sequence b).

[6] Here $R - R_s$ refers the trace that contains elements in R minus the sequence of requests/responses that has been matched to R_s.

[7] https://www.websocket.org/aboutwebsocket.html

[8] https://jquery.com/

[9] https://mootools.net/

[10] For example, *on("event")* in jQuery or the *addEvent* method of Mootools.

[11] In this case, D-ForenRIA will hijack the hijacked code, and will call that hijacked code after having executed its own code.

[12] http://ssrg.site.uottawa.ca/sr/demo.html

[13] http://www.telerik.com/fiddler

[14] The compression was done using 7z algorithm

[15] https://httpd.apache.org/docs/2.4/mod/mod_dumpio.html

[16] https://msdn.microsoft.com/en-us/library/ms227673.aspx

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

[18] In our previous paper [10] we were using PhantomJS (PhantomJS.org) instead of Selenium. PhantomJS is much faster that Selenium and yields better reconstruction time, but only offers limited multi-browser support.

[19] The code coverage was measured using Chrome 59.0 devtools and we excluded the JavaScript code which can never be executed at run-time (dead code).

Acknowledgments
This work is supported by Center for Advanced Studies, IBM® Canada Lab and the Natural Sciences and Engineering Research Council of Canada (NSERC). A special thank to Muhammad Faheem and Sara Baghbanzadeh.

Authors' contributions

SH was the main researcher for this work during his Ph.D. studies. He has participated in the design of the algorithms, the implementation of the

D-ForenRIA, and in conducting the experiments. GVJ (the supervisor of SH), GB, and IVO have made substantial contributions to the design of the algorithms, the technical discussions and on the drafts of the manuscript. RC shared his industrial insight and provided constant feedback on the suitability of D-ForenRIA for real-world use. All authors have read and approved the final version of the manuscript.

Competing interests

The authors declare that they have no competing interests.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Author details

[1]University of Ottawa, 800 King Edward Avenue, K1N 6N5 Ottawa, Canada. [2]IBM Security, Rogers St, MA 02140 Cambridge, USA. [3]IBM Centre for Advanced Studies, Ottawa, Canada.

Received: 29 October 2017 Accepted: 15 March 2018
Published online: 02 May 2018

References

1. Garrett JJ, et al. Ajax: A new approach to web applications. 2005. Available at: http://adaptivepath.org/ideas/ajax-new-approach-web-applications/. Accessed Mar 2018.
2. Marini J. Document Object Model. New York: McGraw-Hill; 2002.
3. Fraternali P, Rossi G, Sánchez-Figueroa F, Vol. 14. Rich internet applications. New York: IEEE Internet Computing; 2010. pp. 9–12.
4. Mikowski MS, Powell JC. Single page web applications. 2013.
5. Nederlof A, Mesbah A, Deursen Av. Software engineering for the web: the state of the practice. In: Companion Proceedings of the 36th International Conference on Software Engineering. New York: ACM; 2014.
6. Mesbah A, Van Deursen A, Lenselink S. Crawling ajax-based web applications through dynamic analysis of user interface state changes. ACM Trans Web (TWEB). 2012;6(1):3.
7. Neasbitt C, Perdisci R, Li K, Nelms T. Clickminer: Towards forensic reconstruction of user-browser interactions from network traces. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM; 2014. p. 1244–1255.
8. Mobasher B. Data mining for web personalization. In: The Adaptive Web. Berlin, Heidelberg: Springer; 2007. p. 90–135.
9. Amalfitano D, Fasolino AR, Tramontana P. Rich internet application testing using execution trace data. In: Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference On. IEEE; 2010. p. 274–83.
10. Baghbanzadeh S, Hooshmand S, Bochmann G, Jourdan GV, Mirtaheri S, Faheem M, Onut IV. Reconstructing interactions with rich internet applications from http traces. In: IFIP International Conference on Digital Forensics. Springer; 2016. p. 147–64.
11. Spiliopoulou M, Mobasher B, Berendt B, Nakagawa M. A framework for the evaluation of session reconstruction heuristics in web-usage analysis. INFORMS J Comput. 2003;15(2):171–90.
12. addEventListener API, Mozilla Developers Network. https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener. Accessed 22 Mar 2017.
13. Chess B, O'Neil YT, West J. 2007. Available at: https://www.infopoint-security.de/open_downloads/alt/JavaScript_Hijacking.pdf. Accessed Mar 2018.
14. Bubbling and capturing in JavaScript. http://javascript.info/tutorial/bubbling-and-capturing. Accessed 22 Mar 2017.
15. W3C HTML5 recommendations. https://www.w3.org/TR/2014/REC-html5-20141028/forms.html#forms. Accessed 22 Mar 2017.
16. Oh J, Moon SM. Snapshot-based loading-time acceleration for web applications. In: Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization. IEEE Computer Society; 2015. p. 179–89.
17. Oh J, Kwon J-w, Park H, Moon SM. Migration of web applications with seamless execution. ACM SIGPLAN Not. 2015;50(7):173–85.
18. Caching in HTTP. https://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html. Accessed 22 Mar 2017.
19. Freier A, Karlton P, Kocher P. The secure sockets layer (ssl) protocol version 3.0. 2011. Available at: https://tools.ietf.org/html/rfc6101. Accessed Jan 2018.
20. Carnavalet XCde, Mannan M. Killed by proxy: Analyzing client-end TLS interception software. In: Network and Distributed System Security Symposium. Internet Society; 2016.
21. Callegati F, Cerroni W, Ramilli M. Man-in-the-middle attack to the https protocol. IEEE Secur Priv. 2009;7(1):78–81.
22. Selenium: Web browser automation. http://www.seleniumhq.org/. Accessed 22 Mar 2017.
23. Srivastava J, Cooley R, Deshpande M, Tan P-N. Web usage mining: Discovery and applications of usage patterns from web data. SIGKDD Explorations. 2000;1(2).
24. Dell RF, Román PE, Velásquez JD. Web user session reconstruction with back button browsing. In: International Conference on Knowledge-Based and Intelligent Information and Engineering Systems. Berlin: Springer; 2009. p. 326–32.
25. Cooley R, Mobasher B, Srivastava J. Data preparation for mining world wide web browsing patterns. Knowl Inf Syst. 1999;1(1):5–32.
26. Dohare MPS, Arya P, Bajpai A. Novel web usage mining for web mining techniques. Int J Emerg Technol Adv Eng. 2012;2(1):253–62.
27. Atterer R. Logging usage of ajax applications with the "usaprox" http proxy. In: Proceedings of the WWW 2006 Workshop on Logging Traces of Web Activity: The Mechanics of Data Collection. ACM; 2006.
28. Clifton B. Advanced Web Metrics with Google Analytics. Indianapolis: John Wiley & Sons; 2012.
29. Clicktale: Light up the digital world. http://www.clicktale.com/. Accessed 22 Mar 2017.
30. Forsee: Web and mobile replay. http://www.foresee.com/products/web-mobile-replay/. Accessed 22 Mar 2017.
31. iMacros for Chrome. https://imacros.net/browser/cr/welcome/. Accessed 22 Mar 2017.
32. Andrica S, Candea G. Warr: A tool for high-fidelity web application record and replay. In: Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference On. Los Alamitos: IEEE; 2011. p. 403–10.
33. Amalfitano D, Fasolino AR, Polcaro A, Tramontana P. The dynaria tool for the comprehension of ajax web applications by dynamic analysis. Innov Syst Softw Eng. 2014;10(1):41–57.
34. Marchetto A, Tonella P, Ricca F. Reajax: a reverse engineering tool for ajax web applications. IET Softw. 2012;6(1):33–49.
35. Zaidman A, Matthijssen N, Storey MA, Van Deursen A. Understanding ajax applications by connecting client and server-side execution traces. Empir Softw Eng. 2013;18(2):181–218.
36. Raghavan S, Raghavan S. Reconstructing tabbed browser sessions using metadata associations. In: IFIP International Conference on Digital Forensics. Switzerland: Springer; 2016. p. 165–88.
37. Hooshmand S, Mahmud A, Bochmann GV, Faheem M, Jourdan GV, Couturier R, Onut IV. D-forenria: Distributed reconstruction of user-interactions for rich internet applications. In: Proceedings of the 25th International Conference Companion on World Wide Web. International World Wide Web Conferences Steering Committee. New York: ACM; 2016. p. 211–4.
38. Xie G, Iliofotou M, Karagiannis T, Faloutsos M, Jin Y. Resurf: Reconstructing web-surfing activity from network traffic. In: IFIP Networking Conference, 2013. New York: IEEE; 2013. p. 1–9.
39. Mesbah A. Software analysis for the web: Achievements and prospects. In: Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference On, vol. 5. Los Alamitos: IEEE; 2016. p. 91–103.
40. Li Y, Feng B, Mao Q. Research on path completion technique in web usage mining. In: Computer Science and Computational Technology, 2008. ISCSCT'08. International Symposium On, vol. 1. Los Alamitos: IEEE; 2008. p. 554–9.
41. Munk M, Kapusta J, Švec P. Data preprocessing evaluation for web log mining: reconstruction of activities of a web visitor. Procedia Comput Sci. 2010;1(1):2273–280.
42. Typo3 CMS. http://cms-next.demo.typo3.org/typo3/. Accessed 22 Mar 2017.