Efficient Reconstruction of User Sessions from HTTP traces for Rich Internet Applications

by

Salman Hooshmand

Thesis submitted in partial fulfillment of the requirements for the
Doctorate in Philosophy Computer Science degree

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

The generated HTTP traffic of users' interactions with a Web application can be logged for further analysis. In this thesis, we present the "Session Reconstruction" problem that 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.

New Web technologies such as AJAX and DOM manipulation have provided more responsive and smoother Web applications, sometimes called "Rich Internet Applications"(RIAs). Despite the benefits of RIAs, the previous session reconstruction methods for traditional Web applications are not effective anymore. Recovering information from a log in RIAs is significantly more challenging as compared with classical Web applications, because the HTTP traffic contains often only application data and no obvious clues about what the user did to trigger that traffic.

This thesis studies applying different techniques for efficient reconstruction of RIA sessions. We define the problem in the context of the client/server applications, and propose a solution for it. We present different algorithms to make the session reconstruction possible in practice: learning mechanisms to guide the session reconstruction process efficiently, techniques for recovering user-inputs and handling client-side randomness, and also algorithms for detections of actions that do not generate any HTTP traffic. In addition, to further reduce the session reconstruction time, we propose a distributed architecture to concurrently reconstruct a RIA session over several nodes.

To measure the effectiveness of our proposed algorithms, a prototype called D-ForenRIA is implemented. The prototype is made of a proxy and a set of browsers. Browsers are responsible for trying candidate actions on each state, and the proxy, which contains the observed HTTP trace, is responsible for responding to browsers' requests and validating attempted actions on each state. We have used this tool to measure the effectiveness of the proposed techniques during session reconstruction process. The results of our evaluation on several RIAs show that the proposed solution can efficiently reconstruct use-sessions in practice.

Acknowledgements

First of all, I would like to extend my sincere gratitude to Professor Guy-Vincent Jourdan for giving me the opportunity to work under his supervision. His invaluable guidance and extensive knowledge of the subject helped me develop a deep understanding of the subject.

I wish to thank Professor Gregor V. Bochmann and Dr. Viorel Iosif Onut. Their support and feedback throughout the research, and their invaluable technical and non-technical advice, are really appreciated by me.

I would like to thank the members of my defense committee: Professor Babak Esfandiari, Professor Ettore Merlo, Professor Liam Peyton and Professor Nejib Zaguia, for providing me with valuable feedback on this thesis.

Many thanks to the (current and former) members of the Software Security Research Group (SSRG) at University of Ottawa: Seyed Mir Taheri, Mustafa Emre Dincturk, Di Zou, Ali Moosavi, Sara Baghbanzade, Alireza Farid Amin, Khaled Ben Hafaiedh, Akib Mahmud and Muhammad Faheem for accompanying me on this journey.

I would like to acknowledge that this work was supported financially by the IBM Center for Advanced Studies (CAS) and the Natural Sciences and Engineering Research Council of Canada (NSERC).

I am very grateful to my parents, my sister and my brother, and my parents-in-law, for their unconditional support and kindness.

Last, but certainly not least, I wish to thank my best friend and wife, Atefeh Shamsian for her endless kindness, patience, and confidence in me.

Table of Contents

List of Tables

List of Figures

List of Acronyms

AJAX Asynchronous JavaScript and XML

API Application Programming Interface

CSS Cascading Style Sheets

DOM Document Object Model

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

JSON JavaScript Object Notation

RIA Rich Internet Application

SR-Browser Session Reconstruction Browser

SR-Proxy Session Reconstruction Proxy

URI Uniform Resource Identifier

URL Uniform Resource Locator

XML eXtensible Markup Language

Chapter 1

Introduction

Over the last few years, an increasing number of application developers have opted for a Web-based solution. This shift has been possible due to the enhanced support of client-side technologies, mainly scripting languages such as JavaScript [33] and asynchronous JavaScript and XML (AJAX [36]). Rich Internet Applications (RIAs) [34] offer a new way to design web applications which are highly interactive, and have user-interfaces similar to desktop applications. RIAs utilize more flexible communication patterns between the client and the server, and some of the computational load is transferred to the client-side browsers. These enhancements help RIAs to be more interactive and user-friendly.

Despite the benefits of RIAs, this shift in Web development technologies has created a whole set of new challenges; many previous methods for analyzing traditional Web applications are not useful for RIAs anymore [51]. 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 the HTTP traffic of users' sessions. In this thesis, we focus on the session reconstruction problem, that is recovering user-browser interactions from the session's HTTP traffic. In this chapter, we first overview the evolution of the Web (Section 1.1), from traditional to Rich Internet Applications (Sections 1.1.1, 1.1.2), and talk about the motivations for the session reconstruction (Section 1.2). We then define the session reconstruction problem (Section 1.3) and list challenges of session reconstruction for RIAs (Section 1.4). We finally conclude the chapter with the contributions of the thesis (Section 1.7) and the organization of the thesis.

1.1 Evolution of the Web

1.1.1 Traditional Web Applications

The Web provides an easy access to Web documents for the users. Web applications follow a client-server architecture where the user's browser (the client) asks for the Web pages on the server (usually over Internet using HTTP). It is possible to classify traditional Web applications into two major categories: static and dynamic applications. In static Web applications, a fixed page is stored on the server, and when the user requests the page a copy of the same page is sent to the user. On the other hand, in dynamic Web applications, the contents of the page is not fixed; when a user sends a request for the page, the server-side application renders a new page (possibly by combining the data from databases) and sends the page to the user. In traditional Web applications, the client's only responsibility is to interact with the user, ask for a page from the server, and then render the response to the user. In addition, when the client receives a new page from the server, it replaces the previous page with this newly received page. In other words, traditional Web applications are composed of a set of pages, each with its URL and each user interaction with the website transfers the browser from the current page to a new page with a new URL. In the following we discuss the main technologies in traditional Web applications:

- **HTML** is the language which describes the structure of a web page. Different kinds of page elements are represented by their tag (i.e. the <img> tag is used for images, <a> tag for links between pages). Each tag can have some attributes that describe each element. For example, the *id* attribute specifies a unique id for an element. Some HTML elements can contain other HTML elements. For example, an HTML table is composed of several rows and columns. In this case, tags are nested within each other. For example, <tr> tags are used for table rows and are nested inside the <table> tag.

- **URL** is a string of characters used to reference a resource, such as a web page, uniquely on the web space. A URL specifies the name and the location of the identified resource and the protocol for accessing it [12].

- **HTTP** is a protocol to exchange data between a client and a server [32]. HTTP

is based on generating requests on the client-side, and generating responses on the server-side. Figures 1.1 and 1.2 present a sample HTTP request and its response. Each HTTP request has a body, which contains a message, and also several headers to describe the request (such as user-agent). The first line lists the request method, usually GET/POST, and also the identifier of the requested page (including the directory and name of the page). If a user wants to submit some data to the server, the data is appended to the URL of the page, or put inside the body of the request. For example, request in Figure 1.1, is a GET request for the /index.html file, that is located at www.example.com.

```
GET /index.html HTTP/1.1
Host: www.example.com
```

Figure 1.1: Sample HTTP Request

When the server receives a request, it generates a response based on the URL, headers, and body of the request. Each HTTP response is composed of a response line, a set of headers, and a body. In the first line, there is a status code which represents the status of the response (based upon predetermined standards - i.e. 200 for successful requests or 404 for the pages that do not exist on the server). Figure 1.2 presents the generated response to the request in Figure 1.1. It shows that the request has been successful (code 200) and it contains the HTML code of the requested page.

HTTP logs: The sequence of exchanged messages between the client and server is typically partially or entirely logged by the Web server hosting the application. The traffic can also easily be captured while it passes through the network. We call the captured traffic generated by a user during a session with the Web application the *User-Log*.

Browser/Server Interactions in Traditional Web applications : When a user enters the address of a Web page in her browser (or clicks on a link to a page), the browser creates a GET request for the URL of the page. The server processes the request and, if the page exists, returns the HTML of the page to the user's browser. The browser parses the HTML and renders the page eventually. If the HTML of the page contains references to other resources (such is the logo.jpg image in Figure 1.2), new requests are generated to the server.

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Content-Type: text/html; charset=UTF-8
Content-Encoding: UTF-8
Content-Length: 138
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Connection: close

<html>
<head>
  <title>An Example Page</title>
</head>
<body>
  Hello World, this is a very simple HTML document.
  <img src="logo.jpg">
</body>
</html>
```

Figure 1.2: Sample HTTP Response

1.1.2 Rich Internet Applications

In traditional Web applications the computation load is on the server and each user interaction causes a round-trip to the server to get a whole new page. A traditional Web application is not interactive, since the user needs to wait to receive the requested page, and the user cannot interact with the application meanwhile. The shift from static web applications to the RIA design pattern has provided the ability to shift some of the computational load from the server to the clients. This is achieved usually through a scripting language such as JavaScript that is executed in browsers.

Initially, the JavaScript runtime's functionality was limited. The code was responsible for implementing some part of the application's logic such as validating the user-input data before submission to the server. With the advent of AJAX, JavaScript plays a more important role in the client-side computation and developing interactive Web applications. With AJAX, JavaScript code can communicate with the server for a request, receive the resource, and update the DOM partially. AJAX requests are asynchronous meaning that

while the code waits for a response, the user can continue its interaction with the application. In addition, the changes in the page are partial and there is no need to receive a complete HTML page from the server. Figure 1.3 compares the communication model of traditional Web applications, and the Rich Internet Application model.

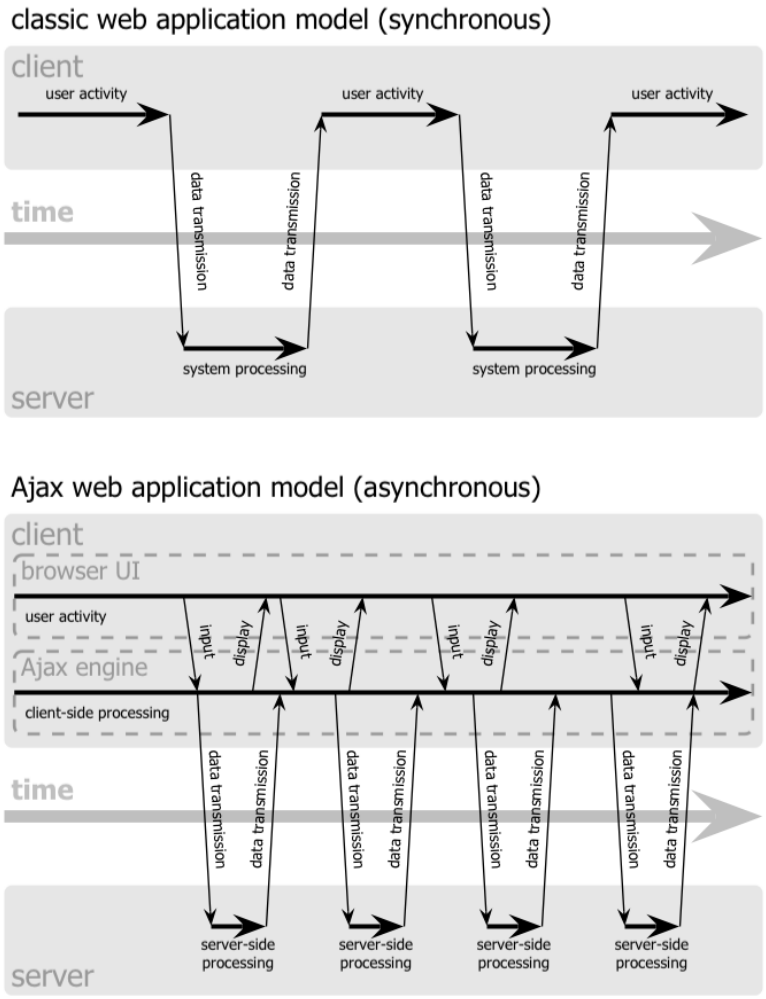


Figure 1.3: Client/Server communication model in Traditional web applications (Top), and RIAs (Bottom) [36]

AJAX is based on several technologies such as XML, JSON and DOM.

- **XML**: is a markup language used to describe data based on tags. The set of tags are not standard and tags can be defined based on the user's needs. XML is usually used to exchange data in a textual format. The XML file contains the data, as well as some tags to describe the meaning of each data.

- **JSON**: is one of the most common data formats used in AJAX. There are built-in JavaScript functions to generate a JSON representation of a JavaScript object and also to parse JSON data to a JavaScript object. Each JSON data can also be converted to a string. Both JSON and XML are platform independent formats and are usually used to provide interoperability between programs written in different languages and platforms.

- **DOM**: provides a tree structure of the page, and also an API to manipulate this structure. The browser, creates the DOM by parsing the HTML of the page. To do so, it creates a node for each HTML element of the page. When some JavaScript code changes the DOM, the browser updates the corresponding HTML element, and renders the updated parts of the page.

The Role of the Browser in RIAs: The browser is the client in a Web application. The client is responsible for interacting with the user, generating requests and rendering results from the server. Browsers are very complex software [40, 17] and should support all the details of several protocols such as HTTP, and languages such as HTML and CSS. Browsers may extend their ability to interpret some of the received resources from the server using extra software (called plugins - examples include Adobe Flash and Java Applets). However, it is the responsibility of the user to install these plugins on the browser.

In recent years there have been some efforts to reduce the need for plugins. To this end, browsers are equipped with built-in abilities to support features previously provided by plugins; for example, HTML5 provides some multimedia features which were once handled by Adobe Flash. Technologies such as HTML5 are supported by modern browsers and do not require the user to install any additional software. Rich Internet Applications are supported by all modern browsers since a JavaScript engine is an essential part of all browsers.

1.2 Motivations for Session Reconstruction

Each session refers to a single visit of a user to a website. During each session, several HTTP messages are exchanged between the browser and servers. Session reconstruction

refers to the extraction of user interactions which have generated the given HTTP traffic. Our goal is to automatically extract details about each action including elements involved, user input provided, and information related to the Web applications client-side state after executing each action (such as elements clicked, values entered in forms and screen-shots of visited pages). There are various applications of session reconstruction such as *forensics analysis*, *Web usage mining*, and *software testing*.

- **Forensics Analysis**: As the Internet usage increases, we also face a larger number of cyber attacks. The detailed forensic analysis of these incidents can be challenging: The forensic analysts must often manually analyze a large volume of incident data to discover the root of a security incident.

  Session reconstruction can be used as a replay tool for forensic analysis after detection of an intrusion. For example, when the owner of a Web application learns that a hacker has found and exploited a vulnerability a few months back, the administrator can find out how the intrusion happened using the HTTP-logs of past user-sessions.

- **Web Usage Mining**: Session reconstruction can be used to understand how users surf a website. This knowledge can be utilized by ISPs, network administrators and researchers for different purposes such as finding vulnerabilities in the network and redesigning the website based on usage analysis.

- **Software Testing**: Another application area is testing Web applications. Session reconstruction can help with debugging, performance evaluation, and even usability analysis of the Web application's user interface [52].

1.3 Session Reconstruction Problem

Figure 1.4 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 (Figure 1.4 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 (Figure 1.4 part b).

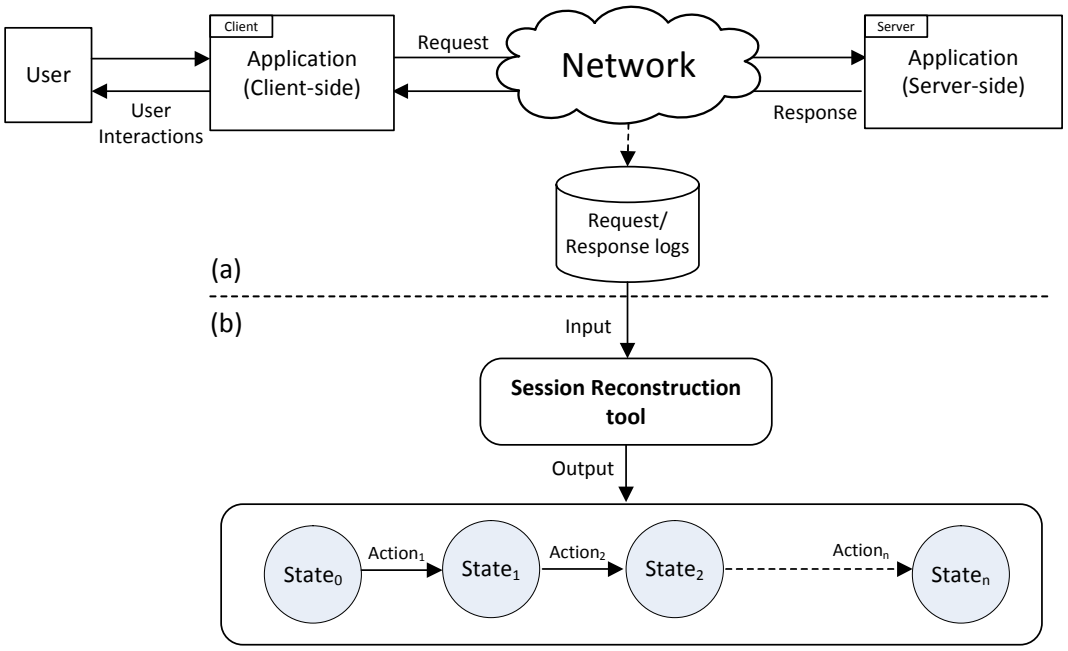


Figure 1.4: (a) client/server architecture , (b) Session reconstruction tool (input and output)

If we assume that there have been n user-interactions during a session, the trace of the session can be presented as $< rs_1, rs_2, ..., rs_n >$, 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 $< rs_1', rs_2'..., rs_n' >$ 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 requests 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 as a state machine. A state machine has a number of states, set of inputs, and set of outputs. Each input transfers the

state machine to a state and generates an output. In our model, the inputs of the state machine are user actions and there are outputs at two interfaces. One output interface is between the user and the client; after performing an action, the user can observe the new user-interface of the client-side of the application. The other output interface is between the client and the server; in this interface, one can observe the sequence of generated requests/responses exchanged between the client and the server after performing each user interaction.

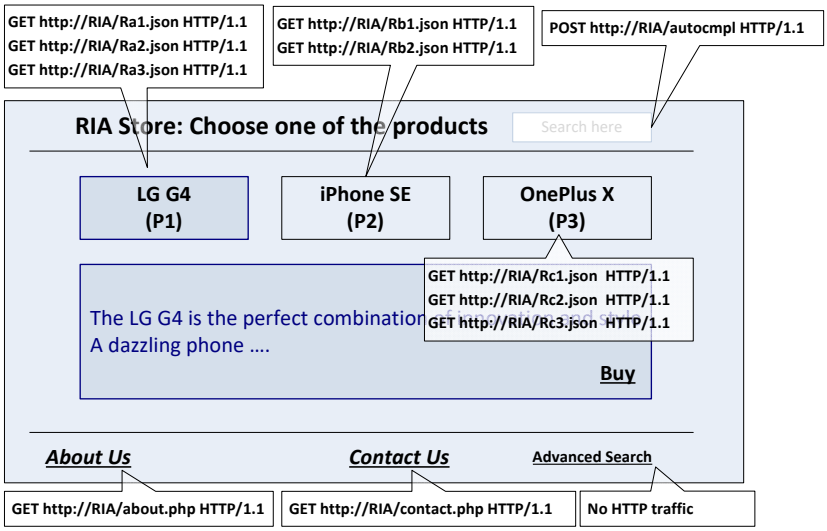
In the case of RIAs, the client is a Web browser which communicates using HTTP with a server. For each state, the session reconstruction tool extracts the DOM of the page, and produces a screenshot of the page as seen by the user. The tool also finds the XPath of the elements clicked, and the values that the user submits during the session.

Consider the example of Figure 1.5. It shows a very simple page 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, $< Request/Response_{Rb1}$, $Request/Response_{Rb2}, Request/Response_{Ra1}, Request/Response_{Ra2}, Request/Response_{Ra3}$ $, Request/Response_{Rc1}, Request/Response_{Rc2}, Request/Response_{Rc3} >$, 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.

1.4 Challenges of Session Reconstruction for RIAs

It is relatively straightforward to reconstruct user-interactions from the log in a "traditional" Web-application since the data sent by the server is more or less what is displayed by the browser. There are some existing tools that could help the administrator for this (e.g. [58]). But if the Web application is a modern RIA, manual reconstruction would be extremely difficult and time-consuming, and up to this point, there was no tool that could be used to help. RIAs have a number of properties which make the session reconstruction difficult for these applications:

- Candidate user-actions: RIAs are event-based applications; it means that any HTML element and not only links can respond to user actions. Once an event is triggered, the browser executes the corresponding event-handler code. These event-handlers can



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
...	...

...

(b)

Figure 1.5: (a) A simple page and generated requests after clicking on each element. (b) Portion of the user's log

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.

- Finding the source of a given request: At each state, the session reconstruction algorithm needs to answer this main question: What action produces the next expected HTTP traffic. This question is easy to answer in traditional web applications and challenging to answer in RIAs. In traditional Web applications, each user-browser

interaction usually redirects the web application to a new page. This redirection usually happens by clicking on a link (*href*) which requests the page from the server. So when there is a request for page p in the user-log, the algorithm simply needs to find an *href* on the current page which links to page p.

The situation is very different for RIAs; The algorithm 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 on the browser, and the algorithm needs to actually "execute" an action via a browser to find which requests an action would generate. In addition, The response typically only contains a small amount of data which is used by the receiving script to partially update the current DOM (see Figure 1.6 for an example[1]). Thus, many of these request/response pairs are part of a series of interactions and cannot be analyzed in isolation. It is thus more difficult to figure out precisely what user-action has generated a given request.

Therefore, extraction of candidate user actions at each state, and finding the source of a given HTTP request is more challenging in RIAs than traditional Web applications. 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). However, the stream of requests/responses can always be logged and used for session reconstruction. We also need to be able to do the reconstruction off-line with no access to the original server to ensure that the tool remains effective even when the server is not available (For example, because of an attack or a bug in the application).

1.5 Working Assumptions

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

[1] Adapted from TYPO3 RIA: `http://cms-next.demo.typo3.org/typo3/`

```
[
{
"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 icon-state-default icon-apps-
    pagetree-page-backend-users\" data-identifier=\"apps-pagetree-page-backend-users
    \">\n\t</span>",
"title": "Backend User Section",
"tooltip": "Backend User Section"
}
],
"debug": ""
}
]
```


Figure 1.6: The body of a typical HTTP response in RIA

- **Determinism**: It is assumed that the application is deterministic from the viewpoint of the user. It means that given a client-state and an input action, the client-state reached should all be equivalent. If the application is not deterministic, 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 [72]) 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.

- **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 in the generated requests are not encoded; 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 detect the set of acceptable values for a user-input action. However, the tool can still choose the correct value from this set of possible values; otherwise, the tool needs to consider a virtually unlimited number of possible values for each user-input action. When this assumption holds, client-side validation of the application is not a barrier for the tool to extract user-input values.

1.6 Research Method and Evaluation

In this thesis, a number of research methods that are common in the field of software engineering have been used. These methods and steps are briefly introduced as follows:

- Formulating the Problem: In Chapter 1, we formulated the session reconstruction problem. We defined the problem and discussed the challenges of the problem and working assumptions of our solution.

- Literature Review: We then performed a literature review in Chapter 2; in a literature review, the goal is to thoroughly search relevant studies and to establish a background knowledge in any research [44]. Based on the literature review, we identified different approaches for session reconstruction, the challenges and also the requirements for a practical session reconstruction tool.

- Proposing techniques and Tools: In our research we emphasized on proposing new algorithms and tools to support the techniques for the session reconstruction problem. In Chapter 3, we presented a general session reconstruction algorithm, and in Chapter 4 we explained how we implemented this algorithm for RIAs. The thesis resulted in the development of a session reconstruction tool called *D-ForenRIA*. Tools

and techniques were developed iteratively to address gaps in existing practice and tool support.

- Experiments: To investigate the performance and applicability of our techniques, we used descriptive RIAs [79], and performed an extensive empirical evaluation [76, 46]. We have tried to use a set of representative RIAs for our experiments; to this end, we used six RIAs from different domains built using different JavaScript frameworks. In addition, we selected RIAs that cover challenges identified in Chapter 1. To select the test user sessions, we used traces that cover the code of each RIA application appropriately (Chapter 6).

  Since it is not easy to generalize the results of a case study [46], Chapter 6 also includes a critical discussion on the findings and generalizations of the experimental results.

1.7 Contributions and Organization of the Thesis

Our research was carried out in close collaboration with the IBM QRadar Incident Forensics team. Our goal was to propose a general and fully automated method which can infer user-interactions from HTTP logs of any complex RIAs.
In this thesis, we have made the following contributions:

- A formal definition of the session reconstruction problem: This thesis introduces the session reconstruction problem in an abstract model of client/server applications. The definition is independent of any specific technology, and can potentially be applied to different client/server applications such as mobile applications or Web services.

- An efficient algorithm to reconstruct sessions of client/server applications: To reconstruct a session, an algorithm needs to consider possible user interactions at each state and try them one by one until it finds the correct action. In this thesis we propose a solution which is made of a proxy (called the SR-Proxy) and a set of browsers (called SR-Browsers). Browsers are responsible for trying candidate actions on each state, and the proxy, which contains the observed trace, is responsible for responding to browsers' requests and validating attempted actions on each state. The algorithm

uses several techniques to efficiently recover user-interactions from the given trace. The most important techniques include the following:

  - Efficient ordering of candidate user interactions: This thesis proposes two techniques (*signature-based* ordering and *similarity-based* ordering) to try the most promising actions first. These techniques are learning mechanisms which are based on the history of generated requests during event executions.
  - Distributed architectures for session reconstruction: A distributed architecture was introduced to try different candidate actions of each state in parallel. In this architecture a single node (the SR-Proxy) calculates the next candidate actions to be tried, and dispatches them to the SR-Browsers.
  - Detection of user-inputs: A technique was introduced to detect actions that include user-inputs. This technique can detect virtually any user-input action regardless of how the input data are formatted inside requests.

- Implementation of a session reconstruction tool for RIAs: The proposed session reconstruction algorithm has been implemented in the context of RIAs, in a tool called *D-ForenRIA*[2], a tool to reconstruct user-browser interactions with RIAs using HTTP traffic as the input. *D-ForenRIA* reconstructs screenshots of the session, recovers user inputs and all the actions taken by the user during that session. The tool uses a distributed architecture and automatically reconstructs the user-interactions in RIAs.

- Experimental evaluation of the proposed system: We empirically evaluate the efficiency of our approach on six RIAs. The results show that our method can efficiently reconstruct user sessions from previously recorded HTTP traffic.

1.7.1 Publications

We have published two papers out of this research:

- Hooshmand, Salman, Faheem, Muhammad, Bochmann, Gregor V, Jourdan, Guy-Vincent and Couturier, Russ and Onut, Iosif-Viorel. *D-ForenRIA: A Distributed*

[2]D-ForenRIA: Distributed system for Forensics reconstruction of Rich Internet Applications

Tool to Reconstruct User Sessions for Rich Internet Applications, in Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering (CASCON 2016), pages 64–74, 2016.

- Baghbanzadeh ,S., Hooshmand ,S., Bochmann , G. V., Jourdan G.-V, Mirtaheri, S., Faheem , M., and Onut, V.
  ForenRIA: The Reconstruction of User-Interactions from HTTP Traces for Rich Internet Applications , in Proceedings of the Twelfth Annual IFIP WG 11.9 International Conference on Digital Forensics, pages 147–164, 2016.

We have also demonstrated our session reconstruction tool in the demo track of World Wide Web 2016 conference:

- Hooshmand, Salman and Mahmud, Akib and Bochmann, Gregor V and Faheem, Muhammad and Jourdan, Guy-Vincent and Couturier, Russ and Onut, Iosif-Viorel, *D-ForenRIA: Distributed Reconstruction of User-Interactions for Rich Internet Applications*, in Proceedings of the 25th International Conference Companion on World Wide Web, pages 211–214, 2016.

In addition, we have also submitted a paper titled *Recovering User-Interactions of Rich Internet Applications through Replaying of HTTP Traces* to the Journal of Web Engineering. The paper is under review.

1.7.2 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 (Figure 1.7). In our case study, the attacker visits the vulnerable web site (part 1 in Figure 1.7), uses an SQL-injection vulnerability to gain access to private information (part 2 in Figure 1.7) and transfers some money to her own account (part 3 in Figure 1.7). She also uncovers a cross-site scripting vulnerability that she can exploit later against another user (part 4 in Figure 1.7). This session creates the trace depicted in

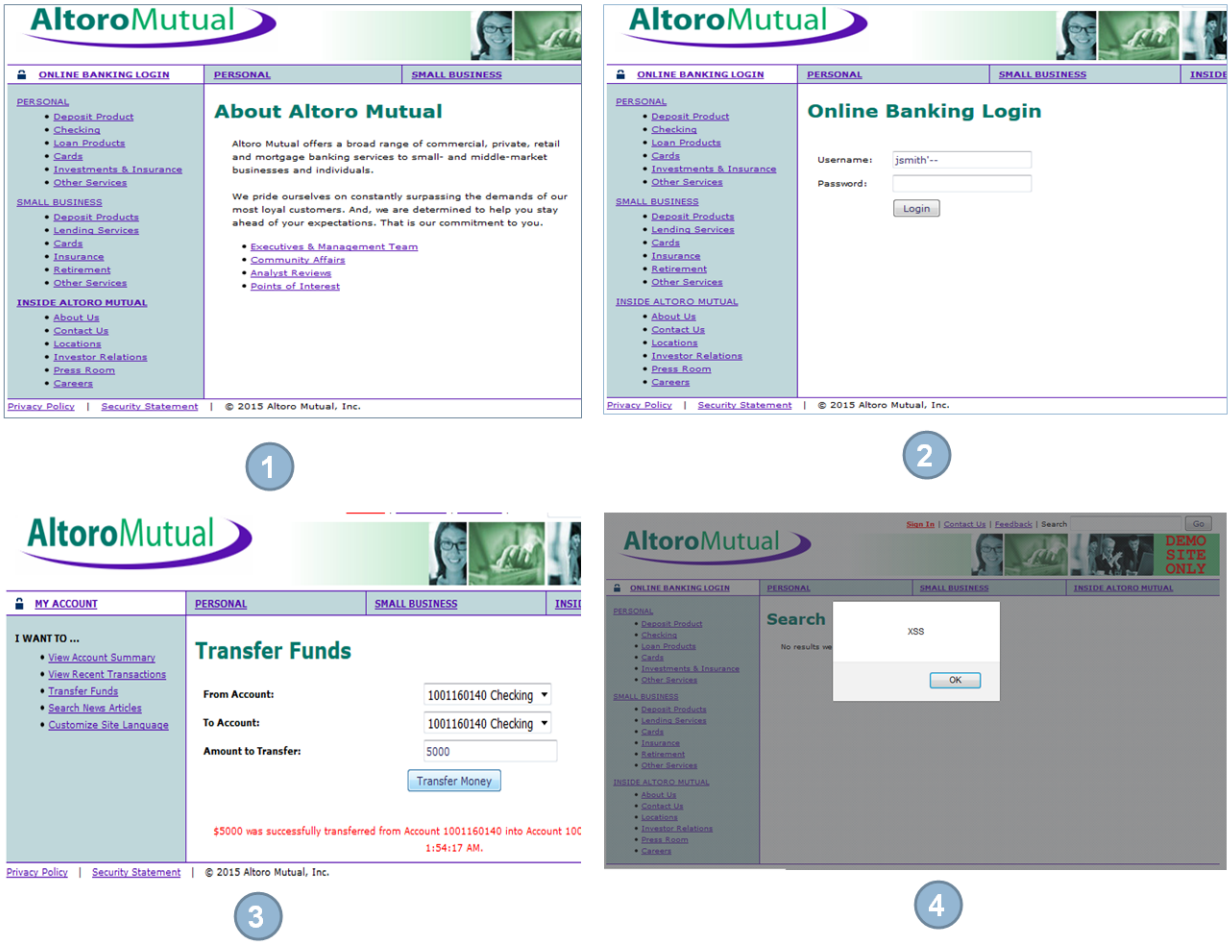


Figure 1.7: Screenshots of an attack scenario

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, ...

Figure 1.8: Portion of the HTTP log for an attack scenario

Figure 1.8. *D-ForenRIA* can help to recover what the attacker has done during the session from this input log.

Figure 1.9 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 Figure 1.9). The users can configure different settings such as the output folder (region 2 in Figure 1.9) and observe some statistics about the progress of the reconstruction (region 3 in Figure 1.9).

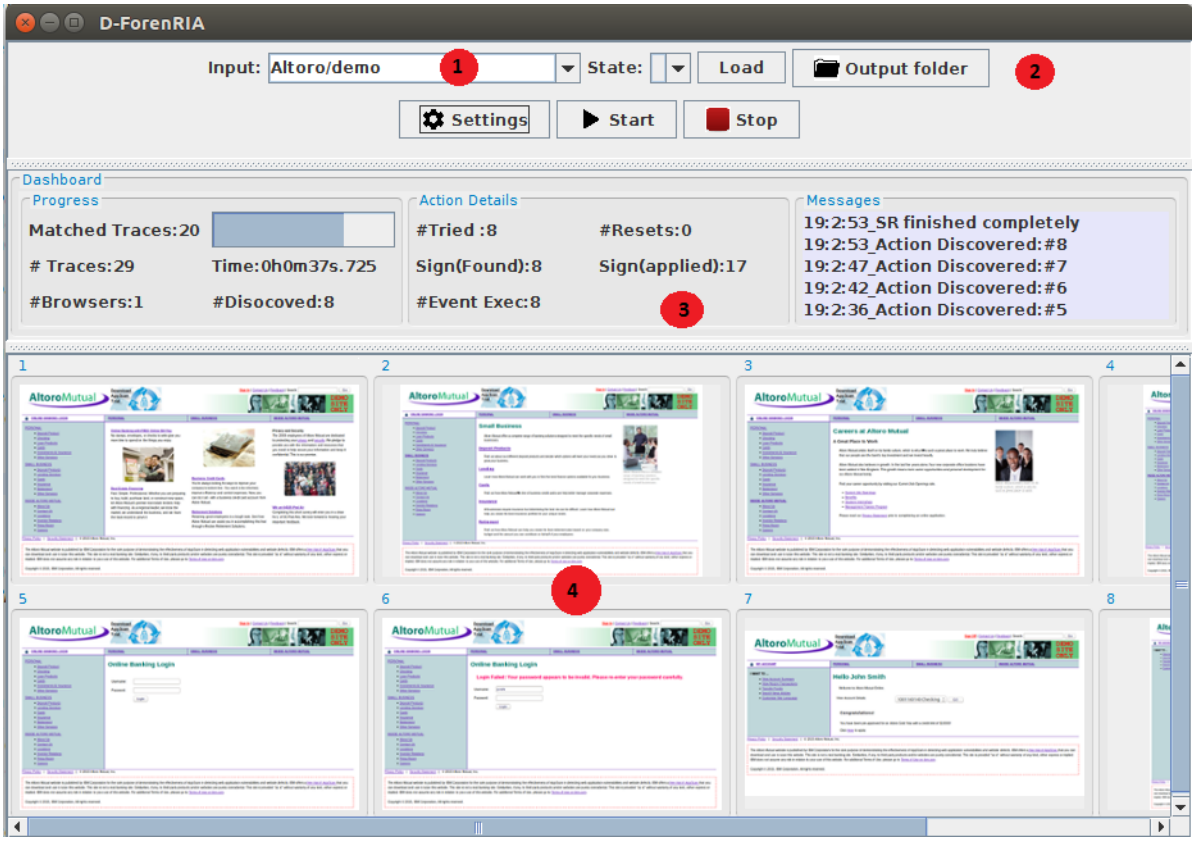


Figure 1.9: Screenshot of the main window of *D-ForenRIA*

Given the full traces of this incident (Figure 1.7), *D-ForenRIA* reconstructs the attack in a couple of seconds. The output includes screenshots of all pages seen by the hacker (region 4 in Figure 1.9). To see the details of a recovered user action, a click on one of the thumbnails opens a new window. For example, Figure 1.10 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. For example (Figure 1.11) presents the DOM of the page after the cross-site scripting attack.

A forensic analysis of the attack would have been quite straightforward using *D-*

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

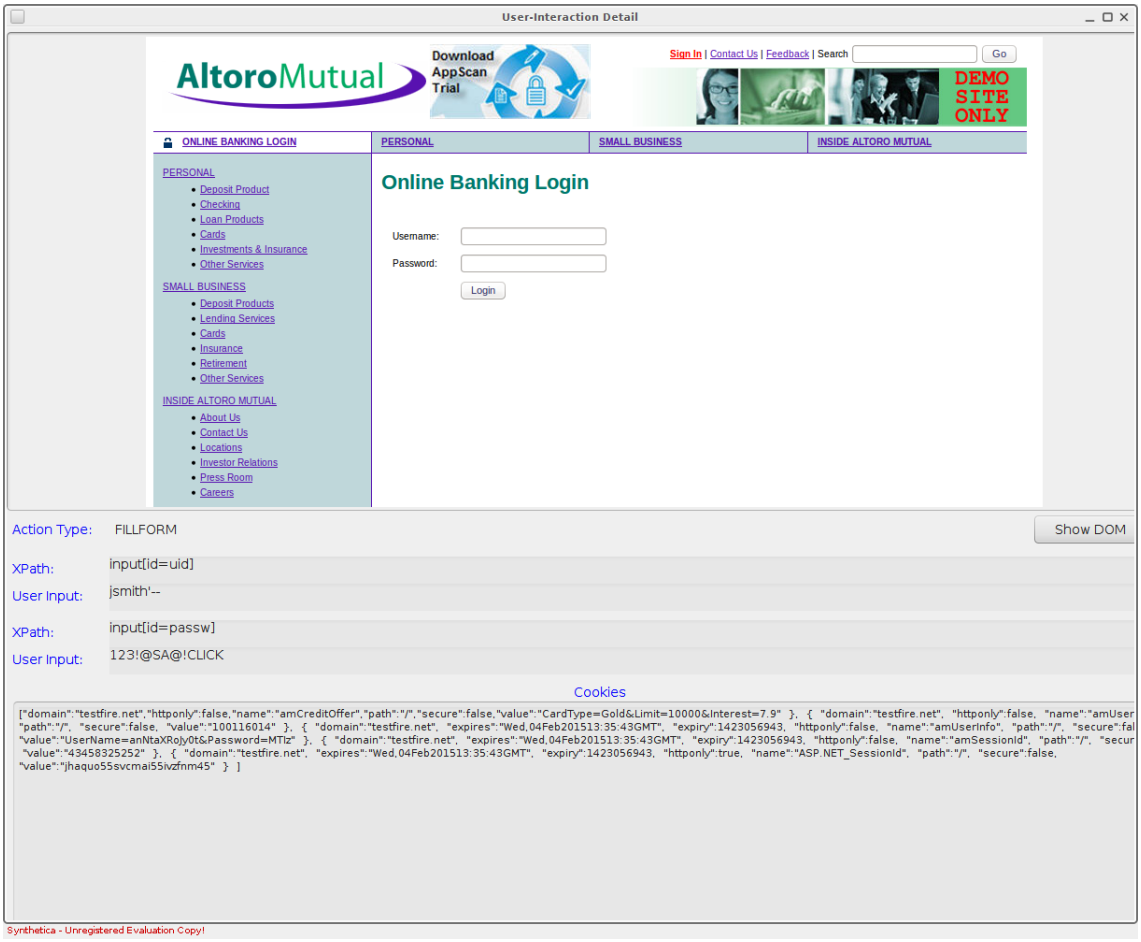


Figure 1.10: Details of an SQL-Injection attack in *D-ForenRIA*

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

1.7.3 Organization

The rest of this thesis is organized as follows:

- In Chapter 2, we present an overview of the existing methods for session reconstruc-

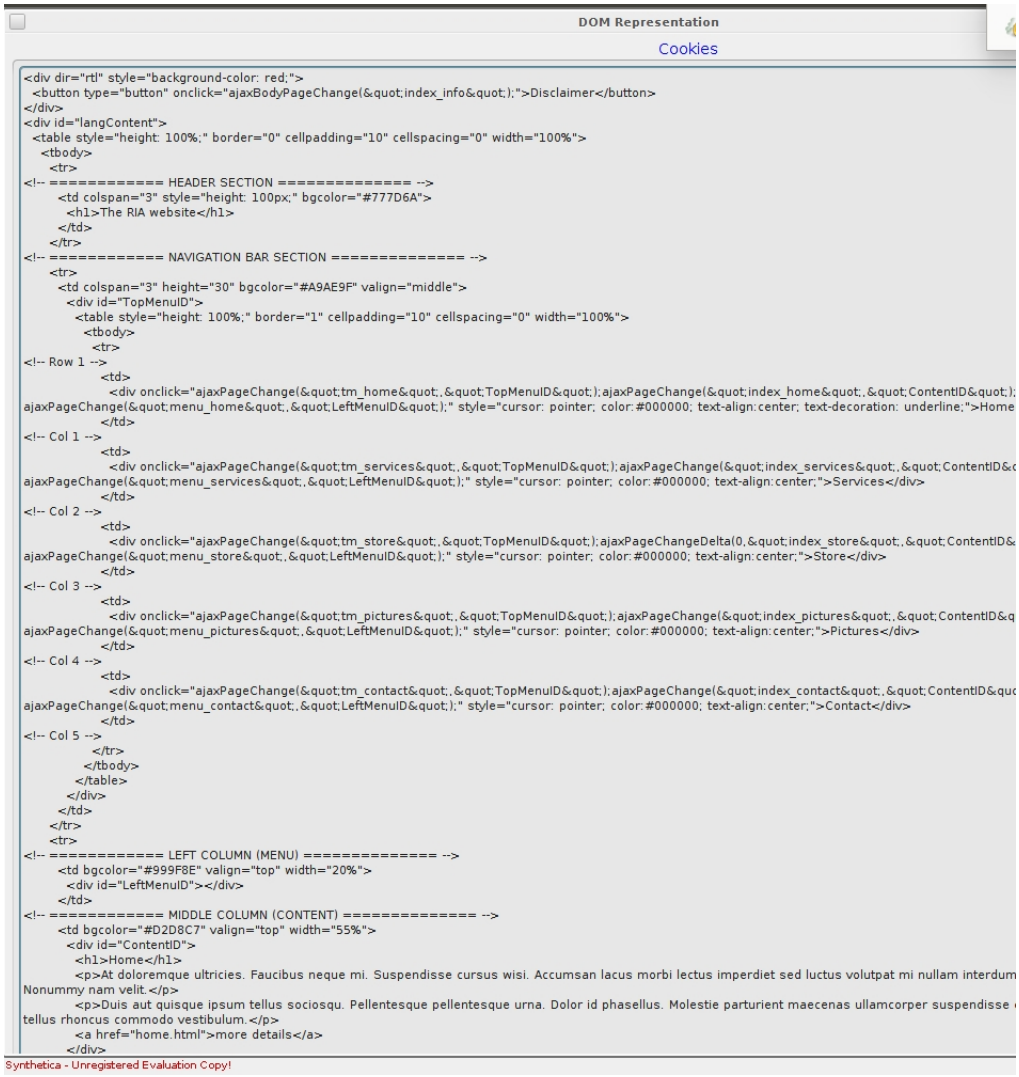


Figure 1.11: Screenshot of the DOM of a recovered state in *D-ForenRIA*

tion. This chapter also includes the challenges faced by session reconstruction tools and the requirements of a practical tool.

- In Chapter 3, an abstract session reconstruction algorithm for client/server applications is introduced. This chapter also includes several improvements to this algorithm. These improvements include parallel session reconstruction, signature-based ordering, extracting input-values, and handling randomness in the requests.

- In Chapter 4, we explain how the abstract algorithm is implemented in the context of RIAs, in a tool called *D-ForenRIA*. We explain the architecture of the tool, extraction

and identification of the candidate user-actions, and other implementation details of the tool.

- In Chapter 5, We explain the similarity-based ordering technique and discuss limitations of this technique.
- In Chapter 6, We have used *D-ForenRIA* to reconstruct sessions of a set of six RIAs. Efficiency of *D-ForenRIA* in different configurations are explained in this chapter. This chapter also includes a critical discussion on the findings of the experimental results.
- Finally, in Chapter 7, we present the concluding remarks and provide future research directions.

Chapter 2

Literature Review

2.1 Introduction

A user session starts when a user visits a website, and includes a set of user-browser interactions. During a session, several HTTP requests/responses are exchanged between a user's browser and the server. These HTTP messages are usually logged for further analysis.

Session reconstruction refers to the identification of sessions and extraction of information about what has happened during the session. There are several related problems in session reconstruction:

1. User Identification: Given traces of multiple users, user identification tries to differentiate between traces of different users.

2. Session Identification: After identifying the traces of a single user, it is desired to detect traces that belong to a single session. Session identification[1] can be considered as the most basic form of reconstructing a session.

3. Head Request Detection: There are different sources for HTTP requests which are triggered during a session. Some requests are generated as the direct result of a user's interaction with a browser (called "Head requests" [77]). On the other hand,

[1]Some researches refer to this task as "session reconstruction"

when the browser receives the response to a head request and starts processing the response, the browser may generate more requests (called "non-Head requests"). For example, the request for an HTML page is a head request and requests for images inside that page are non-head. Head requests are considered more important to detect users' actions. In the head detection problem, the goal is to find head requests out of a given trace for a single session.

4. User-Browser Interactions Detection: Recently, there has been some research to find detailed information about users' actions during a session [58, 9]. The output of user-browser interaction detection includes the elements clicked, user inputs provided and screenshots of the pages visited during the session. In order to extract detailed information about session actions, more advanced techniques and input are required; while session/user identification and head detection can be done by using HTTP headers, the proposed approaches for detection of user-browser interactions need both full HTTP requests/responses and programmable browsers.

2.1.1 Categories of Session Reconstruction Methods

Session reconstruction (SR) methods either try to extract actions from previously captured HTTP traffic (Reactive methods), or they gather information about each action during the actual session (Proactive methods). *Reactive* and *Proactive* methods [27] each have pros and cons as we describe below:

- *Reactive methods*: In the reactive approach, the input of session reconstruction is the previously captured HTTP traffic of a session. The input data is usually given in the form of server logs as HTTP logs. One challenge for reactive methods is actions which do not generate any traffic, or actions that have requests served from the user's browser's cache; since the reconstruction is based on the recorded HTTP traffic, it is not easy to find these actions. The main advantage of the reactive approach is that it does not need additional software on the end-user's browser, and can also be applied to any website without changing the website.

- *Proactive methods*: Proactive methods gather information about a user's actions during the actual session. There are various approaches for implementing proactive

methods; cookies can track what pages a user has visits on the website. Another proactive approach is sending user-tracking scripts to the user's browser to keep track of what happens during the session. User actions can also be recorded using modified browsers [52].

Although proactive methods can reconstruct the session perfectly, these methods have several disadvantages. Firstly, users may be reluctant to participate in user-tracking systems, and they may delete cookies for privacy concerns. Secondly, it is not always easy and convenient for users to install additional software on their machine which makes these approaches less deployable.

In the remainder of this chapter, we discuss the important research in reactive and proactive SR (Sections 2.2, 2.3) and also consider challenges that should be addressed (Section 2.5). Section 2.6 discusses the evaluation of session reconstruction tools, and finally, we present other related topics to SR such as Web usage mining (Section 2.7.2).

2.2 Reactive Session Reconstruction

Historically, session reconstruction involved being able to find which pages a user had visited during a session. Session reconstruction is often considered a preprocessing step for Web usage mining [25, 30, 73][2]. Here we report different subproblems in session reconstruction, which are *session identification*, *head detection* and *user-browser interaction detection*.

2.2.1 Session Identification

In *session identification* the goal is to detect traces for a single session given the traffic, which may contain requests for several sessions. Two main methods *Time oriented* and *Navigational oriented* use different heuristics to solve this problem:

- *Time-oriented*: Each HTTP request has an associated time, which shows when it has been generated. In time-oriented approaches, it is assumed that two requests

[2]Here we use term "*Session identification*" to differentiate between this rather simple task with more detailed session analysis tasks.

which are far away from each other belong to two sessions. The problem with this approach is finding a threshold. The proper value of a threshold depends on several factors such as network speed, the complexity of the page and information on the page [22, 72].

- *Navigational-Oriented*: During a session, a user visits (navigates) different pages of a website. There are some data in HTTP requests which reflect this navigation between pages. The *referer*[3] header of a request for a page represents the previous page which has issued the current request. This navigational information has also been used to detect sessions.

  Cooley et. al. argued that if the *referer* of a page P does not exist in the current session, P represents a new session [22]. However, in the absence of *referer* header, they cannot decide about the inclusion of a page in the session. The empty *referer* can be because of directly typed URLs in the address bar of the browser, or due to the requests made by robots/AJAX calls. Topology-based methods ,which consider actual links between pages, can alleviate problems caused by missing *referer* data.

  There are also application-specific heuristics. Berendt et. Al. assume that if in the given HTTP trace there are two consecutive URLs, A and B, and A is a prefix of B, both A and B belong to the same session since B has more parameters and is supposed to refine the first page [11].

Challenges during session identification: The heuristics used in session identification methods are based on various assumptions about user behavior and are not necessarily correct at all the times. For example, in previous research it is assumed that users navigate mostly through pages by clicking instead of typing a URL. In such an approximation, some activities of a session may be missing while others are erroneously detected. The other source of these session identification errors is noisy and incomplete server logs [72]. Spiliopoulou et. al. discussed the importance of removing noise from server logs, and evaluated several heuristics to reconstruct user sessions [72]. However, the session identification is mostly a problem with websites and not for Web applications; because a Web application usually has the built-in ability to identify sessions.

[3]Originally a misspelling of referrer [38].

2.2.2 Head Request Detection

Various researchers have emphasized the importance of distinguishing between the HTTP requests initiated by a user action (called *Head requests*[4]) and the other requests [77, 69]. Srivastava et. al. considered non-Head requests as redundant since they do not depict the actual behavior of the user, and proposed that these requests should be deleted from HTTP-log by checking the data type of the requested URL [74]. Sha et. al. also proposed removal of images and bad request (HTTP status code 400) as a preprocessing step for Web mining [70]. Several approaches have been proposed to find head requests.

Barford et. al. proposed a naive approach to detect head requests that classifies requests for HTML objects as head requests [10]. A more elaborated approach considers the idle time between two requests; two consecutive requests with a small idle time between are considered to belong to the same user request [49, 71]. However, this approach fails to consider the complexities of the modern HTTP traffic. *StreamStructure* detects head requests based on web analytics beacons [43]. A key limitation of *StreamStructure* is its dependency on Web analytic beacons since these beacons do not necessarily exist in all websites. Schneider et. al. proposed that head requests can be detected by manually defining patterns for these requests [69]. However, this manual approach is not general, and patterns should be defined for any new website.

Resurf uses referer header in order to detect head requests [77]. *Resurf* also considers the data type of requests, the number of embedded objects inside a page, the size of the response and also the delay between the current request and the previous requests. *Resurf* also removes HTTP redirections by combining the endpoints of the HTTP redirection in the *referer* graph. However, since AJAX requests usually don't have a *referer* field, *Resurf* is not applicable to RIAs.

2.2.3 User-Browser Interactions Detection

Detection of user-browser interaction provides a detailed analysis of actions that have been performed by a user during the session. To the best of our knowledge, few works [77, 58, 65]

[4]Xie et.al. [77] refer to *head requests* as *user requests*

have been done for session reconstruction in RIAs. None of the previous methods handle complex RIAs, and their focus is on traditional Web applications.

A graph-based method, RCI tries to reconstruct user-browser interactions [58]. 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 are compared with DOM elements to find a triggering element for each request [77, 58].

ClickMiner reconstructs user sessions from traces recorded by a passive proxy [58]. There are two components in *ClickMiner*; an instrumented browser and a proxy which has access to the recorded traffic. The browser is redirected to the initial URL, and eventually makes successive calls to render the page. The browser then asks proxy for the next expected URL, and tries to navigate to that URL either by clicking on an element or directly navigating to that URL. The output is a set of $(p,\ e,\ q)$ tuples where p is the current page, e is an event on the page and q is the next page. *ClickMiner*, focuses on actions which change the URL of the application, however, many actions in modern RIAs do not alter the URL. In addition, although there is some level of support for JavaScript, *ClickMiner* is lacking the specific capabilities that are required to handle complex RIAs [9].

Recently, Raghavan et. al. have proposed a method, called *Rachna*, to reconstruct multi-tabbed user sessions [65]. HTTP/browser logs do not contain information regarding opened browser tabs[61]. To solve this problem, *Rachna* links browser logs, HTTP logs and traces in lower layers of the network. The working condition of *Rachna* is recording traffic on the user's machine that is often impossible, and may arise users' privacy concerns.

2.3 Proactive Session Reconstruction

In general, proactive tools have more detailed information on disposal. It is due to the fact that these algorithms can log any required information on server/client side when the actual session happens. In other words, they are not limited to minimal information available in the HTTP request/response logs. However, even with full control of recording time, they still face several challenges that will be discussed in Section 2.4.

2.3.1 Session Identification

In proactive session identification, sessions are detected using cookies or user authentication. This identification can also be done by server-side mechanisms such as assigning a session identifier to each browser process which expires after a timeout or after the lifetime of the browser process.

2.3.2 User-Browser Interactions Detection

In proactive methods, the data about each user-browser action is collected during the actual session. This data can be recorded using different approaches. UsaProxy is a tool that uses a proxy which inspects HTTP responses from the server, and injects a specific JavaScript code in the response [6]. The injected code keeps track of user's actions while the user navigates through the website [7, 6]. Although proxy based approaches are easy to deploy, they arise privacy concerns for end-users. Instead of using proxies, developers can also add user-tracking scripts to their code. The most popular example of these systems is *Google Analytics* [21]. The commercial products *ClickTale*[5] and *ForeSee cxReplay* [6] capture mouse and keyboard events in Web applications. The capturing is done by including special JavaScript codes in the pages which track users, and sends aggregate data periodically to the reporting server.

The actions of the user can also be logged with browser add-ons. The Selenium IDE (implemented as Firefox plug-in) [7] and iMacros for Chrome[8] record user actions and replay them later inside the browser. *Mugshot*, records user actions and also some behind-the-scenes activities such as the firing of timer callbacks and random number generation logs [52]. *Mugshot* injects particular script in the web pages of the application, and creates a log entry containing the sequence number and clock time for each recorded event. Andrica et. al., presented a system called *WaRR* to record user-browser interactions [5]. The WaRR recorder is also embedded in a Web browser, and captures a complete interaction trace. The WaRR recorder logs a sequence of commands, where each command is a user action.

[5] http://www.clicktale.com/
[6] http://www.foresee.com/products/web-mobile-replay/
[7] `http://www.seleniumhq.org/`
[8] `http://wiki.imacros.net/iMacros_for_Chrome/`

2.4 Reactive vs Proactive Session Reconstruction

There are different approaches for proactive session reconstruction; one may use cookies to keep track of pages visited by the user. However, users may disable logging mechanisms because of privacy concerns, for example, they may delete cookies to disable being tracked by a website (or even by several websites). Such privacy concerns are valid, and websites should provide an "opt-in" policy for these types of logging [52]. The other proactive method is to use a proxy. However, it is not convenient for a normal user to set the proxy of the browser to enable tracking. Instrumenting the application provides an easy solution for session reconstruction. However, when one does not want or cannot do the instrumentation (such as when there is no access to the application), reactive methods are useful. Most importantly, when no instrumentation was done on the application and you need to reconstruct the session, the only option is to use a reactive approach.

Despite these interesting features, reactive session reconstruction is an inherently difficult problem due to the limited information contained in the recorded traffic. The problem is worse in the case of RIAs where there is no direct clue in HTTP traces to find the triggering actions [58]

2.5 Challenges in Reactive Session Reconstruction

In this research, we propose a reactive session reconstruction tool for RIAs since there are already approaches that can reconstruct sessions for traditional websites [58]. Here, several subproblems that need to be solved are discussed.

2.5.1 Non-Determinism

The non-deterministic behavior of a web application during replay has been considered as one of the challenges for record/replay tools [52, 24]. Here non-determinism means that execution of an action during replay time, generates a different set of HTTP requests which do not match with the previously recorded traffic. Since actions are found using the corresponding HTTP traffic, the session reconstruction cannot easily detect actions that

include random parts. This non-determinism in the generated requests can have different causes.

One reason for non-determinism is JavaScript functions such as *Date*() or *Math.random*(). These functions usually returns different values during record and replay time. So, if a request is generated based on the output of these functions, the requests will probably be different during replay compared to the recorded requests. Developers add random parts to the requests to disable caching. Although caching is indeed a useful feature to save bandwidth, and makes the Web application faster it can prevent the user from receiving the last current version of the Web Application. The reason is that the requests are served by the browser from the old cached response version. However, by adding a small random part of a request, the request does not match with browser's cache and therefore sent to the server.

Another source of non-determinism is AJAX, which allows JavaScript applications to send asynchronous requests to the server and partially update the current user interface. Since the requests are asynchronous, repeated execution of a function can generate the same set of requests in a different order. This fact should be taken into account when trying to find the correct action which has produced a group of requests [9].

Furthermore, any environmental difference between record and replay environments (such as operating system and browser) can cause non-determinism. *Mugshot* replays events inside the same browser used at recording time [52]. Although conceptually straightforward, the recording process is complicated by various browser incompatibilities and implementation deficiencies of some events [52]. Although JavaScript itself is standardized [4], there are many subtle cross-browser incompatibilities. For example, ordering of execution of event handlers when several handlers are attached to a single DOM element is not standard.

There have been several solutions to handle randomness in requests, Neasbitt et. al. proposed an approximate matching solution [58]. The URLs are compared based on the URL, name and number of matching parameters and other temporal features. Finally, the most similar resource is returned to the browser. However, this approach can lead to an incorrect matching, and needs a predefined threshold for similarity. *Mugshut* uses another approach to respond to randomly generated requests based on time functions [52]. *Mugshut* records the current date and seed random generator during the recording time

to be used during replay time. Nevertheless, this method cannot be applied to approaches which rely solely on the previously recorded HTTP traffic. *ForenRIA* uses two browsers to handle random requests [9]. Two browsers are running in parallel, and each action is executed twice so the random parts of requests can be easily extracted by comparing the traffic coming from each browser.

RIAs have less problem with non-determinism compared to multi-threaded applications. In a multi-threaded environment, recording/replay can face several problems to correctly replay the previously recorded state of the application [52]. Since JavaScript applications are single-threaded, the reconstruction can assume that no other function is going to be executed before finishing the current function; this assumption makes the reconstruction easier.

2.5.2 Detection of Stable Condition

If a method uses a browser during session reconstruction, it repeatedly executes actions on the current DOM. After executing an action, the browser should wait until the execution is finished, all generated requests are received, and the DOM is rendered. This state is called a "stable condition" [9] or a "resting state" [58]. In the absence of "stable condition", the browser may execute another action too early, an action that is not present on the DOM yet.

There are different approaches to detect the "stable condition". One naive approach is to pause for a fixed amount of time after the execution of each action. This approach assumes that a stable condition reaches during this predefined pause time [51]. Since this approach uses a fixed amount of time for all actions, it is not efficient; many actions do not need a large amount of waiting time, and can be executed much faster. Selenium IDE [9], verifies reaching a stable condition by checking the presence of DOM elements. Although this approach is accurate, it does not consider receiving all HTTP responses by the browser.

Mugshot uses a proxy at replay time to inject a custom load handler [2] for the target DOM node so that it can determine when the load has finished, and the stable condition is reached [52]. It also injects script tags at the beginning of the DOM, which are guaranteed to be loaded and executed before any other script because of synchronous execution of script

[9] `http://www.seleniumhq.org/`

tags by the browser. *ClickMiner* considers changes in the DOM and the number of pending requests as criteria to detect the stable condition. The system puts an upper limit of 250 seconds waiting time to handle cases where the page repeatedly issues requests to update itself [58]. *ForenRIA* considers pending requests, and overrides callback functions of AJAX requests to handle stable condition more accurately [9]. *ForenRIA* also overwrites timer functions of JavaScript (*settimeout* and *setinterval* [10]) to deal with periodical requests.

2.5.3 Caching

Caching is being used widely in today's Web to save bandwidth and loading time. According to a report by *Internet Archives*, which covers Alexa Top 1,000,000 sites, nearly half of all the downloaded responses can be cached by browsers[11]. When a resource is cached, the requests are served locally from the cached response instead of asking the server to provide the response. Caching saves the network traffic as well as the loading time, especially for large resources (e.g. multimedia objects). Resources are cacheable unless explicitly prevented from being cached using HTTP headers such as no-cache or no-store [39].

When the reconstruction is done by recorded HTTP traffic, caching can present a problem [58]. The reason is that caching prevents the recording of all requests by the server, and thus blurs the picture of user behavior; this side-effect of caching jeopardizes any knowledge discovery task which tries to analyze the navigational behavior of users [72]. 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. Another approach, "cache busting" assumes that caching is disabled [22].

A related problem ,"Path completion", has been discussed in Web mining research [47, 56]. Caching creates missing pages in the user's traversal path for the tool which tries to detect these missing pages. Because users can access pages from the local (or proxy server's) cache, and there will be no record in the servers access log. Path completion differs from the other steps during data preprocessing such as noise removal. While other preprocessing tasks remove some data or divide the data up according to users and sessions, "path completion" adds some information to the user-log. Spiliopoulou et. al. proposed

[10] `http://www.w3schools.com/js/js_timing.asp`
[11] `http://httparchive.org/trends.php#maxage0`

a method to solve path completion problem which is caused by using browser's "Back button" [72]. The method has two steps:

- If page, *L'*, does not have a direct link to the previously requested page, *L*, the method uses the *referer* header. If the referrer page , *X*, is in the recent history of pages, the method concludes that the user navigated from *X* to *L'*.
- If the referrer is not clear, site topology which is a graph with nodes as pages and edges as the links between them can be used to infer missing pages. If we find a sequence of the most recently visited pages, say P_1 ... *L*, where page P_1 has a link to *L'*. The method assumes that the user has clicked on "Back button" to traverse from *L* to P_1, and finally to *L'*. This path is added to the final path to detect cached pages served from cache by clicking the back button.

All previously proposed path completion methods use relationships between pages to address this problem. 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, no research has been reported on path completion in RIAs yet.

2.5.4 Handling RIAs

There has not been much research on session reconstruction in RIAs. One system, *Click-Miner* tries different actions blindly on the page when the application is a RIA [58]. However, this approach is not efficient since it blindly clicks on different candidate actions on the page to find the right action [9]. A more practical method, *ForenRIA*, has been recently reported by our group [9]. *ForenRIA* reconstructs sessions of RIAs much more efficiently since it has a learning mechanism during reconstruction, and improves its knowledge of patterns of request/responses generated by each action. It considers the attributes of elements involved in an action to further order actions, for example hidden elements or element without any event-handler have the minimum priority to be tried.

2.5.5 non-DOM User Interactions

A user can interact with different parts of a browser. The main responsibility of a browser is rendering pages (DOM of the application). When a user performs an action on the page, it interacts with the DOM. However, users can also use other parts of the browser (out of the DOM) to navigate around the website. Users can use the address bar or shortcuts to their favourite websites. Detection of actions that do not involve DOM can be challenging. However, there have been some solutions to address this problem.

ClickMiner considers navigation to a new URL when there is no hyperlink on the current page pointing to that URL as an "unconfirmed" action [58]. This action is called "unconfirmed" since it can have different sources such as typing in the address bar, or clicking on a bookmark or URL redirections from Flash objects. In another approach, *Mugshot* does not log events which are triggered by objects which do not have DOM API (e.g. Flash objects or Java applets) [52].

2.5.6 Recording the Usage Data

Usage data is logged by most web-servers by default, and can be accessed readily for analysis [67]. The basic server log format only involves IP address/DNS hostname of the users host [1]. An extended log file format that is widely used by web-servers such as IIS[12] and Apache[13] allows recording of *user-agent* and *referer*. *user-agent* contain user's browser's name, and *referer* contains the page from which a request was initiated. In addition, HTTP servers such as IIS can be configured to record the full HTTP traffic[14].

2.6 Requirements of Reactive Session Reconstruction

There are several functional and non-functional requirements for a session reconstruction system to be acceptable. The primary functional requirement is the ability to discover all actions from a given HTTP trace. There are several non-functional requirements as well,

[12] `https://msdn.microsoft.com/en-us/library/ms525807%28v=vs.90%29.aspx`
[13] `httpd.apache.org/docs/2.2/mod/mod_log_config.html/`
[14] `https://msdn.microsoft.com/en-us/library/ms227673.aspx`

including ease of deployment, fewer input data, less time to reconstruct the session, and some other requirements:

- *Full Reconstruction*: Full reconstruction refers to correct detection of actions recorded in recording time during replay time [72]. There are a number of criteria, such as precision/recall, that can measure the quality of the reconstruction. Precision is defined as the means of determining how many detected actions are correct, while recall is defined as the means of determining how many of user-interactions have been found by the session reconstruction algorithm. Full reconstruction happens when we have the perfect precision and recall.

- *Reconstruction Time*: The speed of the reconstruction algorithm can be measured by the time needed to reconstruct each action. It can also be measured as the time required to restore a session with different lengths (as used in *ReSurf* [77]).

- *Minimum Required Information*: Different reconstruction algorithms also vary in the amount of information needed in order to have a successful reconstruction. It can be as minimal as HTTP request headers (as in *Resurf* [77]) or as detailed as full HTTP request/response (as in *ClickMiner* [58]). However, there is a natural trade-off between efficiency and accuracy. While methods which use minimal input information are quick, they cannot provide accurate information when compared to methods which utilize full HTTP logs.

- *Easy Deployment*: Developers prefer that clients need not install recording tools on their machines, nor change the Web application to be able to record/replay traffic. Easy deployment makes users interested in using the system, and makes the record/replay tool effective since it provides an unobtrusive record of events during logging time [52].

- *User-Log Recording Overhead*: Log recording must not cause an overhead in storage and computation. In order to measure storage efficiency, Mugshot proposed a KB/Min ratio. In order to measure computational efficiency, Mugshot proposed measuring the rate at which execution time decreases [52].

- *Detailed Reconstruction*: Session reconstruction tells us about the user interactions during a session. The detailed reconstruction finds for each action all the information

required to execute that action. For example, with keyboard events, the system requires the user to record the GUI element that has received the event and the character code for the relevant key. For form submissions, the tool needs to record input-elements, their corresponding value, and the element which submits the form. However, the less detailed reconstruction only reveals some of the meta-data about actions; for example, it detects which HTTP requests are initiated by an action and which are triggered by the browser.

- *Resistance to Disabled Recording*: The quality of reconstruction depends on recording sufficient data for the algorithm. There are a number of different scenarios that find the user disabling the recording mechanism. For instance, users can delete cookies for privacy concerns, or attackers may trigger interactions through a variety of elements which do not have a DOM API [58]. Reactive methods are more resistant to disabling traffic recording since the traffic is logged out of the user's machine.

2.7 Related Topics

2.7.1 Specially Instrumented Execution Environments

Some of the prior session reconstruction systems are intended to debug an application. These systems attach to a running application, and record its execution state [57, 37, 28]. Xu et. al. mention the non-determinism problem which is caused by multi-threading and non-repeatable inputs, and proposed a practical low-overhead hardware recorder [78]. *DejaVu* also introduces a method for deterministic replay of Java multi-threaded applications [20], and *Retrospect* [13] presents a method for record/replay for programs written using an MPI interface. The *liblog* system [37] provides a high-level recording of all inputs/outputs of an application with the operating system. The recording is done using a *C-library* layer which stands between the application and operating system. Another approach *TimeLapse* [15] provides deterministic record/replay of user-browser actions. *TimeLapse* has been built on a special framework (called *Dolos*) which records all inputs (including network inputs) of a Web application. Developers can use debug mode commands such as break-point to test actions. In addition, logging does not have consid-

erable overhead. *Jockey* is another system which records invocations of system calls and CPU instructions for debugging Linux programs [66].

However, these systems usually face deployment problems since they require instrumenting the execution environments such as a custom kernel to capture a programs execution. Besides, these instrumented execution environments produce log data at a high rate, and only report the last few seconds of system state before error [52]. These approaches also do not consider the single-threaded non-preemptive nature of JavaScript web applications. The browser will never interrupt the execution of one handler to run another. JavaScript provides sufficient overriding capabilities on unmodified browsers to log client-side non-determinism [52]. Thus, to test RIAs it is enough to record the content and the ordering of events for successful replay. The single-threaded nature of JavaScript, dramatically reduces the overhead of logging, both in processor time and storage requirements, since the recording is done on user interface and a single-threaded model instead of instruction level.

Ripley provides a mechanism for automatically checking the security of AJAX applications [75]. *Ripley* replicates a copy of client-side computation on the server. When the client triggers an event, it is being sent to the server, and forwarded to the replica for execution. The server only executes a client RPC[15] if the replica also generates it. *Ripley* can be considered as on-the-fly record/replay tool which tries to replicate events from client-side to server-side for an integrity checking.

2.7.2 Web Usage Mining

Web usage mining refers to the automatic discovery of patterns from an HTTP server logs [48]. Web usage mining may utilize access logs, the HTML files that make up the site, and any optional data such as registration data or remote agent logs [22]. The goal of Web usage mining is finding straightforward statistics, such as page access frequency, as well as conducting a more sophisticated form of analysis such as behavioural patterns of users with common needs. Usage mining has several applications ranging from redesigning of the website's user interface, planning a physical data layout or a caching scheme for a distributed Web server [47]. Another application of Web usage mining is "page prefetching" in which the next request needs to be predicted given the requests performed

[15]Remote procedure call [23].

thus far [72].

Much research has been conducted on Web usage mining [54, 14, 45]. There are three phases in Web usage mining process: preprocessing [42], pattern discovery and pattern analysis [62]. Data preparation is considered essential during any knowledge discovery process. It is due to the fact that without this step, the data becomes useless for further analysis [42]. The most frequent data preparation tasks are detection and handling of erroneous and missing values [72]. In Web usage mining, it is essential to eliminate outliers and irrelevant HTTP request/response pairs, to reliably identify unique users and user sessions within a server log, and to identify semantically important transactions. Proper preprocessing needs some information about the application such as the website designers view of how a site should be used [22]. The generated patterns are usually presented as association rules such as "customers who visit page A are likely to visit page B", sequential patterns, and similarity analysis based on Web access patterns [47].

One of the subproblems in Web usage mining is the *Session Identification*. A session refers to one visit to a website and *session identification* relates to identifying the subset of HTTP requests/responses which belong to a single user's session given a set of logs which belong to several users. This task is a considered as a fundamental preprocessing task in Web mining [72].

One of the difficulties in *session identification* is the existence of proxy servers which assign the same IP address to different users. Li et. al., proposed a referer-based method to solve this problem [47]. It first tries to distinguish users by their IP, it then differentiates users with the same IP with their browser and the operating system (available in HTTP headers). To break ties further, the method uses the *referer* information: If the URL in *referer* header has not been accessed previously or there is a significant time gap between the current request and the previous request (more than 10 seconds [72]) this request is considered as the beginning of a new session. Cooley et. al. also use a timeout to detect sessions. If the time between two subsequent requests exceeds the timeout, it is considered as a new session. The average timeout of 30 minutes have been reported, and being used in many commercial web mining tools [18]. However, these timeouts are based on the characteristic of traditional web traffic, and they may not be accurate because of rapidly changing nature of new Web traffic.

Cooley et. al. argued that preprocessing tasks can be done more efficiently using site

topology which can be extracted by crawling the website [22]. They also discuss that preprocessing tools should differentiate between what they call "content pages" which are the main pages on the website for users to interact with and "auxiliary pages" (such as site-map and look-up pages) which provides access to the content pages. They proposed using physical and usage characteristics to classify pages of a website. In addition, they suggest using site topology to detect session. If there is a page request that is not reachable from any pages which have been requested by the user, the method assumes that another user has requested this page. However, this method cannot handle situations where one user has two browsers, a user who types URLs directly in browser's address bar, and two users with the same set of visited pages on a single machine.

In Web mining research, there have also been several methods in data preprocessing phase to filter some HTTP request/response pairs. Cooley et. al. also argued that irrelevant information should be eliminated, and it is crucial to any web log analysis [22]. Li et. al. proposed elimination of the following categories of HTTP records [47]:

- *Records of multimedia objects*: These records have filename suffixes such as GIF, JPEG, CSS which can be found in the URI field of the every HTTP request. Usually, it is sufficient to keep requests for HTML files in the server logs. The reason is that the Web usage mining algorithm wants to model users' behaviour, and users do not usually ask for non-HTML files directly. Removing multimedia objects depends on the website, and should be defined by analyzing expected user's behavior [22].
- *Records with failed HTTP status code*: By examining the status field of every record in the web access log, the records with status codes over 299 or under 200 which present failed requests are removed.

2.7.3 Record/Replay at non-HTTP Layers of the network

Some traffic replay systems [41] have focused on replaying traffic at an IP-level or TCP/UDP-level. These systems have been used to generate test traffic by replaying a recorded stream of IP packets. Hong et. al. present a system which performs an interactive replay of traffic by emulating a TCP protocol stack [41]. This approach is however too low level for

an efficient RIA reconstruction since these systems cannot extract information about the actions that the user has performed during the session.

2.7.4 Complementary Tools for Session Reconstruction

During replaying a previously recorded session, a developer can use tools such as Firebug[16] to inspect the internal state of the application and execute/pause actions one by one. In addition, over the last years, numerous network forensic analysis tools have been constructed [64]. These tools can aid forensics analysis by capturing, monitoring and replaying network traffic, and eventually generate appropriate incident responses. Session reconstruction tools can get input from the existing network forensics tools, and provide high-level information regarding user-browser interactions in a given session.

2.8 Conclusion

In this chapter, we talked about two main approaches for session reconstruction: reactive and proactive. The input for reactive methods is the previously HTTP traffic. On the other hand, proactive methods record detailed information about actions during the actual session and usually do not deal with the recorded HTTP. Proactive methods have deployment problems, and can arise users' privacy concerns while reactive methods do not face any of these challenges. We also talked about subproblems (such as caching, non-determinism) that should be addressed during session reconstruction. In addition, we discussed functional and non-functional requirements of session reconstruction systems and mentioned some related topics to such systems such as Web usage mining.

There has been little research on the reconstruction of user-browser interactions from HTTP logs, and none of them deal with the complexities of RIAs. The focus of our research is on efficient strategies to reconstruct sessions in RIAs.

[16] `http://getfirebug.com/`

Chapter 3

A Session Reconstruction Solution for Client/Server Applications

3.1 Introduction

In this chapter, we introduce a general solution to solve the session reconstruction problem that we introduced in Chapter 1. We start by providing a simple session reconstruction algorithm (Section 3.2). In Section 3.3 we mention the limitations of the algorithm, and propose several directions of improvements. These improvements include: efficient ordering of candidate actions (Section 3.3.1), concurrent evaluation of candidate actions (Section 3.3.2), extracting action parameters (Section 3.3.3), handling randomness in requests (Section 3.3.4), and handling non-user initiated requests (Section 3.3.5).

3.2 A General Session Reconstruction Algorithm

Here we present an algorithm for the session reconstruction of applications which are based on the client/server model. The algorithm uses three components: 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 the first/all solution(s) *findAll*
Output: The sequence of user-interactions that generate the given log, *sol*

```
1: Procedure Main()
2:    S_0 ← InitialState(R)
3:    sols ← {}
4:    SR(S_0, R, {})

5: Procedure SR(State S_n, Trace R, ActionPath ap)
6:    if R = {} then
7:       sols.add(ap)
8:       if not findAll then
9:          terminate()
10:      end if
11:   else
12:      A ← ExtractCandidateActions(S_n)
13:      for a ∈ A do
14:         R_s ← Evaluate(a, S_n)
15:         if Match(R_s, begin(|R_s|, R)) then
16:            S_{n+1} ← GetState()
17:            SR(S_{n+1}, R − R_s, ap + a)
18:         end if
19:      end for
20:   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 5. In this approach, the algorithm extracts all possible candidate actions of the current state of the application S_n (line 12), and tries them one by one (line 14). 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 R (line 15), a is considered a possible correct action at the current state[1]. Since the requests can be generated in different orders, the order of elements in R_s does not matter for the *Match* function (line 15). The action is also accepted if it does not generate any requests (Such as clicking on the "Advanced Search" link in Figure 1.5).

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 17)[2]. The algorithm finds a solution when all requests in the input trace are matched. The session reconstruction algorithm starts from the initial state of the algorithm (line 4). 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 15). 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 7).

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 12), the *Robot* triggers the action on the current state (line 14) and the *Proxy*, responds to the requests generated during the executing the action by the client (line 14).

[1]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.

[2]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.

3.3 An Improved Session Reconstruction Algorithm

Although the general algorithm (Algorithm 1) is easy and straightforward, several issues must be addressed to make it practical.

3.3.1 Signature-Based Ordering of Candidate Actions

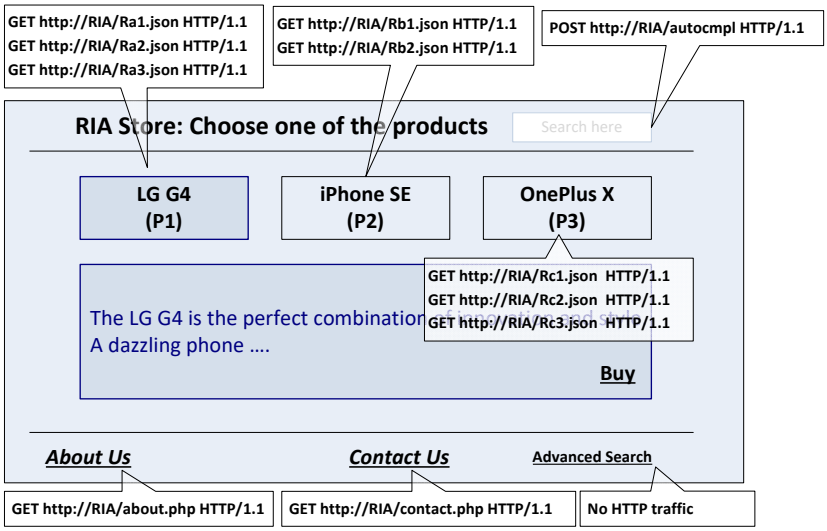
Algorithm 1 blindly tries every action at each state to find the correct one (lines 13-14). However, in practice there may be a large number of candidate actions at a given state, so the algorithm needs a smarter way to order candidate actions, from the most promising to the least promising. We propose 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 Figure 1.5 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 Figure 1.5 can be discovered by just looking at its *href* property in the DOM (Figure 4.4). If the signature for an action cannot be discovered before executing the action, the signature is extracted once the action is evaluated for the first time. Therefore, the session reconstruction algorithm does not necessarily have the signature of all actions.

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 $\in \{0, 0.5, 1\}$ to all candidate actions on the current state. The most promising actions will be assigned a higher priority-value. If the signature of an action does not match the next expected traffic, we should decrease the priority of the action (priority is 0). On the other hand, if a signature of the action matches the next unconsumed trace, that action should be considered a promising

action (priority is 1). When the signature of the action is not available (when the algorithm has not yet executed the action from any state), the priority value of 0.5 is assigned.



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
...	...

...

(b)

Figure 1.5: (a) A simple page and generated requests after clicking on each element. (b) Portion of the user's log (repeated from page 10 for convenience)

Example: Consider the simple example in Figure 1.5a and the given trace of Figure 1.5b. 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.
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 , $< Ra_1, Ra_2, Ra_3, Rc_1, ... >$, 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.

3.3.2 Concurrent Evaluation of Candidate Actions:

Algorithm 1 is sequential and single-threaded. At each state the algorithm extracts the list of candidate actions (line 14), and executes them one by one using the client (the **for** loop in lines 13-19). 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 14), 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.

3.3.3 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

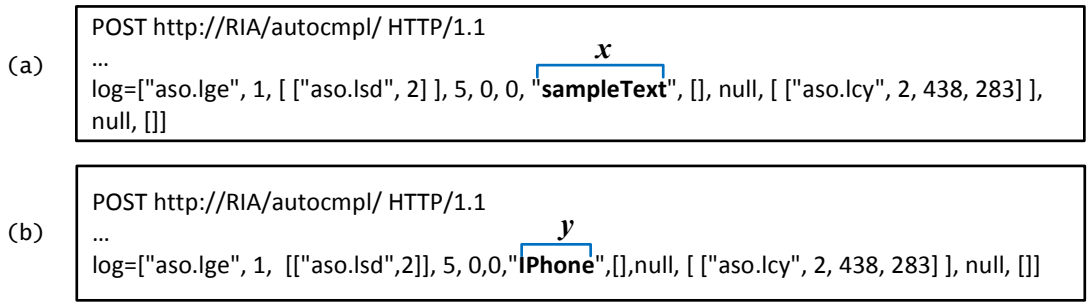


Figure 3.1: (a) The generated HTTP request after performing a user-input action using sample data. (b) the expected HTTP traffic in the trace. (x, y represent the sample and actual user-input values respectively)

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 Figure 1.5 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 (Figure 3.1a, 3.1b) 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.

3.3.4 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 15 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 order of requests/responses can be non-deterministic because of the asynchronous requests. In this case, as we explained in Section 3.2, the *Match* function ignores the order of generated requests to find the matching responses. Another form of randomness happens when the values of some parameters can change between two executions, for example when the value is dependent on the current time, or when the value is based on a randomly generated value. To handle the randomness in this case, we use the following approach:

Suppose that once the system executes action a from state S_n (line 14 in algorithm 1), it observes that the generated requests have a *partial* match with the next expected traffic. It means that two requests have the same number of parameters and match, if we ignore differences in values for some *parameters* of two requests. 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.

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.

(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*

Figure 3.2: (a),(b): Two HTTP requests that include a parameter with a changing value

Figure 3.2 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 Figure 3.2) with the 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.

3.3.5 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[3] 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 3.3.1). Timers are also one of the possibly actions in the current state, so the algorithm first detects them on the current state (line 12 in algorithm 1) and evaluates each timer (line 14 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 send from the server. The *proxy* in our general algorithm has to be changed to detect these server-initiated requests; when the proxy observers that the next expected traffic is server-initiated, it just *sends* these requests to the client.

3.4 Conclusion

In this chapter, we first presented a simple recursive method for reconstructing sessions of users. We then extended this method to be able to be useful in practice. The additions include: more efficient ordering of actions, concurrent trying of candidate actions, handling randomness in the generated requests, extraction of user-inputs submitted during the session, and finally detection of actions that include timers and server-initiated requests.

The proposed method solves the general session reconstruction problem for client/server applications, and is independent of any specific implementation. In the next Chapter, we present *D-ForenRIA* that realizes this improved algorithm in the context of RIAs.

[3] `https://www.websocket.org/aboutwebsocket.html`

Chapter 4

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

4.1 Introduction

In Chapter 3, 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 (Chapter 3 Section 3.3) in the context of RIAs and addresses several challenges mentioned in the previous chapter.

Our solution is implemented in a tool called *D-ForenRIA*. In this Chapter, we first present the evolution of *D-ForenRIA*. We then discuss the most important component of *D-ForenRIA* and description of the messages exchanged between components, followed by details of each component.

4.2 The Evolution of D-ForenRIA

The initial evaluation of the feasibility of solving the session reconstruction problem was done earlier in my research and was published in a tool called ForenRIA [9]. However, there

Table 4.1: Comparison between ForenRIA and D-ForenRIA

Feature	Tool	
	ForenRIA	D-ForenRIA
Distributed Architecture	×	✓
Standard User input support	✓	✓
Complex User Input support	×	✓
Handling Timers	×	✓
Handling Actions without Traffic	×	✓

are limitations in ForenRIA and the techniques provided in *D-ForenRIA* tried to address these limitations. To have a working solution, we designed *D-ForenRIA* from scratch and it has nothing in common with ForenRIA.

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 features required to reconstruct our test cases (Chapter 6). Therefore, we did not use ForenRIA as a baseline tool in our experiment. Table 4.1 summarizes the main differences between ForenRIA and *D-ForenRIA*.

4.3 Architecture of D-ForenRIA

Figure 4.1 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. The SR-Proxy performs the role of the original Web server, and responds to SR-Browsers' requests from the given input HTTP trace. Based on our session reconstruction algorithm, we infer which user-browser interactions are performed during the session.

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,

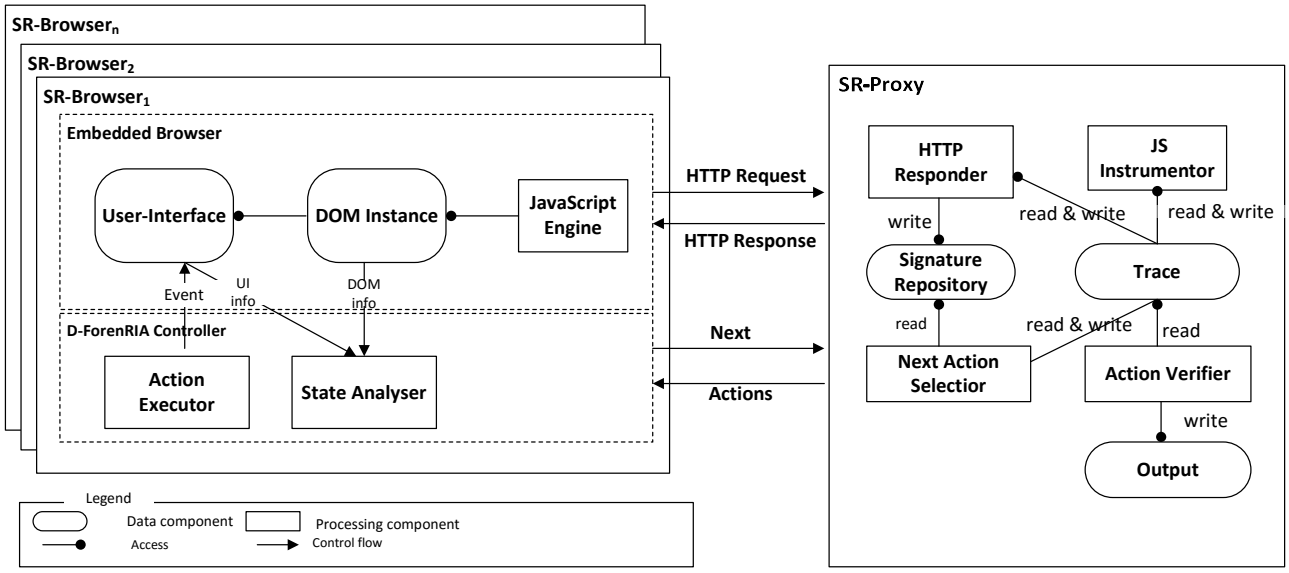


Figure 4.1: Architecture of *D-ForenRIA*

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.

4.3.1 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 Figure 4.2). We call this repetitive process *iteration*. Figure 4.2 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 3.3.1, 4.5).
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.

4.3.2 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.

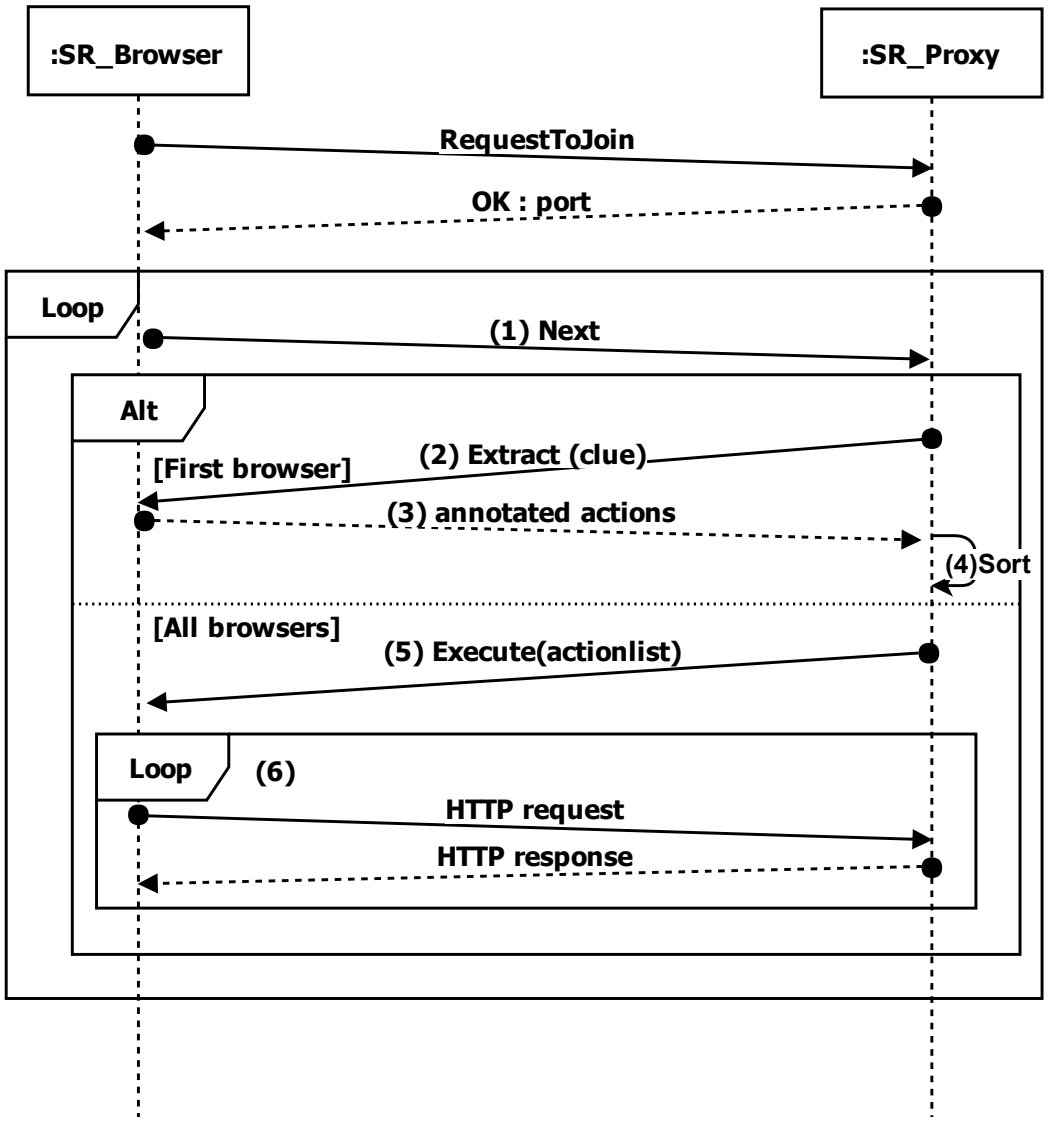


Figure 4.2: Sequence diagram of messages between an SR-Browser and the SR-Proxy

- 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 4.4, 4.6).
- 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.

4.3.3 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.

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

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
```

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.

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

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.ExtractNextCandidateAction(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
```

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, 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). The main steps of the session reconstruction procedure are explained below.

4.4 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 Figure 4.5, *FetchData(0)* is the event-handler for the *onclick* 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*,...).

```
1  var addEventListenerOrig = Element.prototype.addEventListener;
2  var EventListener=function(type,listener) {
3  notifyDynamicHandler(this, type, listener);
4  addEventListenerOrig.call(this, type,listener);
5  };
6  Element.prototype.addEventListener= EventListener;
```

Figure 4.3: Hijacking the built-in JavaScript AddEventListener function to detect dynamically assigned handlers.

2. *Dynamically assigned handlers*: in JavaScript, dynamically assigned handlers are set using the *addEventListener* function[1]. As shown in Figure 4.3, *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 [19] and is realized by the JavaScript instrumentor that injects the code shown in Figure 4.3 in the responses sent to SR-browsers from the SR-Proxy.

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"[2]. 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.

Identification of Actions: The session reconstruction tool should be able to differentiate between actions of each state; to this end, a unique identifier should be assigned to each candidate action. This identifier should be deterministic [26]. That is, given a state and an identifier we should be able to determine the corresponding action. This is

[1] `https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener`
[2] `http://javascript.info/tutorial/bubbling-and-capturing`

required because SR-Browsers may need to trigger a given action from a state at a later time; for example, when the SR-Proxy sends a sequence of action identifiers to transfer the SR-Browser to a state.

In addition, the identifier should identify instances of the same action in different states. This feature enables the session reconstruction tool to predict the behavior of an action, based on the other instances of the same action. Therefore, the identifier should be unique for an action across all the states. Moreover, these identifiers can be considered as one of the outputs of the session reconstruction tool. Therefore, the identifier should be descriptive and understandable by the user.

D-ForenRIA calculates the identifiers based on the parameters of the action. For click actions, we use the combination of the *XPath* + *innerHTML* of the clicked element. The XPath is complete and fully defined; to calculate the XPath of an element, the tool finds the path to the element from the root of the DOM. Then, for each node along the path it computes the *tagName + index of the node* in the child list, and concatenates these values. For example in Figure 4.4, the XPath of the element with id="p2" is *Body[2]/Span[2]*. This *XPath* uniquely identifies the element in the current state; the *innerHTML* used for two reasons: First, to differentiate between actions with the same *XPath* across two different states. Second, the *innerHTML* provides the textual representation of the element, that is usually readable for the user. For the user-input actions, we need to consider more parameters: For each input element, the identifier includes the *XPath* + *innerHTML* of the element, and also the value entered by the user. In addition, *D-ForenRIA* also adds the identifier for the action that submits the entered values to the server (usually a click action).

4.5 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 Chapter 6). *D-ForenRIA*, uses the signature-based ordering to order candidate actions at each state. As we discussed in Section 3.3, 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

```
1  <meta name="SSRG" content="Sample RIA">
2  <body onload = "attachHandler()">
3  <div style="visibility:hidden">SSRG 2016 </div>
4  <span>RIA Store:Choose one of the products</span>
5  <hr>
6  <span id='p1'> LG G4 </span>
7  <span id='p2'> iPhone SE </span>
8  <span id='p3' onclick="FetchData(2)"> OnePlus X </span>
9  <div id="container">--</div>
10 <hr>
11 <a href="about.php">About Us</a>
12 <a href="contactus.php">Contact Us</a>
13 <div id="news"></div>
14 <input type="text" id="srch" onkeypress='autocmpl(event);'>
15 <span>Advanced Search</span>
16 </body>
```

Figure 4.4: A simple DOM instance

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 4.4), or have tags with which users usually do not interact (e.g. *script*, *head*). For example, in the DOM of Figure 4.4, 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.

4.6 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.

```
1  var reqs =
2  [ ["ra1.json", "ra2.json", "ra3.json"],
3  ["rb1.json","rb2.json"],  ["rc1.json","rc2.json","rc3.json"]];
4  //Attch handlers
5  function attachHandler(){
6  $("#p1").on("click", function(){ FetchData(0); });
7  document.getElementById("p2").onclick =
8    function(){ FetchData(1);}
9  }
10 //Fetching data
11 function FetchData(id){
12 $('#container').empty();
13 for (res of reqs[id]) {
14 $.get( res, function( data ,status ) {
15 $('#container').append(data); }, 'text'); }
16 }
```

Figure 4.5: A simple JavaScript code snippet

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 *WebSockets* 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.

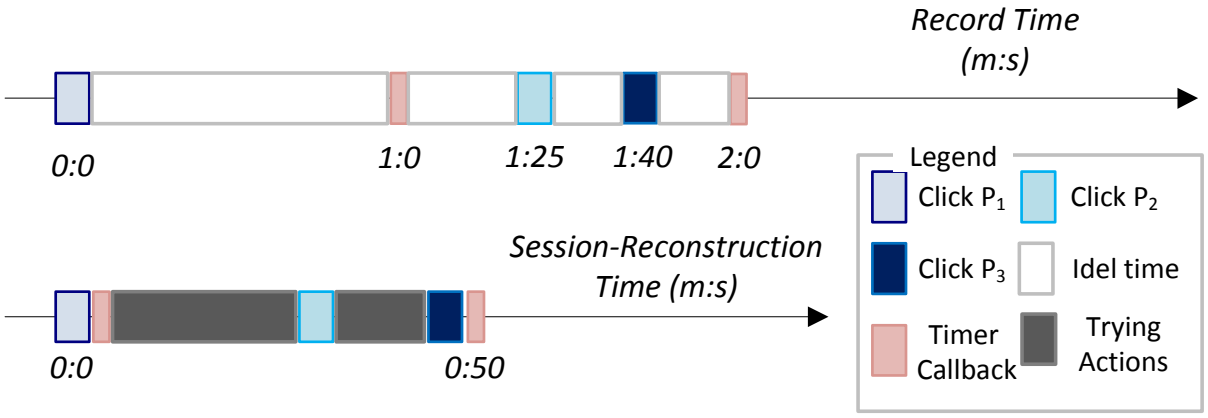


Figure 4.6: A session with a timer: time flows from left to right along the axis (top) at Recording, and (bottom) at Session-Reconstruction time.

```
1  function updatenews() {
2  $.get( 'latestnews.json',
3  function( data ,status ) {
4  $('#news').html(data); }, 'text');
5  }
6  setInterval(updatenews, 60000);
```

Figure 4.7: A timer registered using setInerval to fetch the latest news.

***Example**:* Assume that our example in Figure 1.5 also has a timer which registers itself using a setInterval call to fetch the latest news from the server (Figure 4.7). 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 4.6 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.

4.7 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 [3], 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, when an SR-Browser is being asked to extract actions, it detects all form elements on the current DOM as candidate actions (Section 4.4). 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 3.3. 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 text-box 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.

4.8 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.

4.9 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 [60].

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 [59]. 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.

4.10 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

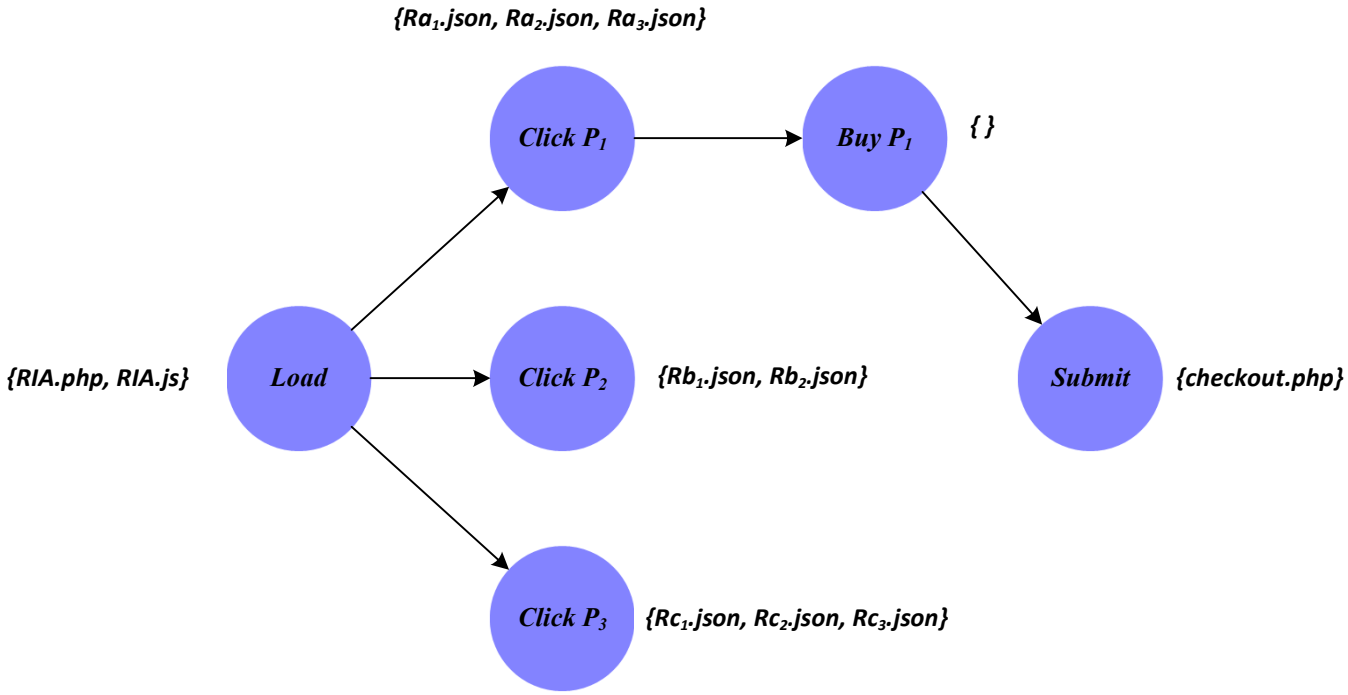


Figure 4.8: Portion of the action graph of Figure 1.5.

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 [50]).

We acknowledge that there are languages that provide a formal description of actions and their behavior in a RIA. Languages such as CCS[3] [53] and pi-calculus [68] would be used for this purpose. As an example, an alternative to the proposed action-graph would be using pi-calculus to express states (as processes) and actions. However, in this research we did not need that level of formalism and we used a simple semi-formal graph-based notation.

[3]Calculus of Communicating Systems

Example: Consider the example in Figure 1.5. In this example, we have a node for each tried action during the reconstruction. Figure 4.8 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.

4.11 Choosing the Next Candidate Action (SR-Proxy)

As we mentioned in the SR-Proxy's algorithm (Algorithm 3), once the SR-Proxy receives the pool of candidate actions, it sorts them from the most promising to the least likely (line 15). Then, the "Next Action Selector" component (Figure 4.1), selects one of these actions each time and assigns it to one of the SR-Browsers. Having discussed the details of *D-ForenRIA*, we can summarize how the *ExtractNextCandidateAction* function (line 18 of Algorithm 3) chooses the next candidate action.

Function ExtractNextCandidateAction(*SR-B*, *c*, *t*)

Input: An SR-Browser *SR-B*,Traces *t*, Pool of Actions *c*
Output: The next action to be tried *e*

```
1: if UserInputCandidate(SR-B.lastRequests, t) then
2:    e ← ExtractUserInputAction(SR-B.lastAssignedAction, t)
3: else if RandomCandidate(SR-B.lastRequests, t) then
4:    e ← SR-B.lastAssignedAction
5: else
6:    if c = {} then
7:       c ← FindPathsfromActionGraph()
8:    end if
9:    e ← EnqueueACandidateAction(c)
10: end if
11: return e
```

This function is being executed when the SR-Proxy assigns an action to an SR-Browser. If the last action performed by the SR-Browser is detected as a candidate user-input action, the algorithm extracts the actual input values and asks the SR-Browser to repeat the action

using the actual values (line 1-2). In the case of "random values", *D-ForenRIA* asks the SR-Browser to repeat the last assigned action (line 4). Otherwise, the algorithm assigns the next candidate action from the pool (line 9). Once there is no other candidate action in the pool of actions, the algorithm tries actions extracted from the action graph (line 7).

4.12 Conclusion

In this chapter a new session reconstruction tool, called *D-ForenRIA*, was introduced. *D-ForenRIA* realizes the general session reconstruction algorithm, which was explained in Chapter 3, in the context of RIAs. The system is composed of a central coordinator (called SR-Proxy) and a set of browsers (called SR-Browsers). SR-Browsers try different candidate actions in parallel and the SR-Proxy is responsible for responding to the generated HTTP requests and validating the tried actions. We also described how we used the capabilities of JavaScript and DOM to implement different components of *D-ForenRIA* (such as extracting/executing user actions, and reloading the browser's previous state).

In the next chapter, we will explain the idea of similarity-based ordering and in Chapter 6 we present some experiments about the performance of *D-ForenRIA* compared to other methods.

Chapter 5

Similarity-based Ordering of Candidate Actions

5.1 Introduction

Our proposed technique to sort candidate actions (Section 3.3.1) relies on the history of the previous executions of an action. Therefore, if the tool has not tried an action previously it becomes ineffective and cannot decide whether the action is promising or not. Consequently, when there is a large set of new elements on a page, *D-ForenRIA* needs to test many actions to find the correct action; however, we believe that each RIA is usually composed of a set of similar actions (such as a list of products in an e-commerce website, or a list of comments on a blog) that generate rather similar HTTP requests.

We propose to use pre-existing knowledge we have about similar actions to decide upon actions that we have not tried yet. This idea can potentially save a considerable amount of time during the reconstruction. The algorithm first tries to assign a score to an action using its signature (if we have already tried the action, and have its signature), otherwise the algorithm will try to find a similar action that has already been tried, and uses that similar action's signature to predict the score of the action.

The idea of detecting similar events on the DOM, has also been discussed in previous research (e.g. [8, 31, 29, 55]). In this chapter, we extend the ideas presented in [55] which has been used to diversify the crawling results, and adapt it to the session reconstruction

problem. However, there are significant differences between the proposed method and the method presented in [55]:

- *Goal of the similarity detection*: In our method, the goal of similarity-based ordering is to guide the session reconstruction tool to try the more promising candidate actions first. In other words, the technique *diversifies* the set of candidate actions when the set of similar actions does not look promising at the moment. The algorithm also *intensifies* by trying similar actions when it observes that the similar actions generate requests that are similar to the next expected requests in the log. On the other hand, the goal in [55] is finding similar actions on each page, and diversifying the crawling results by first exploring dissimilar actions.

- *Similarity measures*: In [55] it is assumed that we have access to the website during the crawl. Therefore the similarity is defined based on the features of the DOM after executing an action. On the other hand, during session reconstruction we do not have access to the server. Therefore, trying an action may generate requests that do not exist in the given log and eventually would not generate a valid DOM. Consequently, we cannot find similar actions based on the resulting DOM after performing an action. Therefore, we introduced a new similarity measure which is based on similarity of the generated requests.

Example: To better explain the idea of ordering candidate actions based on the generated HTTP requests, consider a state of a sample RIA presented in Figure 5.1. In this state of the RIA, users can see a list of products, buy each product using the *buy* button or *like* the product. Here we observe similar groups of HTTP requests. For example, by clicking on any *like* button, a single request to the same resource is made. The only difference between requests is the value of a query string parameter (i.e., pid). We can also see a similar pattern between the generated requests after clicking on any *buy* link. Each click on a *buy* link requests a resource called "buy" and an image. The names of the requested resources are similar, but there are subtle differences. Suppose that *D-ForenRIA* already knows the signature of the first *buy* link (of product P1) and also the signature of the first *like* button (for P1) and the next expected traffic is {*buy?pid=3*, *img3.png*}. Here *D-ForenRIA* can compare the signature of P1's *buy* link and P1's *like* button with

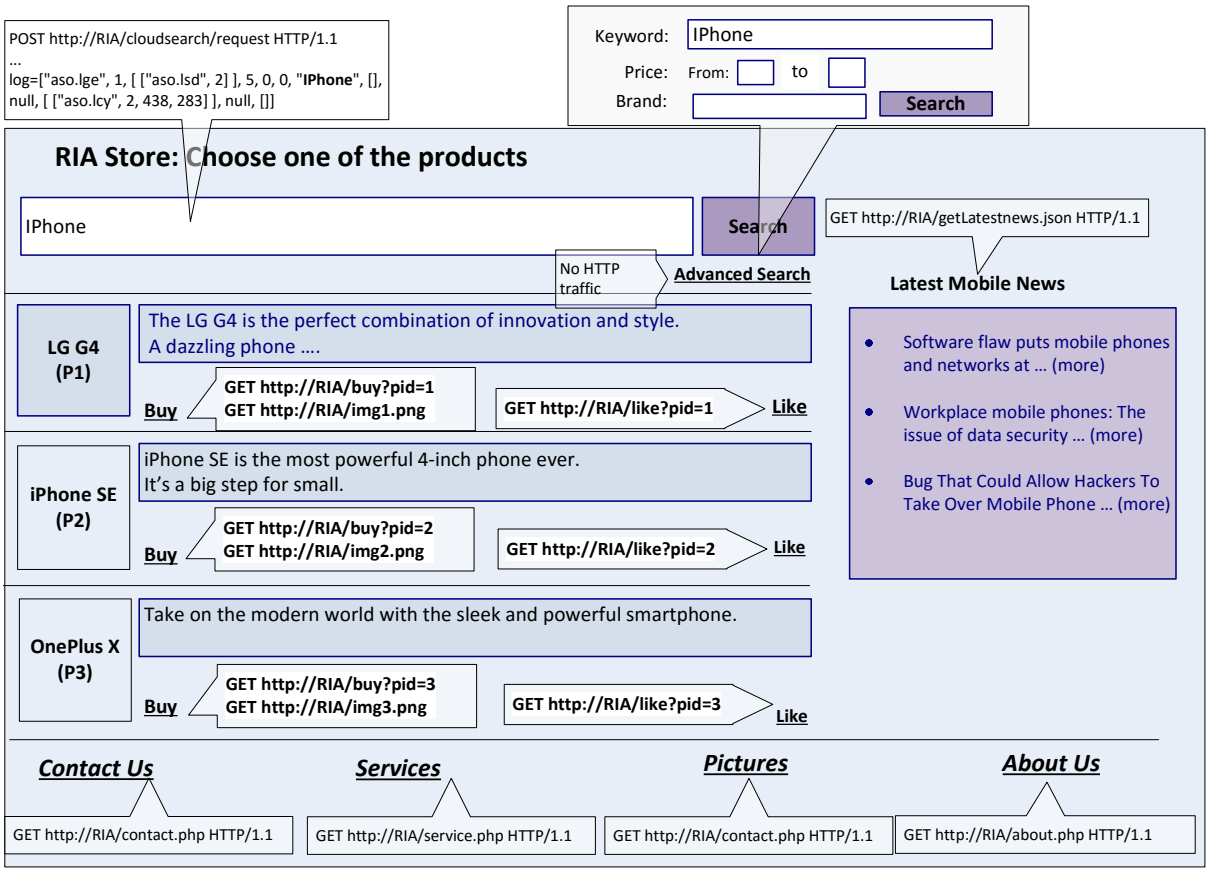


Figure 5.1: state of the simple RIA and the generated requests after clicking on DOM elements

the next expected traffic, and see that the signature of the *buy* link is quite similar to the next expected requests. Based on this observation, *D-ForenRIA* can increase the priority of clicking on *buy* links and also decrease the priority of clicking on *like* buttons.

5.2 Solution Overview

In similarity based ordering, the priority value of an action (which does not have a signature), is *predicted*, from a *similar* action (that has a signature). We use the term "Group" to refer to the set of similar actions (e.g. the set of *like* buttons in Figure 5.1 form a group, and set of *buy* buttons form another group). The SR-Proxy keeps track of the groups and updates them according to the generated traffic after performing an action from that group.

Later, when the algorithm needs to assign a priority-value to an action without a signature, it uses the signature of a similar action (in the same group) to predict the priority-value of the action. In the following section we explain how the SR-Proxy algorithm forms groups and calculates similarity-based score.

5.3 Solution Elaboration

Forming Groups: We assume that there is a heuristic function m that calculates an ID for each action. This function assigns the same value to all members of a group:

$$\forall\, g \in G, \quad a_1, a_2 \in g \Leftrightarrow m(a_1) = m(a_2) \tag{5.1}$$

In other words, function m partitions the set of actions in the current state. It is notable that function m should not dependent on executing events, since the function is being called when the SR-Proxy want to decide the execution order of candidate actions. In *D-ForenRIA*, we use a simple definition of m: two actions are the member of the same group if their corresponding HTML elements have the same "TagPath". To calculate the *TagPath* of an element, the algorithm finds the path to the element from the root of the DOM and concatenates the tag of the nodes in the path. For example in the DOM of Figure 5.2, we observe several groups: For example there is a group for *like* buttons, which corresponds to the TagPath *Body/Table/TR/TD/Div*, and another group for the list of products (which has the TagPath of *Body/Table/TR/TD/Span*). We also observe a group of the menu items in the bottom of the page ("Contact-us", "Services", "About Us" etc.). All members of this group have the same TagPath of *Body/A*.

Labelling Groups: The "similarity-based" ordering is based on "induction". We infer a general behavior (that is generating similar requests after execution) by observing the behavior of particular instances of the group. Therefore we expect that the execution of all group members generate rather similar requests. However, this assumption may not hold in practice. For example, the menu items in the Figure 5.1 generate totally different requests. Therefore, we need to verify the groups to check whether they actually generate similar requests or not. To this end, we assign a label to each group. This label can have three values : N/A, "similar", "dissimilar".

```
1  <body>
2    <Div style="visibility:hidden">SSRG 2016 </Div>
3    <span>RIA Store: Choose one of the products</span><hr>
4    <table>
5    <tr>
6    <td>
7    <span id='p1' >LG G4</span>
8    </td>
9    <td>
10   <p>The LG G4 is...</p>
11   <div>Buy</div>
12   </td>
13   </tr>
14   <tr>
15     <td>
16     <span id='p2' >iPhone SE</span>
17     </td>
18     <td>
19     <p>iPhone SE is the most powerful...</p>
20     <div>Buy</div>
21     </td>
22   </tr>
23   <tr>
24     <td>
25     <span id='p3' >OnePlus X</span>
26     </td>
27     <td>
28     <p>Take on the modern world...</p>
29     <div>Buy</div>
30     </td>
31   </tr>
32   </table>
33   <hr>
34   <a href="contactus.php">Contact Us</a>
35   <a href="services.php">Services</a>
36   <a href="services.php">Pictures</a>
37   <a href="about.php">About Us</a>
38   <body>
```

Figure 5.2: The DOM of a simple RIA

To assign the label we use the input value b (which is a threshold), and the similarity function s. The label is calculated using the following formula:

$$label(g) = \begin{cases} \text{“}similar\text{”} & \forall\, a_1, a_2 \in g \;\; s(a_1, a_2)) \wedge |\{a \in g | signature(a) \neq unknwon\}| \geq b \\ \text{“}dissimilar\text{”} & \exists\, a_1, a_2 \in g \;\; \neg s(a_1, a_2) \\ \text{“}N/A\text{”} & otherwise \end{cases} \tag{5.2}$$

The label of a group is based on the signature of the group members. Therefore, when the algorithm has not executed any action of a group, the label is “N/A”. To verify that a group is composed of actions that generate similar requests, the algorithm needs to execute at least b members of a group. The value of b is given as an input to the algorithm. The higher values of b makes the algorithm more conservative, since the algorithm needs to execute more actions to decide about the label of a group. We assume that there is a heuristic function s, which checks whether two sets of requests are similar or not (the similarity is above a minimum threshold). We will provide the details of s function in the next section. The label of a group becomes “dissimilar” if the SR-Proxy observes that two actions of that group generate dissimilar requests.

The following pseudo-codes summarize how the algorithm updates the list of groups and calculates the similarity-based score:

Procedure updateGroups(*a*, *groups*, *signatures*)
Input: An action *a*, list of all groups *groups*, signatures of all actions *signatures*

```
1: g ← m(a);
2: if (!groups.contains(g)) then
3:    groups.add(g))
4:    groups.setLabel(g, “N/A”);
5: else
6:    if (!groups.similarReqs(g, signatures)) then
7:       groups.setLabel(g, “dissimilar”);
8:    else if (groups.getSize(g) ≥ b) then
9:       groups.setLabel(g, “similar”);
10:   end if
11: end if
```

Function similarityBasedScore(a, *groups*, *signatures*)
Input: An action a, list of all groups *groups*, signatures of all actions *signatures*
Output: The predicted score of a based on similar actions

```
1: g ← m(a);
2: if groups.contains(g) and groups.getLabel(g) == "similar" and signatures.get(a) ==
   "N/A" then
3:    a' ← groups.actWithSignature(g);
4:    return signatureScore(a');
5: else
6:    return N/A
7: end if
```

The *updateGroups* procedure is responsible for updating the list of groups. This function is called after a candidate action is executed. The function first calculates the group of the given action (line 1). If it was the first executed action of the group, it creates a new group for a, and labels the group as "N/A" (lines 2-4). If we observe any dissimilarity between members of the group, we label the group as "dissimilar" (line 7). In addition, if we observe that b members of a group has generated similar requests, the group is labelled as "similar" (line 9).

The *similarityBasedScore* function calculates the score of an action a based on similar actions. The function first calculates the group of a (line 1). The algorithm then checks whether we have previously tried some actions from that group, and if those actions have produced similar sets of requests (line 2). If such a group is found, it uses the signature of an action from that group ,a', to predict the score of a (since a group labelled as "similar" contains at least b members with similar signature, we are sure that a' exists). The output of the function is "N/A" if we could not predict the score of a using a similar group (line 6).

The group-based ordering should be reapplied after executing each action in the same state. The reason for this is that after performing each action, the list of groups (and their labels) is updated, and this information can be potentially useful to find the current action. Otherwise, the efficiency of the algorithm decreases. For example, assume that there is a set of similar actions on the page, and the user has not clicked on any of them. In this case, after trying an adequate number of these similar actions, there is no need to continue trying these actions in the current state.

Similarity between HTTP requests: As we discussed in the previous section, members of a "similar" group should generate a set of similar requests. Here we present the details of the heuristic function s that determines whether two sets of requests are similar or not. Function s is defined as follows:

$$s(a_1, a_2) \Leftrightarrow \frac{d_1(r_1, r_2) + d_2(r_1, r_2)}{2} \leq \alpha \tag{5.3}$$

where a_1 and a_2 are two actions and r_1 and r_2 are their associated signatures respectively. d_1 and d_2 compare the signatures of the two actions from different perspectives; d_1 compares two sets of requests based on some *statistical* features, while d_2 compares two sets of requests based on theirs *structural* features. Each function assigns a distance measure within [0, 1] to the given sets of requests. If the combined distance of d_1 and d_2 are less than a threshold α, the signatures of actions are considered similar (we use the value of 0.1 for α). d_1 and d_2 functions are defined as follows [63]:

- Statistical distance (d_1): To calculate the statistical distance of two request sets, $d_1(r_1, r_2)$, we calculate the following statistical features for each request sets:
  - the number of GET/POST requests
  - the average length of the URLs
  - the average number of parameters in the requests
  - the average amount of data sent by POST requests

  We represent statistical features of r_1 and r_2 as a vector. We then compute the normalized Euclidean distance[1] between these two vectors[2].

- Structural distance (d_2): To measure the structural distance between two HTTP requests p , q their distance is measured based on the following structural features (Figure 5.3):

[1]The Euclidean distance between two vectors p=$(p_1, p_2, .., p_n)$ and q=$(q_1, q_2, ..., q_n)$ is defined as $\sum_{i=1}^{n} \sqrt{(p_i - q_i)^2}$

[2]Since the range of different features are quite different, two features f_1 and f_2 are compared using the formula $\frac{|f_1 - f_2|}{max(f_1, f_2)}$

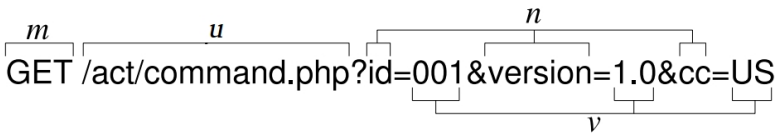


Figure 5.3: Structure of an HTTP request m=Method; u=Page; n=Parameter Names; v=Parameter Values [63]

- m represents the request method (e.g., GET, POST, HEAD, etc.). We define a distance function $d_m(p, q)$ that is 0 if the request method of p and q are equal and is 1 otherwise.
- u refers to the requested page (the first part of URL that includes path and the page but does not include parameters). We define $d_u(p, q)$ which measures the normalized Levenshtein distance[3] between the requested page in each request.
- n represents the set of parameter names (i.e., n = {id,version,cc} in the example of Figure 5.3). We define $d_n(p, q)$ that is calculated as the Jaccard distance[4] between set of parameter names in two requests.
- v is the set of parameter values. We define $d_v(p, q)$ to be equal to the normalized Levenshtein distance between between strings obtained by concatenating all parameter values in each request(e.g., 001;1.0;US).

The following formula calculates the overall distance between two HTTP requests:

$$d_2(p, q) = w_m.d_m(p, q) + w_u.d_u(p, q) + w_n.d_n(p, q) + w_v.d_v(p, q) \tag{5.4}$$

where w_m, w_p, w_n and w_v re predefined weights for different aspects of the distance between two features (we use $w_m = 0.45$, $w_u = 0.35$, $w_n = 0.15$ and $w_v = 0.05$ as suggested in [63]).

To calculate the structural distance between the two request sets, for each request in one

[3]The normalized Levenshtein distance between two strings w_1 and w_2 is defined as $NormLevenshtein(w_1, w_2) = \frac{Levenshtein(w_1,w_2))}{max(length(w_1),length(w_2)))}$ where $Levenshtein(w_1, w_2)$ represents the minimum edit distance (number of characters that needs to be added, removed or changed) to convert w_1 to w_2 and $NormLevenshtein(w_1, w_2) \in [0, 1]$.

[4]The Jaccard distance between two sets A and B, measures the dissimilarity between two sets and defined as $J(A, B) = 1 - \frac{A \cap B}{A \cup B}$

set the request with the least distance in the other set is found; then, the average of these distances are considered as the structural distance between two sets.

Example: Assume that a RIA states includes three groups of similar actions namely A1=$\{a_1, a_2, ..., a_{20}\}$, A2=$\{a_{21}, a_{22}, ..., a_{30}\}$, A3 = $\{a_{31}, a_{32}, ..., a_{35}\}$, and the user has performed the sequence of actions $< a_{35}, a_5, a_{25} >$; suppose that the similarity-based ordering is disabled and the list of actions are evaluated in $a_1, a_2, ..., a_{35}$ order. In this case, *D-ForenRIA* needs to evaluate all 35 elements to find the first action. However, since the algorithm learns the signature of a_1, a_2, these two actions are selected immediately. Therefore when the similarity-ordering is disabled, the algorithm needs to execute 37 actions to reconstruct the session.

On the other hand, when the similarity-based ordering is enabled the algorithm initially just evaluates two actions from A1 and A2 groups (b = 2), and does not further continue searching for a_{35} in these two groups. Then, the algorithm examines all members of A3 to find the first action, a_{35}. Therefore $9 = 2 + 2 + 5$ actions are evaluated to find the first user interaction. During finding the first action, the algorithm forms and labels the three groups; therefore, to find the second action it just tries three more actions in A2 (since a_1, a_2 are already tested). The last actions, a_{25}, is also selected by limiting the search to A2 group, and can be found after trying three candidate actions.

As we can see in this example, the similarity-based ordering can sometimes increase the cost of finding a particular action (for example a_5 can be found immediately when the similarity detection is off, but requires 3 more actions to be evaluated when the similarity based detection is enabled). However, the cost-reduction is significant in other cases (for example, the cost to discover the first action decreases from 35 to 9 by applying the similarity-based ordering).
Therefore, we expect that when the similarity-based ordering is enabled, the algorithm would not be trapped in a particular region of the page, and continues the search in more promising regions of the page, which eventually reduces the total cost of the reconstruction. The results of our experiments confirmed this expectation.

5.4 Discussion

In this Chapter, we introduced a framework that given a similarity measure can direct the session reconstruction to more promising regions of the page. The contribution of our work is not the similarity measure, but rather a method that uses a similarity measure given as an input to our algorithm, for a more efficient session reconstruction.

We used a similarity measure which uses statistical/structural similarity of HTTP requests; this similarity measure has been used by other researchers to compare HTTP-requests generated by malware [63]. However, the application domain of our similarity-based ordering is totally different; here, we compare two sets of HTTP-requests that correspond to the traffic generated by performing two different actions.

If we had used a different measure, we might had different results; using other HTTP similarity measures and observing their impact on the efficiency of similarity-based ordering, would be an interesting future work.

5.5 Conclusion

In this Chapter we have introduced the idea of "similarity-based" ordering of candidate actions. This idea is based on grouping similar actions on each page. Each group is labelled as "similar" if the previously tried actions have generated similar sets of requests. We have defined a heuristic function to form the groups on each page, and another function to detect the similarity between two sets of HTTP requests. The session reconstruction algorithm first tries to assign a priority-value to an action based on its observed signature. However, if an action has not been tried yet, the algorithm uses the signature of similar actions to predict the priority of the action. In the next Chapter, we present some experiments about the efficiency of the similarity-based ordering and other components of *D-ForenRIA*.

Chapter 6

Experimental Results

6.1 Introduction

In this chapter, we present our experimental results regarding different aspects of our session reconstruction method. The remainder of this chapter is organized as follows:

- In Section 6.2 we describe our test RIAs.
- The comparison criteria that are used to compared the performance of session reconstruction algorithms are explained in Section 6.3.
- Section 6.4 describes the experimental setup.
- The results for different experiments are presented is Section 6.5:
  - Section 6.5.1 includes the results of comparing the performance of *D-ForenRIA* against a basic session reconstruction algorithm.
  - Experimental results on the effect of the distributed architecture on the time required to reconstruct a session are presented in Section 6.5.2.
  - Section 6.5.3 studies the performance of different techniques for ordering candidate actions on each state.
  - In Section 6.5.4, we discuss the log storage requirements for *D-ForenRIA*.
  - Section 6.5.5 includes the results of the "similarity-detection" technique.

Finally, in Section 6.6 we discuss the results obtained during these experiments. Our experimental data along with the videos are available for download[1].

6.2 Test Applications

In this thesis, we limit our test-cases to RIAs. The reason to focus on RIAs is that other tools can already perform user-interaction reconstruction of non-AJAX Web applications (e.g. [58]). We used six target applications with different technologies and from different domains, to measure the performance of our session reconstruction tool. The test applications are explained in the following:

6.2.1 Elfinder

Elfinder (Figure 6.1) is a Web-based open-source file manager. The user can list the files/-folders on the server, and perform operations such as copying/deleting files and searching in a directory. This website is written in JavaScript using jQuery and jQuery UI. All of the generated HTTP requests contain a random parameter (that is based on the current time). *Elfinder* also does not use HTML forms, and has its own way to submit user-input values.

6.2.2 AltoroMutual

AltoroMutual (Figure 6.2) is an Ajaxified version of the 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. The site includes HTML form based user-input actions (such as login, search). There is no randomness in the generated sequence of the requests.

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

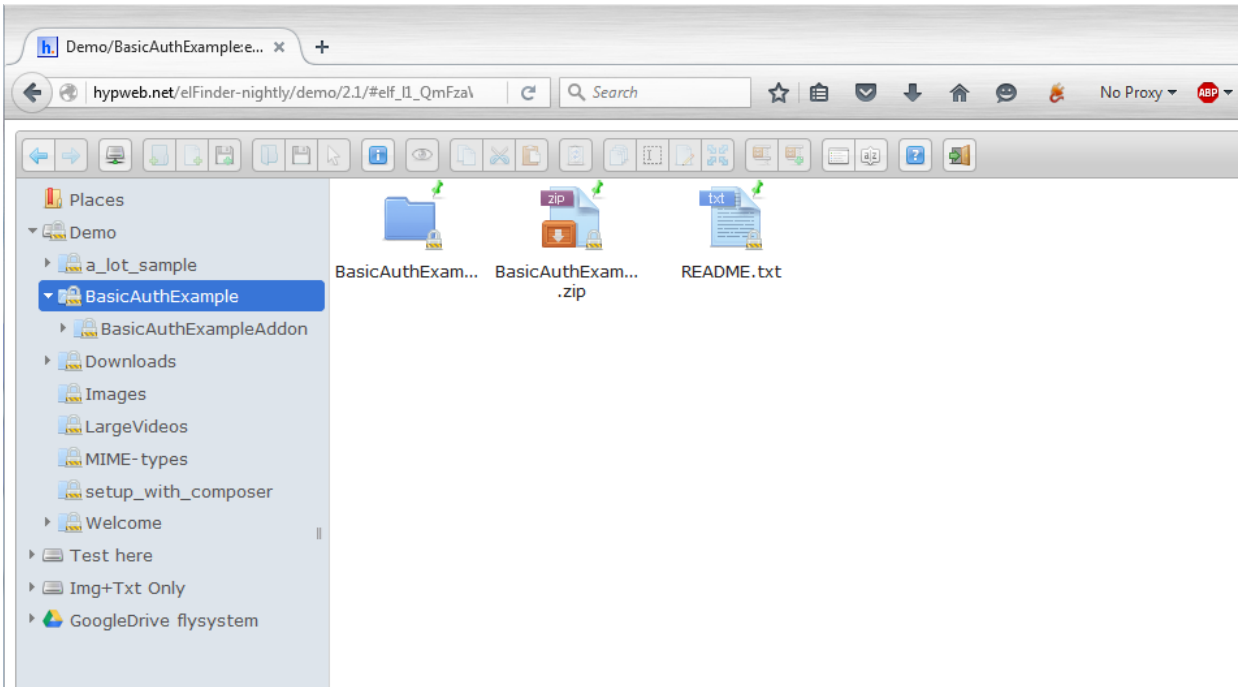


Figure 6.1: A state of Elfinder

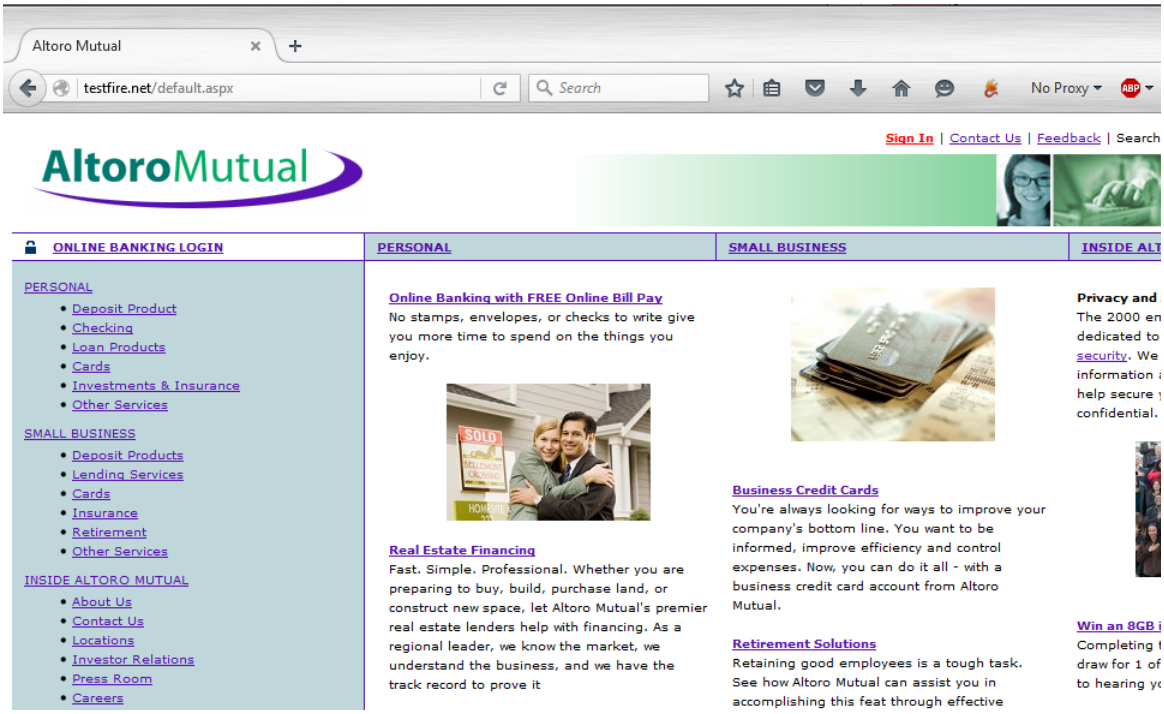


Figure 6.2: A state of AltoroMutual

6.2.3 Engage

Engage (Figure 6.3) is a Web-based goal setting and performance management application built using *Google Web toolkit*. The users can define projects and assign tasks to team members. In addition, users can communicate regarding each task, and use several dashboards to keep track of the progress of each project.

This website has many advanced features that made the reconstruction challenging: The first feature was that the generated traffic after performing an action depends on the user's browser type; therefore, the session reconstruction tool needs to use the same browser as the user's browser. Moreover, the site does not use HTML forms, and uses JSON to submit the input-data to the server.

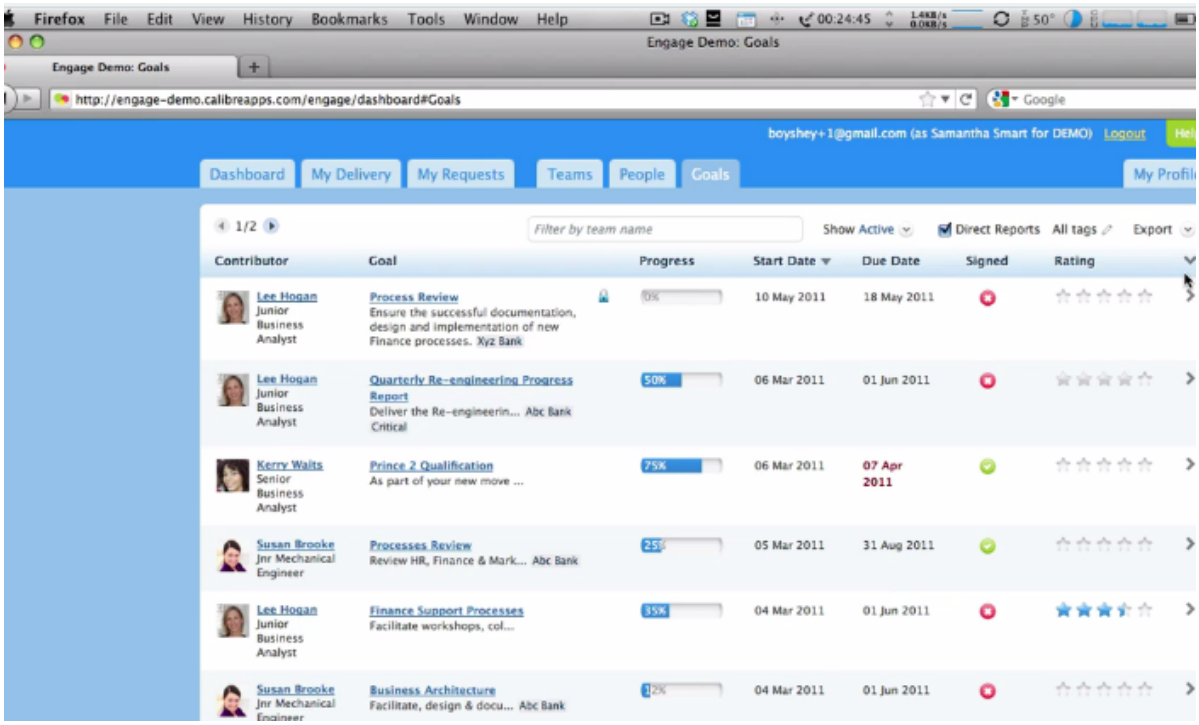


Figure 6.3: A state of Engage

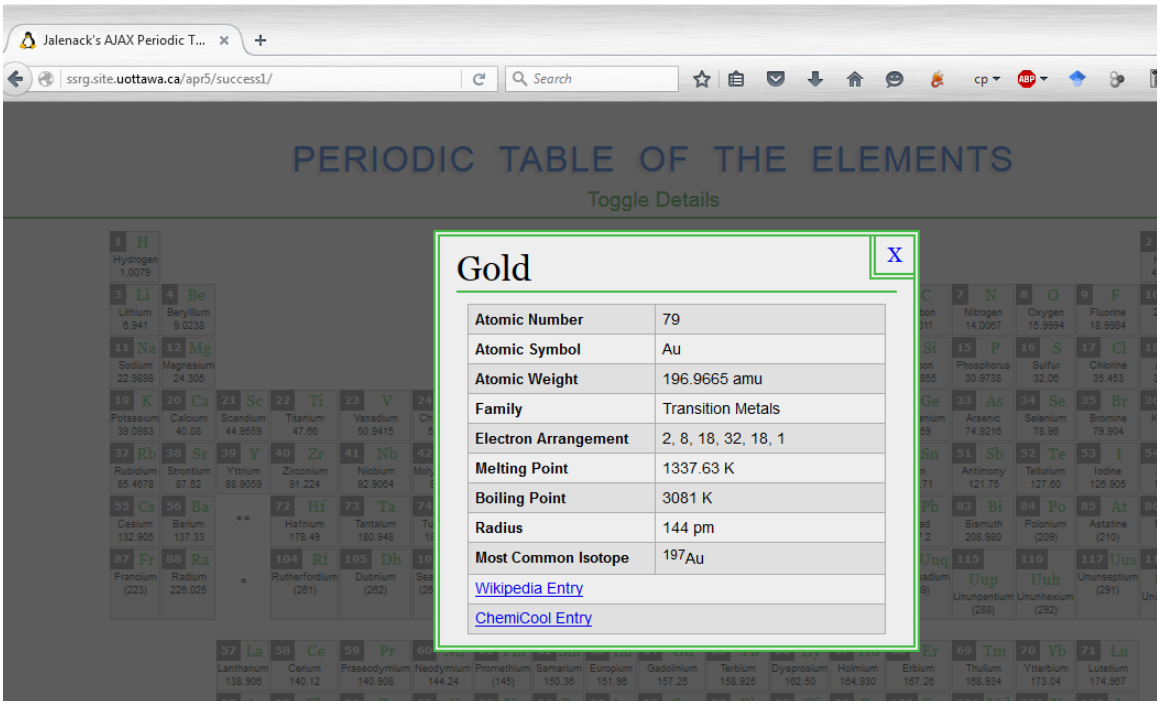


Figure 6.4: A state of PeriodicTable

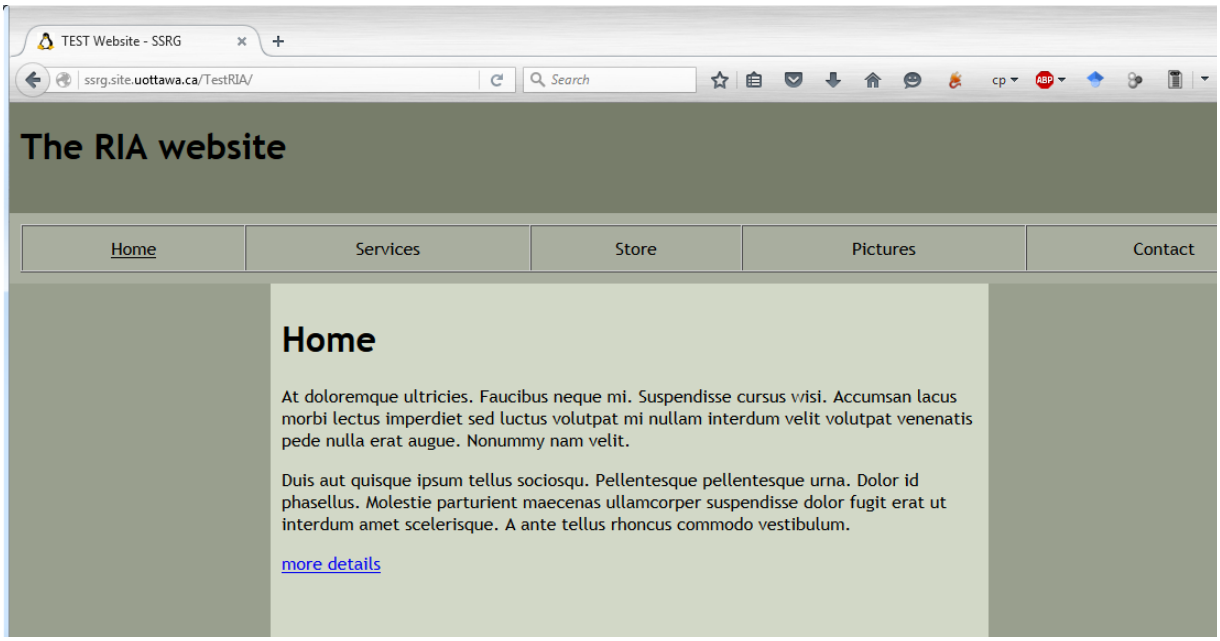


Figure 6.5: The initial state of TestRIA

6.2.4 PeriodicTable

PeriodicTable (Figure 6.4) is a fully AJAX-based periodic table. The user can select one of the elements of the periodic table, and see more detailed information in a new window. Also, there is an event at the top of each page (Toggle Details), which changes the style of the periodic table. This application is implemented in PHP and JavaScript.

6.2.5 TestRIA

TestRIA (Figure 6.5) is a website developed by our team which represents a typical company website (personal homepage). The user can navigate between different sections of the site that are home, services store, pictures and contact-us. Although it is a very simple RIA, it presents a pure RIA with a single URL. This example incorporated many features such as, asynchronous calls to update the DOM, and actions that do not generate any HTTP traffic. The previous session reconstruction methods cannot handle even this toy RIA.

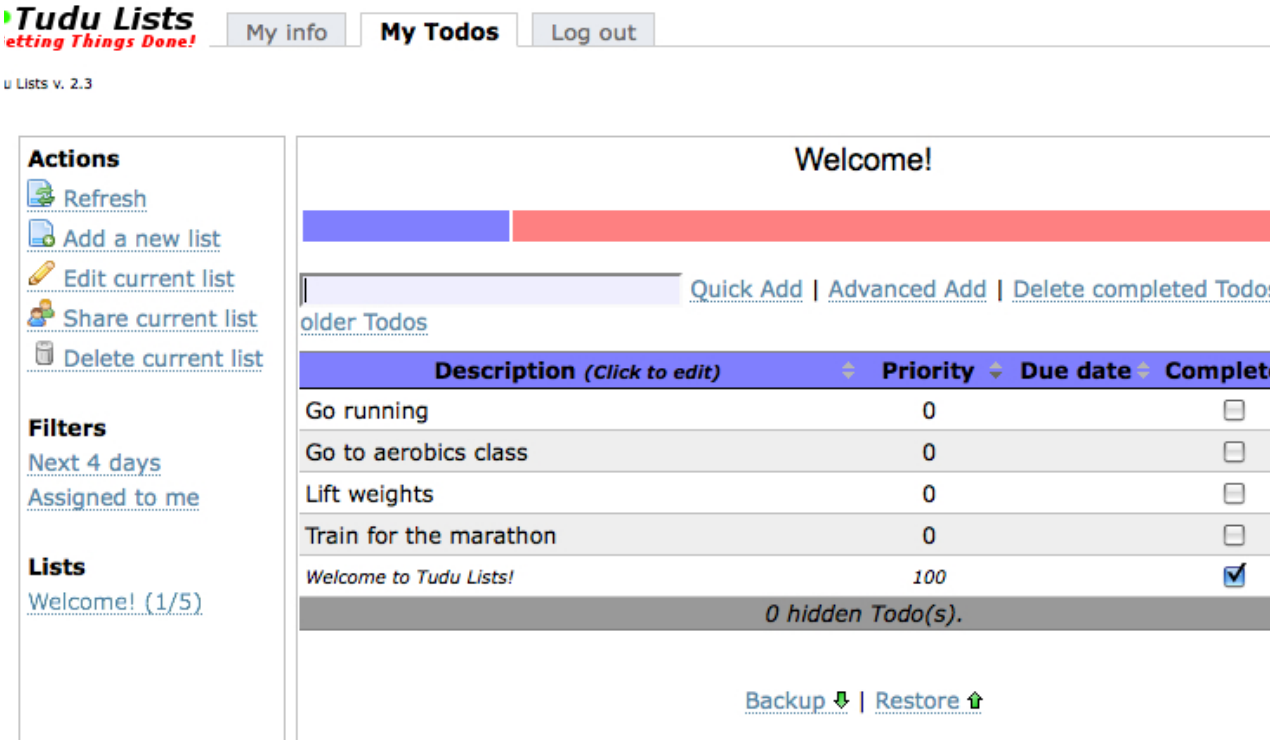


Figure 6.6: The initial state of TuDu Lists

6.2.6 Tudu Lists

Tudu Lists (Figure 6.6) is an open source project management tool, built using the Spring framework, and DWR (to handle AJAX). In *TuDu Lists*, users can create to-do lists, edit them, and share lists with other users. In addition, the users can assign tasks of each list to different users.

Reconstruction of the users' sessions with this RIA was not an easy task; the site has a large number of clickables in each state, random parameters in each generated HTTP request, and actions that do not generate any HTTP traffic. Moreover, *TuDu Lists* uses an advanced method to submit user inputs (based on remote procedure calling).

6.3 Measuring Efficiency

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:

$$n_e + \sum_{i=1}^{n_r} c(r_i) \tag{6.1}$$

where n_e is the number of actions in the user's session, and there are n_r resets (see Section 4.9) 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, 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.

6.3.1 Strategies Used for Comparison

We compare the session reconstruction algorithms presented in this thesis, with the following strategies:

- **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 4.5. To the best of our 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.

6.4 Experimental Setup

Experiments are performed on Linux-based computers with an Intel® Core™2 CPU at 3GHz and 3GB of RAM on a 100Mbps LAN. To implement the *D-ForenRIA* SR-Browsers, we used Selenium. *D-ForenRIA*'s SR-Proxy is implemented as a Java application. SR-Browsers built using Selenium 2.46.0. We have successfully tested *D-ForenRIA* using FireFox, Chrome and PhantomJS on Linux Ubuntu and Mac OS X 10.9. During session reconstruction, while we need to restore a previous state, we use Clear Cache 2.0.1.1 FireFox add-on[2] to clear the cache. We have implemented a similar plug-in for Chrome

[2] `https://addons.mozilla.org/en-US/firefox/addon/clearcache/`

Table 6.1: Subject applications and characteristics of the recorded user-sessions

Application	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/

as well[3]. For each test application, we recorded the full HTTP traffic of user interactions with the application using *Fiddler*[4]. Table 6.1 presents characteristics of our test-cases.

To measure the impact of the distributed architecture, 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 scalability of the system. To study the impact of the candidate ordering mechanisms, we ran *D-ForenRIA* using a single browser and measure how effective is applying each of the element/signature ordering. We also report the storage requirements for each action in the compressed format and the effect of pruning multimedia resources from traces. In all experiments, the similarity-based detection is disabled; we will present the results of this technique in section 6.5.5.

6.5 Experimental Results

6.5.1 Efficiency of D-ForenRIA

Table 6.2 presents the time and cost of reconstruction of full sessions using *D-ForenRIA*, and the basic solution. In this experiment we use just a single SR-Browser. We report

[3]This plug-in has been developed by *Bowen Cheng* and *Alexander Peter Charles Clarke*
[4]`http://www.telerik.com/fiddler`

Table 6.2: Time and cost of reconstruction using *D-ForenRIA*, the basic solution, and Min-Time

Application	D-ForenRIA		Basic Solution		Min-Time	
	#Events	Time (H:m:s)	#Events	Time (H:m:s)	#Events	Time (H:m:s)
C1	183	0:02:44	102933	09:51:26	150	00:02:21
C2	52	0:02:25	34505	04:31:57	50	00:02:06
C3	1325	0:04:22	308548	19:28:48	45	00:01:12
C4	3506	0:19:47	21518	02:12:01	25	00:01:36
C5	319	0:01:31	14847	00:48:29	31	00:00:39
C6	631	0:11:24	22529	02:32:39	30	00:05:21

time measurement for several browsers in the next section.

D-ForenRIA outperforms the basic solution in all cases. On average it takes the execution of 34 events to find a user-browser interaction while the basic solution needs 1720 events. Regarding the execution time, *D-ForenRIA* (even using a single browser) is orders of magnitudes faster than the basic solution. On average *D-ForenRIA* needs 12.7 seconds to detect an action while the basic solution needs around 8 minutes to detect an action.

Number of Resets per Action: Figures 6.7, 6.8 present 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 (Figure 6.8), 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.

Action Discovery Results: To better compare the performance of *D-ForenRIA* with the basic solution, we measured the cost to discover actions during the session reconstruction (Figures 6.9, 6.10, 6.11, 6.12, 6.13, 6.14). In all cases, the cost of action discovery in *D-ForenRIA* is several orders of magnitude less than the basic solution. However, one may

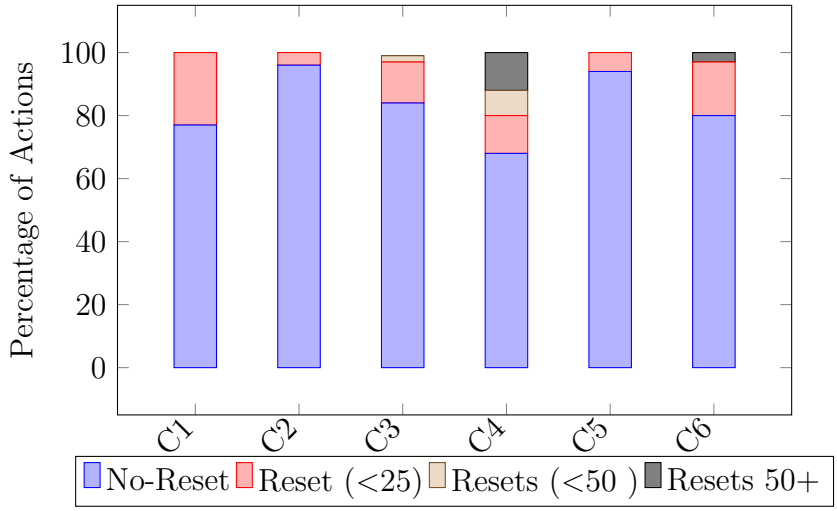


Figure 6.7: Breakdown of the number of resets needed to identify a user-browser interaction in *D-ForenRIA*

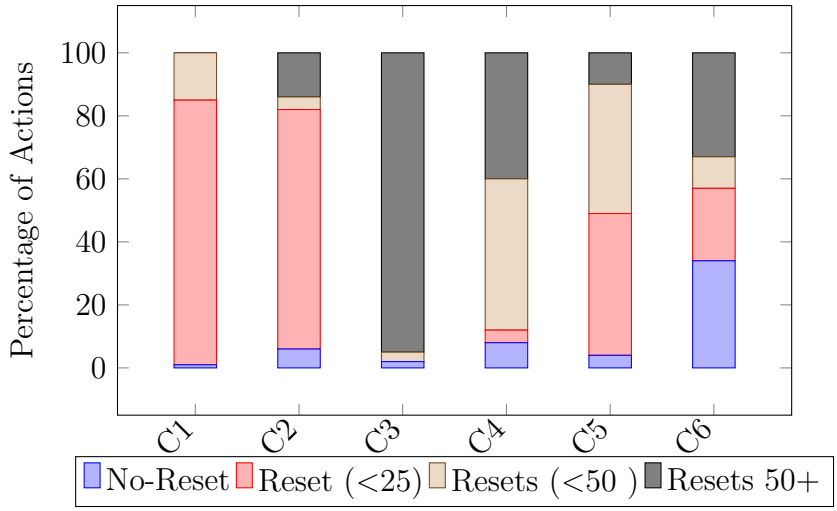


Figure 6.8: Breakdown of the number of resets needed to identify a user-browser interaction in the basic solution

observe that the difference between the basic solution and *D-ForenRIA* is less in some cases(such as C6) compared to other cases (such as C1). One reason behind this behavior is that we added the ability to handle randomness and user-input actions to the basic solution (Section 6.3.1).

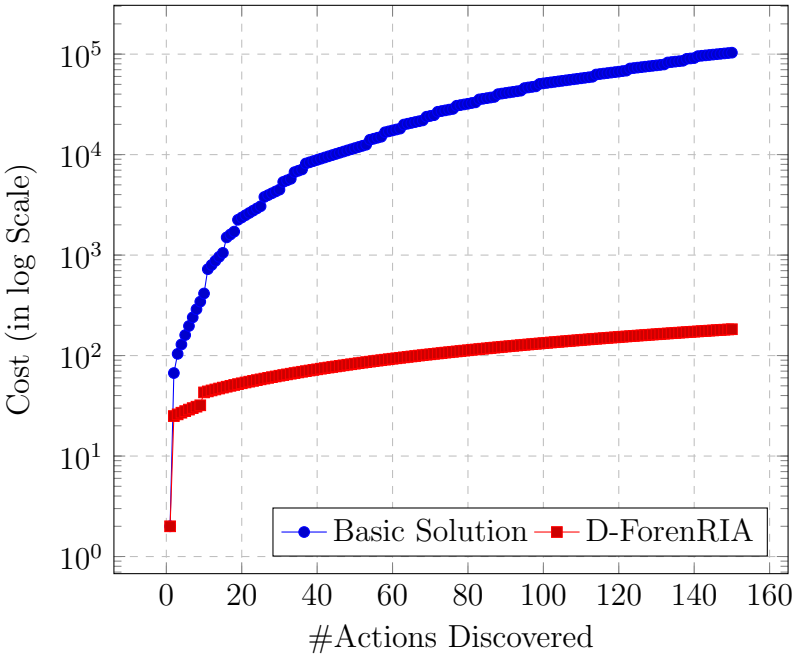


Figure 6.9: Action discovery cost for C1 (in log Scale)

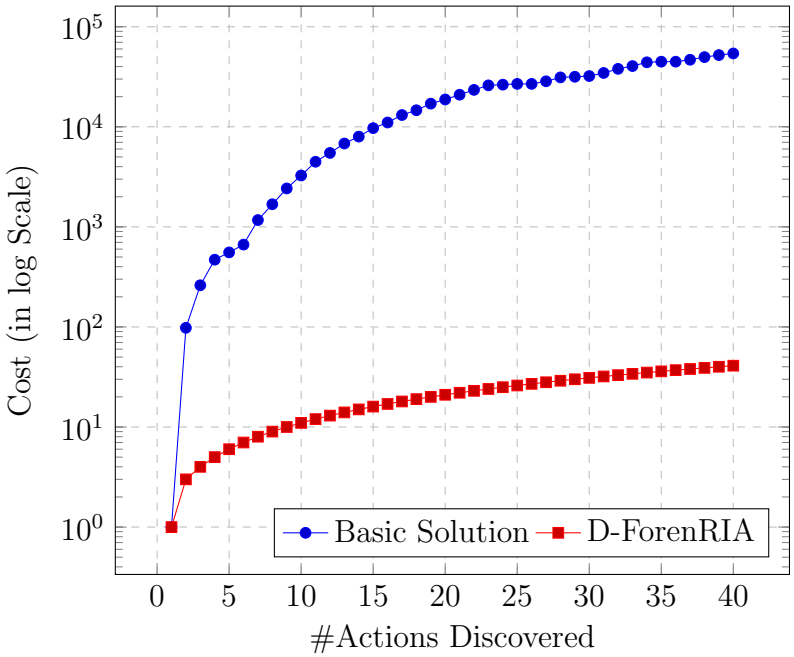


Figure 6.10: Action discovery cost for C2 (in log Scale)

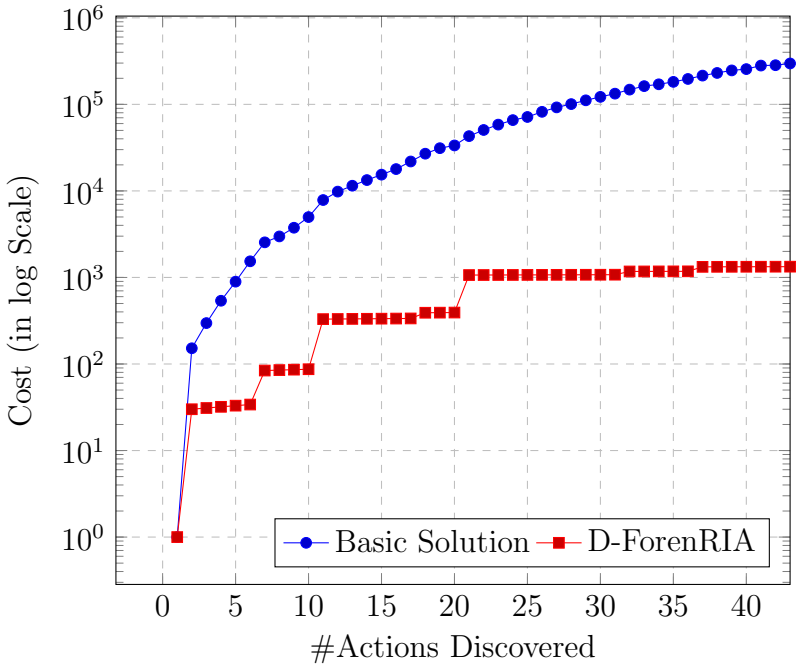


Figure 6.11: Action discovery cost for C3 (in log Scale)

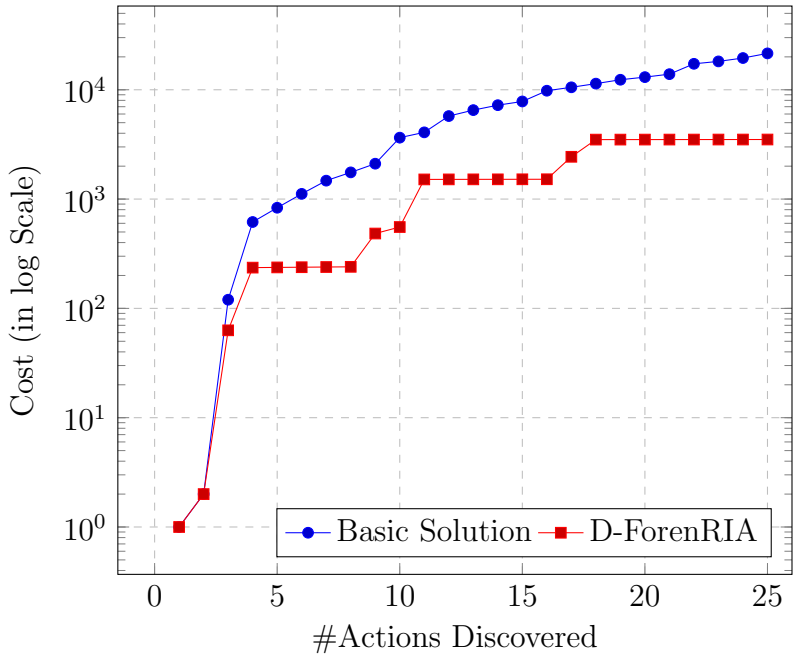


Figure 6.12: Action discovery cost for C4 (in log Scale)

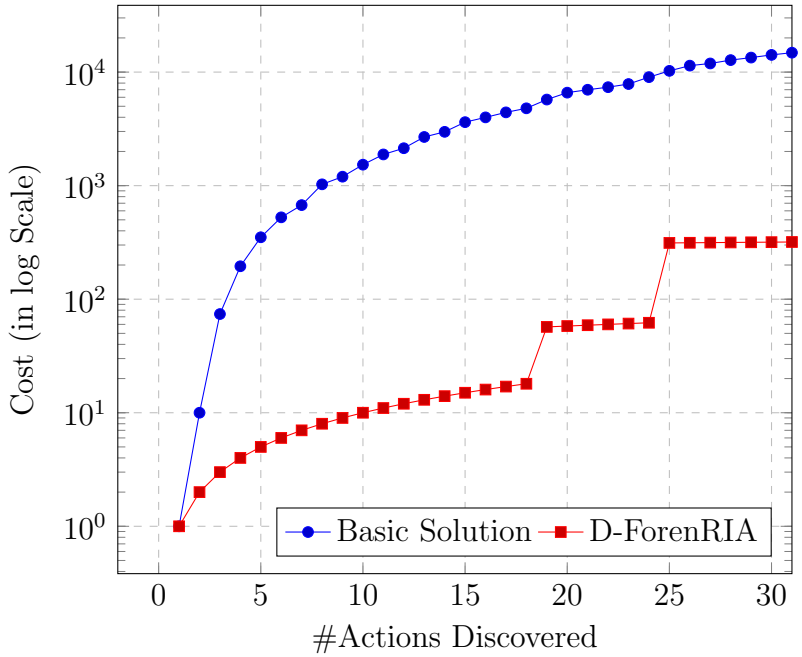


Figure 6.13: Action discovery cost for C5 (in log Scale)

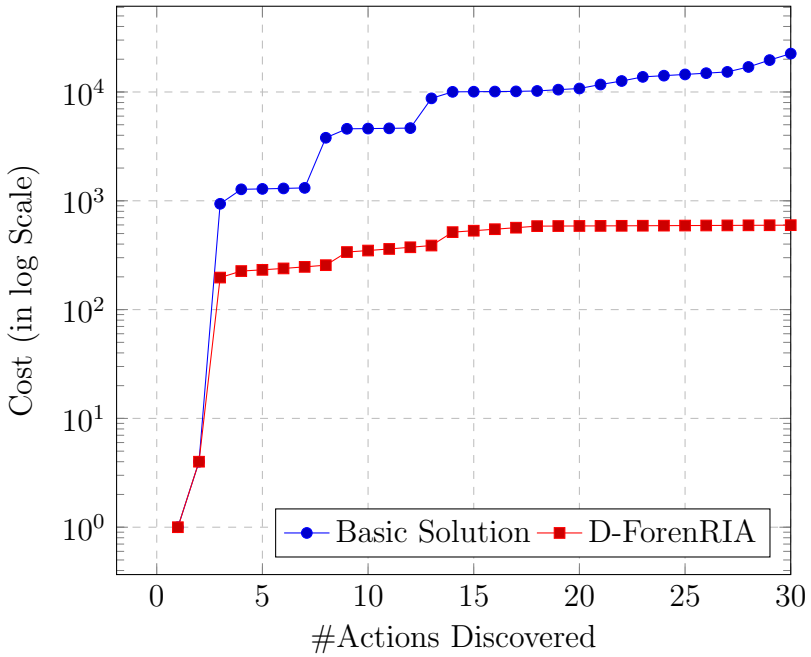


Figure 6.14: Action discovery cost for C6 (in log Scale)

Action discovery results also confirm the effectiveness of the candidate action ordering techniques in *D-ForenRIA*: there are a few slopes in each chart, followed by steps (near steps) in all cases. The slopes represent a large number of resets that are required to find an action. However, these resets usually lead to the discovery of new signatures, which in turn, helps *D-ForenRIA* to find other actions less costly.

6.5.2 Performance of the Distributed Architecture

Figures 6.15, 6.16, 6.17, 6.18, 6.19, 6.20 present 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 (Figures 6.17, 6.18, 6.20 respectively) where we have the largest number of resets (See Figure 6.8). For C5 (Figure 6.19) 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 4.5)). Sometimes, adding more browsers is not beneficial; for example in C1 and C2 (Figures 6.15 and 6.16), 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.

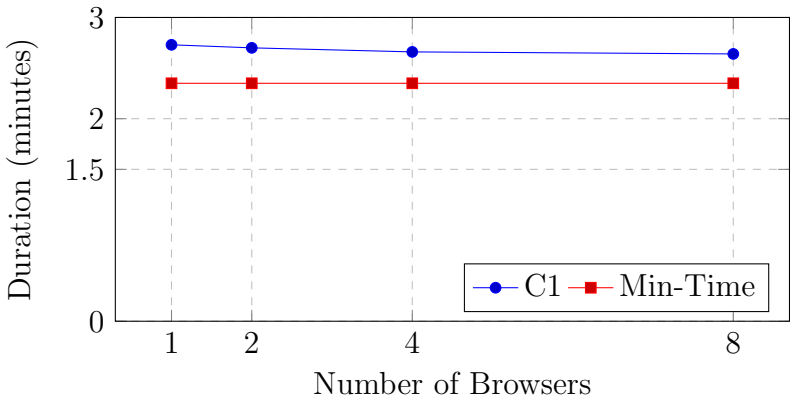


Figure 6.15: Scalability of *D-ForenRIA* in C1 compared to the Min-Time

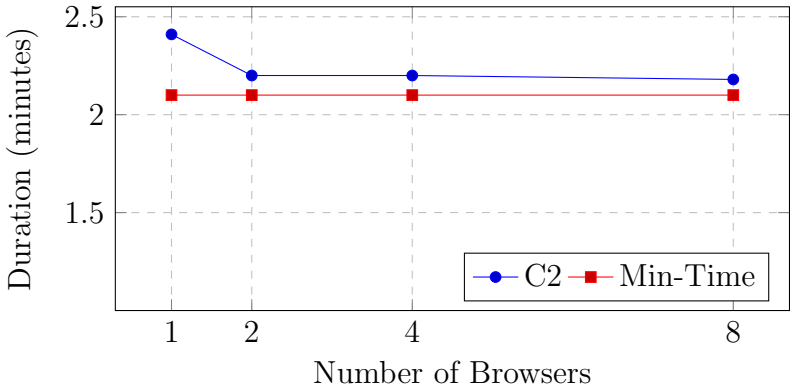


Figure 6.16: Scalability of *D-ForenRIA* in C2 compared to the Min-Time

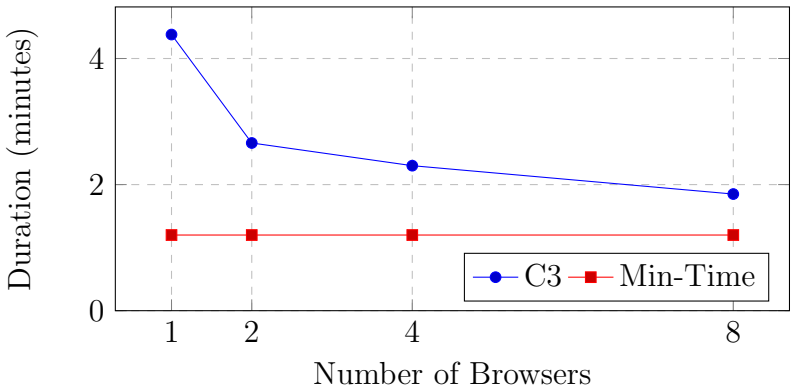


Figure 6.17: Scalability of *D-ForenRIA* in C3 compared to the Min-Time

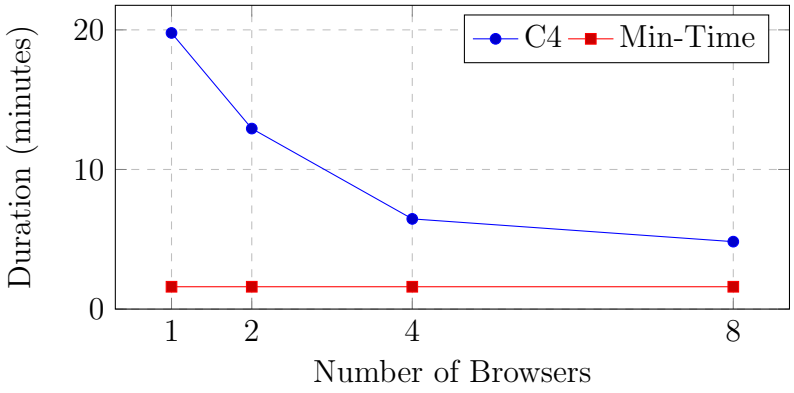


Figure 6.18: Scalability of *D-ForenRIA* in C4 compared to the Min-Time

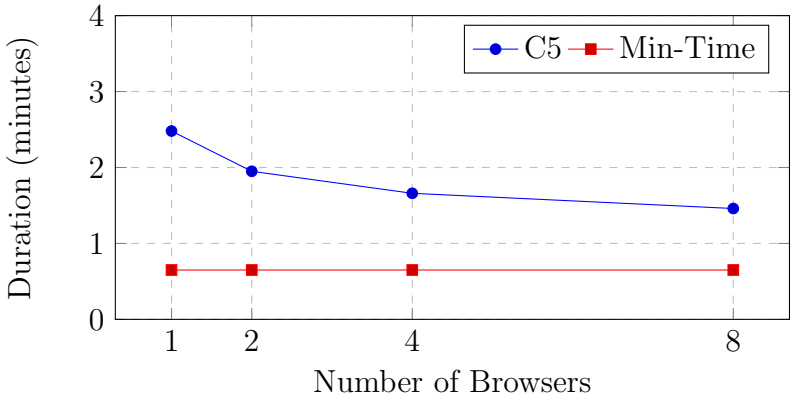


Figure 6.19: Scalability of *D-ForenRIA* in C5 compared to the Min-Time

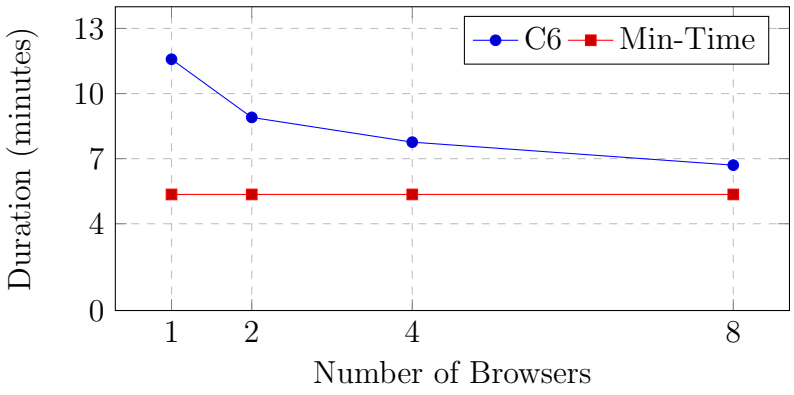


Figure 6.20: Scalability of *D-ForenRIA* in C6 compared to the Min-Time

6.5.2.1 Size of the Control Messages:

As we are going to evaluate the performance of *D-ForenRIA*, the performance of the network should be analyzed. In this section, we have a look at the size of the messages which are exchanged between the SR-Proxy and SR-Browsers. Here we report the size of important control messages that are exchanged between SR-Browsers and the SR-Proxy, and investigate whether these messages can be a bottleneck in the execution time of our system.
Although there are several types of control messages exchanged between SR-Browsers and SR-Proxy (Section 4.3.1), here we limit our experiments to *ExecuteAction* and *SendState* since these two messages are much larger in size compared to other control messages (such as the *Next* message). In addition to control messages, several HTTP messages are exchanged after performing an action; we will present some experiments regarding the size of these HTTP messages in Section 6.5.4.

Results for the ExecuteAction message: This message is sent from SR-Proxy to an SR-Browser and contains a sequence of actions that need to be executed by an SR-Browser. Figures 6.21, 6.22, 6.23, 6.24, 6.25, 6.26 plot the size of the "*ExecuteAction*" messages in C1-C6 respectively. In these figures, we observe larger *ExecuteAction* messages as the session reconstruction progress. This behaviour is related to *resets*; since when an SR-Browser tries an incorrect action, it needs to reload the last known good state. In order to transfer to the last discovered state, the SR-Proxy needs to send the list of all previously discovered actions to the SR-Browser. Therefore, as more actions are discovered, the transfer sequence and consequently the size of the *ExecuteAction* messages increase. However, in cases such as C1 and C2 where we have few resets, we observe fewer variations in the size of this message.

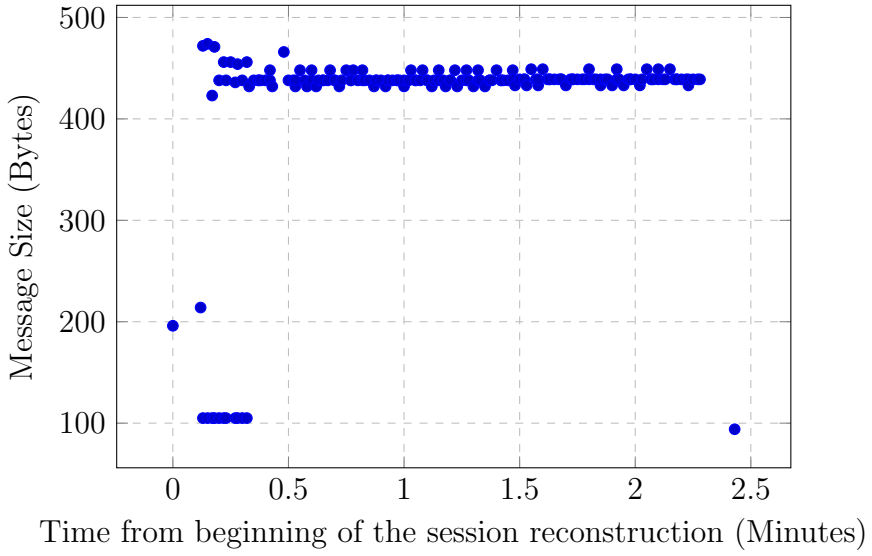


Figure 6.21: The size of the *ExecuteAction* message during session reconstruction of C1

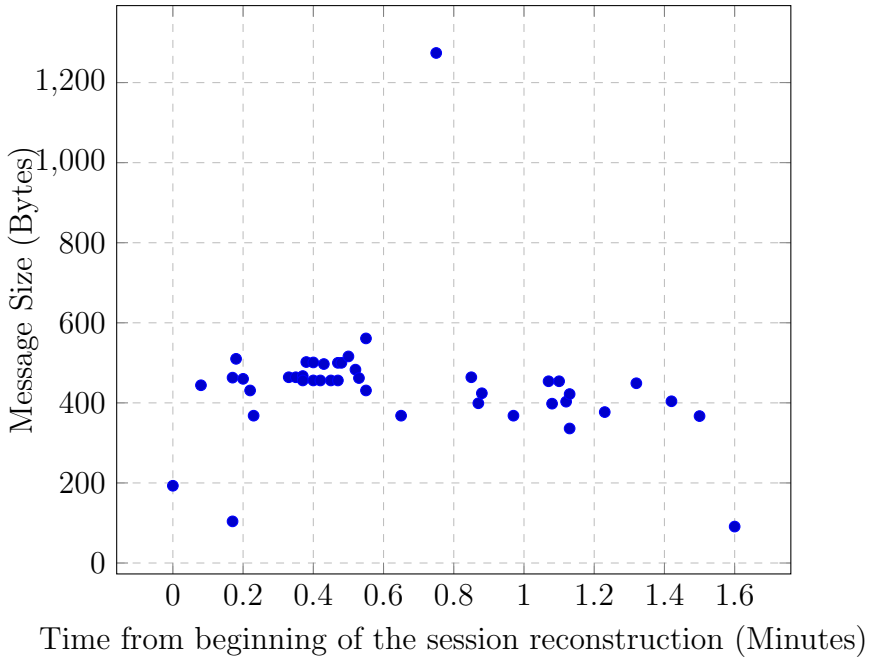


Figure 6.22: The size of the *ExecuteAction* message during session reconstruction of C2

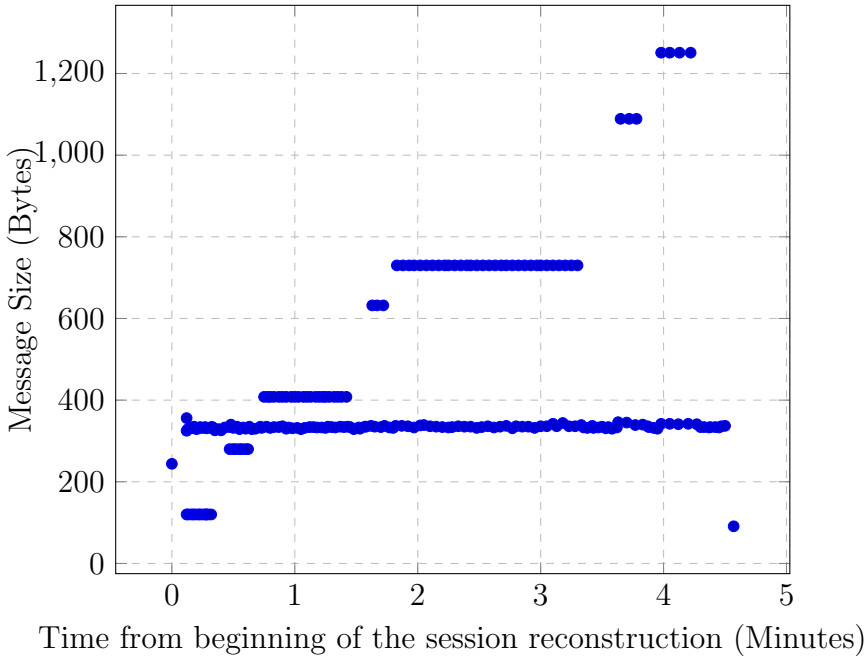


Figure 6.23: The size of the *ExecuteAction* message during session reconstruction of C3

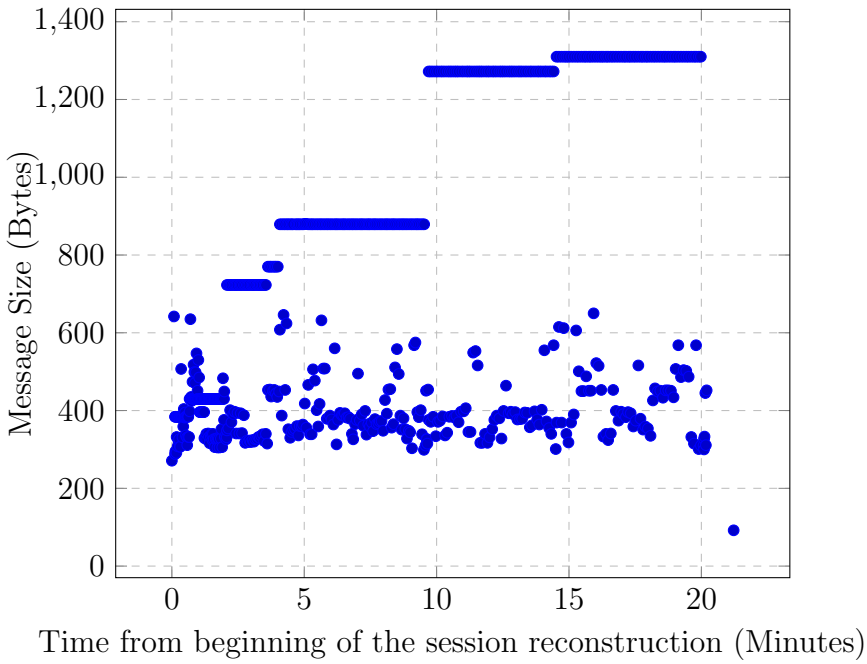


Figure 6.24: The size of the *ExecuteAction* message during session reconstruction of C4

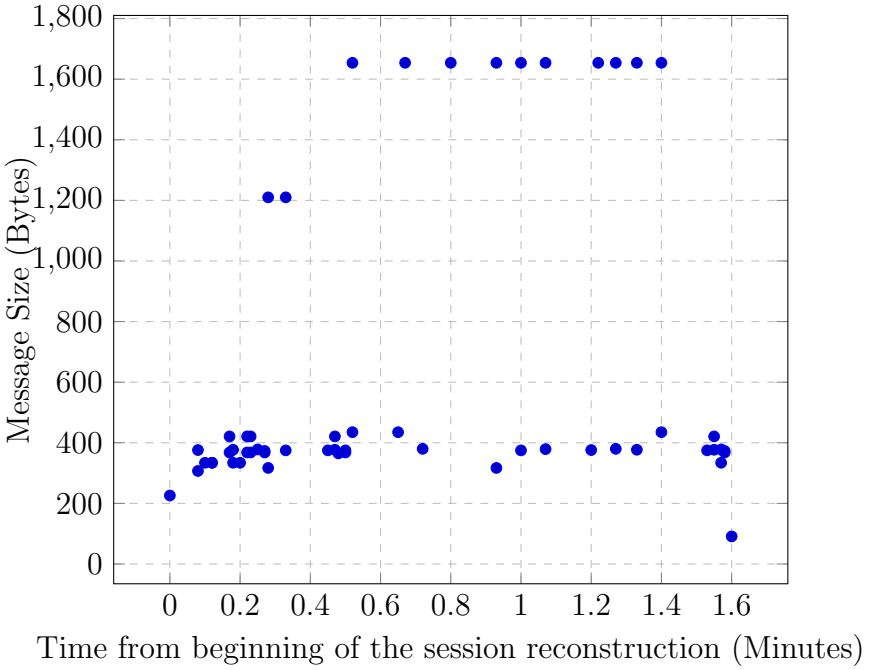


Figure 6.25: The size of the *ExecuteAction* message during session reconstruction of C5

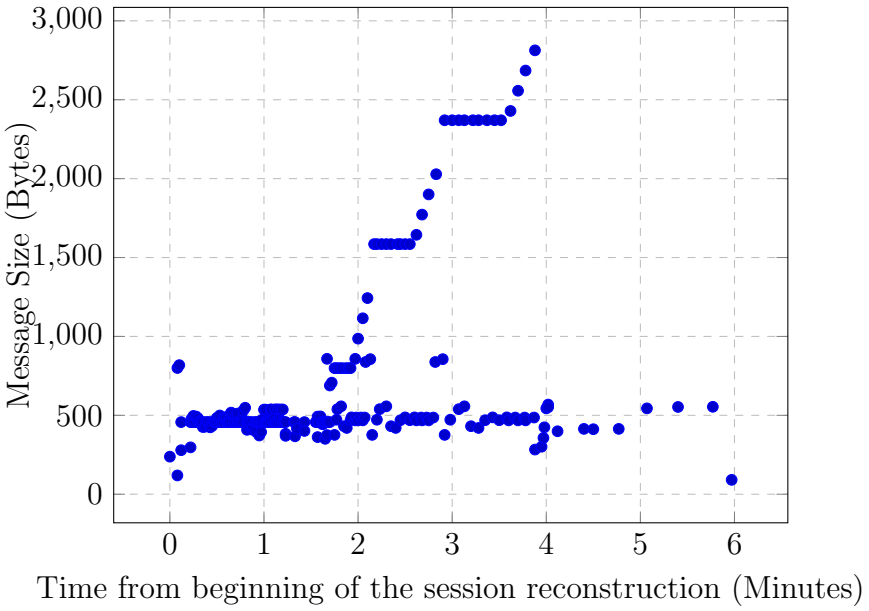


Figure 6.26: The size of the *ExecuteAction* message during session reconstruction of C6

Results for the SendState message: This message is sent from an SR-Browser to the SR-Proxy once we find a user interaction. The message includes the list of discovered user interaction in the current DOM and extra information that describe each action. Figure 6.27 plots the distribution of the size of "*SendState*" messages exchanged between the SR-Browser and the SR-Proxy. The average size of the "*SendState*" messages is 8.15 KBs. Although the average size of the *SendState* message is much larger than the average size of the *ExecuteAction* message (which is 0.52 KBs), the *SendState* will be sent once per user-interaction. These results show that the overhead of these messages are negligible; we show that the real bottleneck of the system is in browser-side execution of actions, as we will see in Section 6.6.

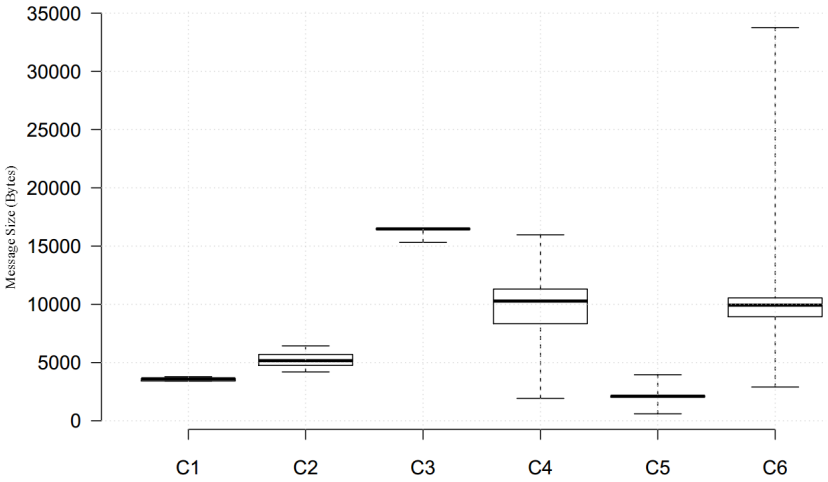


Figure 6.27: The Size of the "*SendState*" Message in Test Cases

6.5.3 Efficiency of Candidate Actions Ordering Techniques

As we discussed in Section 4.5, 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 6.3 presents the average of these measurements for all DOMS of the test cases.

Table 6.3: Characteristics of DOM elements and ratio of actions with signatures per DOM

Application	Number of Elements	Visible Elements(%)	Handler Elements(%)	DOM Leaves(%)	Signatures 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

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 4.5). 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 Figure 6.7).

Table 6.4: Log size features for test cases

Application	Average Requests per Action	Average Log Size per Action (KB)	Average Pruned Log Size per Action (KB)
C1	1.16	1.58	1.20
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

6.5.4 HTTP-Log Storage Requirements

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 6.4).

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[5]. 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.

[5]The compression was done using 7z algorithm

Table 6.5: Effect of the similarity-based ordering on the session reconstruction cost and time in different sites

Application	Similarity-Based Ordering Disabled		Similarity-Based Ordering Enabled	
	#Events	Time (H:m:s)	#Events	Time (H:m:s)
C1	183	0:02:44	183	0:02:50
C2	52	0:02:25	52	0:02:32
C3	1325	0:04:22	1320	0:04:30
C4	3506	0:19:47	878	0:05:18
C5	319	0:01:31	269	0:01:19
C6	631	0:11:24	631	0:11:25

6.5.5 Efficiency of Similarity-based Detection

To measure the efficiency of the similarity-based ordering, we ran *D-ForenRIA* with a single browser, and enabled similarity-based ordering. Table 6.5 presents the results of applying the similarity-based ordering during the reconstruction:

The performance of the similarity-based ordering varies; as we mentioned in previous sections, the similarity-based ordering is only applied when the signature-based ordering is not applicable. Therefore, in the test cases that the signature-based ordering is usually effective, we do not expect any improvement by using similarity-based ordering. The results we obtained verified this argument.

The best speedup happens in C4. In this website, the similarity-based ordering decreased the session reconstruction cost by 75%. Figure 6.28 presents the action discovery results by applying similarity-based ordering in C4. As we can see in Figure 6.28, enabling similarity-based ordering successfully helps us to reconstruct the session in a less costly way. In this website, there are several groups of similar actions such as the list of the comments, the list of the team members, and the list of project goals. The requests generated after performing any action of these groups are quite similar. Therefore after examining 2 actions of each group, the similarity-based ordering algorithm detects these groups, and helps *D-ForenRIA* to perform a better ordering of candidate actions at each state.

To better investigate the impact of similarity-based ordering in C4, we plot the distribution of the number of resets required to detect an action in the box-plot of Figure 6.29.

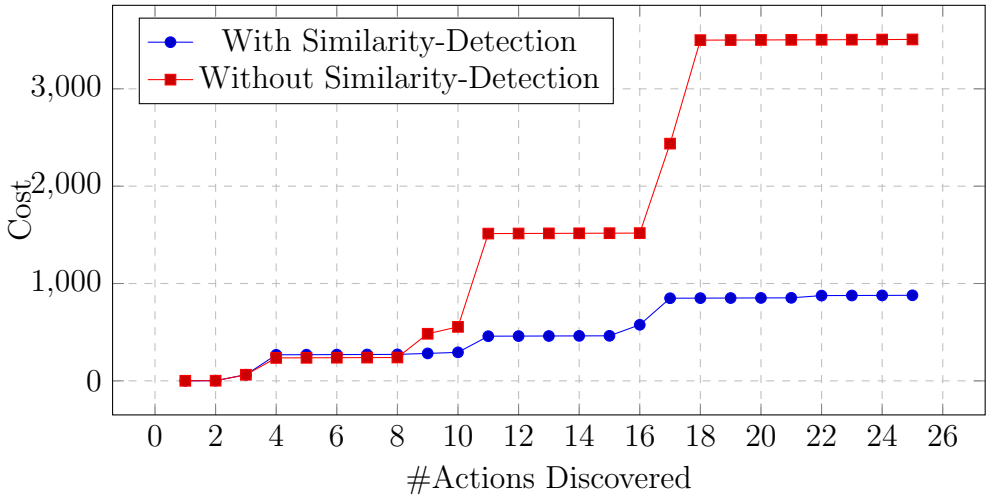


Figure 6.28: Effect of Similarity-Based Ordering on Action Discovery Cost for C4

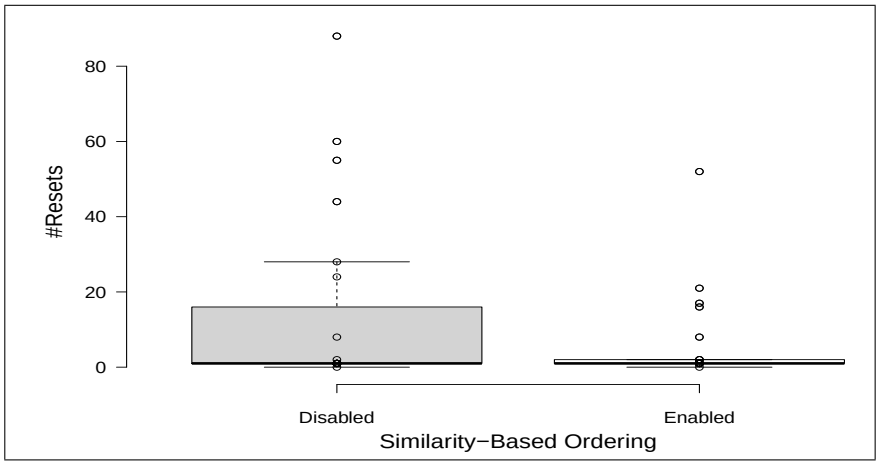


Figure 6.29: Distribution of the number of resets required to
detect an action in C4 when the similarity-based ordering is enabled/disabled

On average, when we enable the similarity-based ordering the number of the resets required per action decreases from 13.5 to 5.6 .

The similarity-based ordering is also effective for C5. On C5, there are several lists of products, pictures, and services.The similarity-based detection algorithm detects these lists (as similar groups) and guides the session reconstruction algorithm to try more promising actions first. Figure 6.30 presents the action discovery results for this website. In this website the majority of actions are found immediately, even when the similarity detection

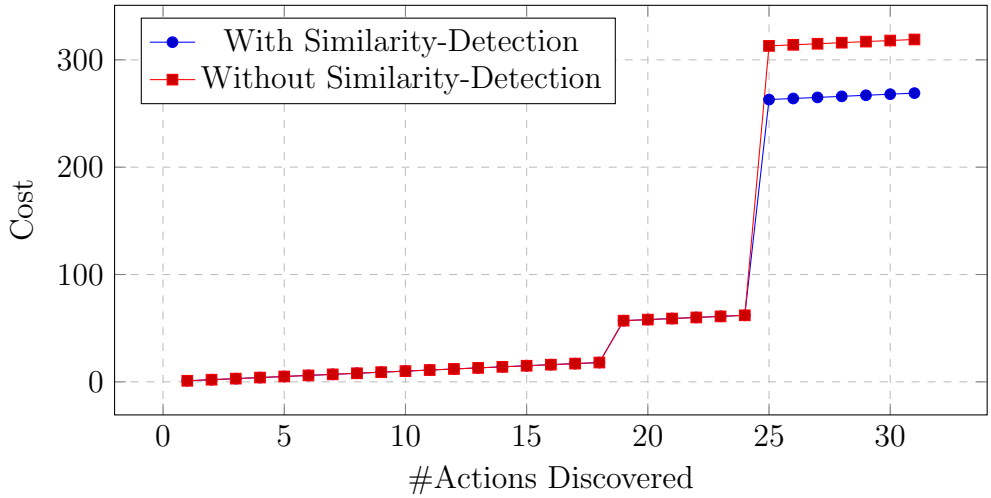


Figure 6.30: Effect of Similarity-Based Ordering on Action Discovery Cost for C5

is disabled, however, the similarity detection further improves the execution cost by 15%.

Sometimes similarity-based is not effective at all. For example in C1, and C2 the signature-based ordering is quite effective and most of the actions are found immediately. Therefore there is no need to apply similarity-based ordering. For C3, using similarity-based ordering did not change the cost of reconstruction. The reason is that the similarity detection algorithm works well when there are "several" groups in a state, and trying some groups are more promising than the others. Therefore, when the majority of elements are in a single group on a page, the session reconstruction is not remarkably effective. This situation happens in C3, when the algorithm detects a single group of similar elements for the whole elements of the periodic table.

Table 6.5 also presents the effect of enabling similarity-based detection on session reconstruction time. In cases where the similarity-based ordering is not effective (C1, C2, C3 and C6), the execution time is increased. However, the computational cost of this technique is negligible; in our test cases, the average overhead time of enabling similarity-based detection was 3%.

To summarize, the similarity-based algorithm allows for better ordering of candidate actions in some cases. However, when there the signature-based ordering is effective, our current algorithm does not provide useful changes in the efficiency of the session reconstruction.

6.6 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[6] or IIS[7]) 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[8].

In addition, the traffic can also be captured while it goes through the network using other tools such as proxies. 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 HTTP traffic: In *D-ForenRIA* it is assumed that the traffic is in plain text format and no decryption is needed on the input. Although SSL[35] provides a secure connection between a client and the server it does not encrypt logs on the server. Therefore SSL is not a barrier for recording the traffic on the server.
However, a real issue with SSL-enabled sites exists during the session reconstruction process: The SR-Browsers communicate with the SR-Proxy as if the latter was the real server. Therefore the exchanged 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[16]; 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

[6] `http://httpd.apache.org/docs/2.2/mod/moddumpio.html`
[7] `https://msdn.microsoft.com/en-us/library/ms227673.aspx`
[8] It is notable that the SR-Proxy in *D-ForenRIA* is just used during the reconstruction and not for recording the traffic.

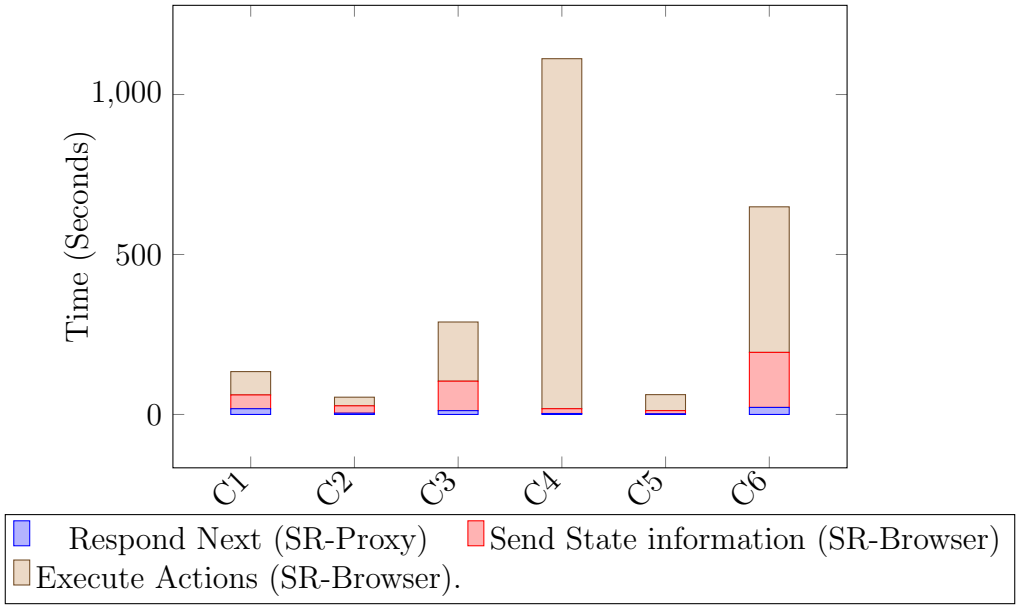


Figure 6.31: Breakdown of the Amount of Time Spent to Respond to the Control Messages during Session Reconstruction

as sample data to detect user-input actions are going to be observed in the request generated after performing the action (Section 4.7). 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 haven't seen many RIAs that encode textual user inputs.

Bottlenecks in D-ForenRIA's execution time: To further analyze the total execution time during session reconstruction, we have calculated how much time is spent

to process the control messages exchanged between the SR-Proxy and SR-Browsers (Figure 4.2): here we report the time needed to respond to the "*Next*" messages, the time needed by the SR-Browser to extract the state information, and also the time needed to execute the actions in the SR-Browser.

As depicted in Figure 6.31, a large part of the execution time is spent on executing the actions in the SR-Browser. On average, 56% of the time is spent in this part. We have also measured the time spent on reloading the last known good state (Section 4.9). Reloading the last good state is required when an SR-Browser reaches an incorrect state (i.e. incorrect action was executed). For our test cases, on average, 39% of the time is time is spent to transfer the SR-Browser to its previous state. Therefore, if we implement instant reloading of the previous state, *D-ForenRIA* can gain a considerable speedup. The SR-Browser also spends 19% of the reconstruction time, to respond to the "*SendState*" messages. On the other hand, responding to the "*Next*" messages by the SR-Proxy, takes 4% of the total execution time. These results have shown that session reconstruction is heavily dependent on the browser-side execution and we expect much faster reconstruction with more efficient methods for executing actions, and reloading the last known good state.

Temporal Headers in Responses: HTTP responses can have temporal headers such as *Date*, *expires* and *last-modified* which may impact replay since the time of replay does not correspond to the time of capture. For example if a resource has an *expires* header set to a time t_1 and it is replayed at time $t_2 > t_1$, the SR-Browser will automatically request the resource again, impacting the replay. To solve this issue, the SR-Proxy updates all these headers using the current time and the relative time offset at the time of recording.

Variations in an action's signatures: The signature-based ordering assumes that an action would generate the same set of requests at different states. When it 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 and our experimental results verify this claim (Figure 6.7), there are a few cases where this assumption is not satisfied. In our test applications, we observed this case in one of our test-cases; in C5 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. In this website, 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.

Incomplete Traces: A trace may be incomplete and does not contain all the requests/responses needed to reconstruct a session. For example, suppose that an action is performed at time t_1 by the user, and the generated requests/responses are cached by the browser. If the user again performs the same action at time $t_2 > t_1$, it may not generate any request to the server. If we record the traffic after t_1, it can cause a problem for the reconstruction tool: when the session reconstruction tool tries the action, it generates some requests that do not exist in the given trace (since they had been cached by the user's browser). In this case, the session reconstruction tool cannot recover the action. However, if we have recorded the traffic before time t_1, the generated requests exist in the traces and the session reconstructions tool can recover the action.

Importance of multi-browser support: Selenium 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 the Selenium is that has the support of widely used 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.

Genezationralization 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 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 6.6). In addition, each of these applications is bringing new challenges for the session reconstruction tool. Table 6.7 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.

Table 6.6: 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

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[9]. As Table 6.8 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.

[9]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).

Table 6.7: Challenges in our test cases

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

Table 6.8: Code coverage of our traces

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

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

6.7 Conclusion

This chapter studied the performance of *D-ForenRIA* to reconstruct sessions of six RIAs. We evaluated our method by reporting the number of JavaScript events executed during the reconstruction of each session. *D-ForenRIA* (even with a single browser), is several orders of magnitude more efficient than a basic session reconstruction approach. Adding more SR-Browsers to *D-ForenRIA* were also beneficial in cases where the signature-based was not effective. We also measured the HTTP-log storage requirements of *D-ForenRIA* and studied the effect of different candidate actions's ordering strategies on the efficiency of the system. We have also reported some experimental results about similarity-detection

technique; the results show that the proposed idea can sometimes speedup the session reconstruction process; however, it does not make a significant difference when the majority of actions can be detected based on signature-based ordering, or when the RIA does not contain several similar groups. This chapter also included a critical discussion on the limitations of *D-ForenRIA* and findings of the experimental results.

Chapter 7

Conclusion and Future Works

7.1 Conclusion

The emergence of AJAX, has provided more interactive and user-friendly Web applications. Despite the benefits of RIAs, the previous session reconstruction methods for traditional Web applications are not effective anymore. It means that we are no longer able to extract user-browser interactions from the HTTP log of a session.

To reconstruct a session, a basic reconstruction tool considers all possible user interactions in each RIA state, tries them one by one using a browser, and verifies whether the generate requests match the requests in the user log or not. This process is repeated in other states of the RIA until all the user-interactions are recovered. For large Web applications, where a large number of candidate user interactions exist in each RIA state, this process is very time-consuming.

In this thesis, we focused on providing efficient strategies to reconstruct RIA sessions in less time. To this end, we proposed signature-based ordering and similarity-based ordering to prioritize candidate actions based on the observed HTTP traffics during the reconstruction. To further reduce the reconstruction time, we also used a distributed set of browsers to concurrently try candidate actions in each state. On top of these improvements, to have a practical session reconstruction tool, we proposed techniques to extract user-input values from the user-log, techniques to handle randomness in request's parameters, and to recover actions that do not generate any traffic. We conducted experiments on six RIAs

and reported the effectiveness of our method on reducing the session reconstruction time. Some of the contributions of this thesis include:

- *A General Session Reconstruction Algorithm*: To the best of our knowledge, this work is the first work that formally introduces the session reconstruction problem. The problem was generalized to extracting user-interactions from a request/response log of any client/server application. We also offered a solution for the efficient session reconstruction problem. This solution is implemented, in the context of RIAs, in a tool called *D-ForenRIA*.

- *Distributed Session Reconstruction Architecture*: This thesis is the first work to introduce a distributed architecture for the reconstruction of sessions. The architecture is composed of a set of browsers and a coordinator proxy. The browsers can be added to the system, at any time and provide the ability to concurrently try different actions on each state. The experiments proved that this architecture is complementary to the signature-based ordering, and reduces the session reconstruction time when the signature-based ordering is not effective.

- *Signature-Based Ordering of Candidate Actions*: This thesis offered the idea of using the "signature" of actions to order the candidate actions on each state. This is a learning mechanism which is based on the history of generated requests during event executions. The experiments showed that on average this technique, can detect the correct user action on 83% of cases.

- *Similarity-Based Ordering of Candidate Actions*: In addition, in this thesis we introduced the idea of "similarity-based" ordering of candidate actions; the similarity-detection algorithm predicts the behaviour of an action based on the signature of similar actions. The similarity detection is potentially helpful when the signature of actions are unknown.

- *Recovering User-Input Data*: In this thesis, we proposed the first general method to extract user-input actions from HTTP logs. The method is applicable to any method of transforming user-input data into a request. This feature is especially important in the case of RIAs, where several different technologies are used to submit user-inputs.

We talked about the limitations of the method and provided some ideas to overcome some of these limitations.

- *A Path-Completion technique for RIAs*: In this thesis, we took the first step toward solving the path completion problem for RIAs. Some of the actions that their traffic is missing in the log, can be inferred by using the proposed method. This method uses a graph search in an auxiliary graph, action-graph, that is extracted during the session reconstruction.

We conclude the thesis with some future directions.

7.2 Future Works

7.2.1 More Efficient Ordering of Candidate Actions

In this thesis we proposed to use the signature-based ordering to prioritize candidate actions at each state; however, when the signature of an action is unknown, the algorithm is neutral about the priority of an action. Therefore, when there are a lot of new actions on a state, the signature based ordering becomes ineffective. To address this issue, we have introduced the similarity-based ordering. However, this technique is based on *heuristic* functions which determine the similarity between actions based on the similarity of HTTP requests. These heuristics are not always precise and can sometimes categorize dissimilar actions as similar. Therefore, proposing better criteria to define similarity between actions can be considered as a future direction of this research.

7.2.2 Multi-User Session Reconstruction

We have assumed that the input for the session reconstruction tool is for a single-user. In practice, the reconstruction may be done for several users of a RIA. *D-ForenRIA* can be changed to sequentially reconstruct several users' sessions. Another approach would be to concurrently run several instances of the session reconstruction tool.

The information that is found during the reconstruction of a user, may be useful for the reconstruction of another user's session. One example of such information is the signatures

of actions; at each run of the session reconstruction tool, the set of extracted signatures can be saved, and later loaded for the next run of the session reconstruction tool. In the concurrent multi-user session reconstruction, once an instance of the session reconstruction tool finds a signature, it can send this information to the other instances. Another approach would be to use a central repository of signatures that can be accessed by all instances of session reconstruction tool.

In multi-user session reconstruction we record the traffic of users over time; since RIAs usually evolve, recorded traffics may correspond to different versions of the RIA. Therefore, the extracted signatures from previous session reconstructions may not be valid anymore; for example when the behaviour of an action changes and performing the action generates a totally different set of requests. In this case, the algorithm uses the new set of generated requests as the signature of such actions.

7.2.3 Relaxing Assumptions about Access to the Server during Reconstruction

We have assumed that during reconstruction we have no access to the server. This assumption enables the tool to operate even when the server is not accessible, for example because of an attack. However, relaxing this assumption may help the session reconstruction to be performed more efficiently; if the session reconstruction tool have an on-line access to the server, it can potentially try all actions of the RIA (using a crawler) and extract a large number of signatures; Using a large number of signatures, the algorithm should be able to reconstruct sessions more efficiently. However since RIAs change over time, a challenge to this preprocessing approach is to detect RIA changes and update the set of signatures appropriately.

7.2.4 Better Handling of the non-Determinism

We have assumed that the application is deterministic from the viewpoint of the user. It means that given a state and an input action, the next state is uniquely defined. On the other hand, we assumed that we expect that an action may generate different sequences of

requests whenever performed from a given state. We have proposed a method to handle this kind of randomness.

However, the method to handle the randomness in requests, cannot cope with all the possible cases of the randomness; the method assumes that the randomness happens in the values of the parameters of the requests, and the number of requests and parameters are fixed. However, In real Web applications, this assumption may not hold. The matching mechanism of the session reconstruction algorithm should be improved to handle complex cases of the randomness.

7.2.5 New Application Domains

Exploring how *D-ForenRIA* can be used in a Web usage mining tool is another research area. The existing Web usage mining tools work well with traditional Web applications. An assumption that is made is that the user interactions during a session are associated with visited pages. However, these tools cannot handle RIAs. We believe that our session reconstruction tool can be applied as a preprocessing step for a Web usage mining process; the extracted actions, can be considered as the input to the Web usage mining tool.
Web usage mining tools extract useful patterns from traces of several users. Therefore, *D-ForenRIA* needs to be extended to effectively reconstruct traces of several users.

The session reconstruction tool may also be used for debugging a RIA. If you have either report of a bug, or observe an error message in the logs, the session reconstruction tool can be used to create the session that has lead to the bug.

7.2.6 Relaxing Assumptions about User-Input Actions

For user-input actions, we made two assumptions that may be limiting in practice: The assumption that the algorithm can detect the domain of possible values for each user-input action, and the assumption that the user-input values can be observed in the requests. These assumptions may not satisfy in all RIAs. A text input that expects a user to enter an input in a predefined format (such as a date) breaks the first assumption. Client-side coded values such as a password value that is coded in the client-side, break the

second assumption. To relax these assumptions, the session reconstruction tool needs to be improved.

The research in deep Web crawling may be useful here; a deep web crawler tries to find the content hidden behind HTML forms. Deep web crawlers apply different techniques to select the proper values to submit a form. These techniques may be used here to improve how the session reconstruction tool selects input values.

7.2.7 Applying the Abstract Algorithm to other Client/Server Applications

In this thesis, we provided an abstract session reconstruction algorithm for any application that is based on a client/server architecture. The algorithm can potentially be adapted to different contexts (such as mobile applications/ Web services). The reason is that the proposed session reconstruction algorithm is independent of any specific technology. The underlying technology of an application affects the algorithm to detect possible user-actions, and to define the format of the exchanged request/responses, which are given to the session reconstruction algorithm as inputs.

7.2.8 Handling Incomplete Traces

Another assumption we made in this thesis is that the input trace is complete; it means that during the reconstruction, there is no need for a traffic from previous sessions. This assumption may not hold in practice; the input trace may not be recorded from the beginning of the session, and the resources that are cached in the previous session would not exist in the current session of a user. The session reconstruction algorithm in these cases may require to either infer the missing traces (possibly from another user's trace), or to redefine the matching mechanism for requests.

References

[1] World Wide Web Consortium (W3C). Logging Control in W3C httpd. `https://www.w3.org/Daemon/User/Config/Logging.html`, 1995. Accessed: 2017-06-08.

[2] World Wide Web Consortium (W3C). Document Model Events. `https://www.w3.org/TR/DOM-Level-2-Events/events.html`, 2000. Accessed: 2017-06-08.

[3] World Wide Web Consortium (W3C). HTML5 forms recommendations. `https://www.w3.org/TR/2014/REC-html5-20141028/forms.html#forms`, 2014. Accessed: 2016-11-25.

[4] ECMAScript 2016 Language Specification. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, 2016. Accessed: 2016-05-23.

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

[6] Richard Atterer. Logging usage of ajax applications with the "usaproxy" http proxy. In *Proceedings of the WWW 2006 Workshop on Logging Traces of Web Activity: The Mechanics of Data Collection*, 2006.

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

[8] Khalil Ayoub, Hosam Aly, and Jason Walsh. Document object model (dom) based page uniqueness detection, July 16 2013. US Patent 8,489,605.

[9] Sara Baghbanzadeh, Salman Hooshmand, Gregor Bochmann, Guy-Vincent Jourdan, Seyed M. Mirtaheri, Muhammad Faheem, and Iosif Viorel Onut. Reconstructing interactions with rich internet applications from http traces. In *Proceedings of the Twelfth Annual IFIP WG 11.9 International Conference on Digital Forensics*, pages 147–164. Springer, 2016.

[10] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. *ACM SIGMETRICS Performance Evaluation Review*, 26(1):pages 151–160, 1998.

[11] Bettina Berendt and Myra Spiliopoulou. Analysis of navigation behaviour in web sites integrating multiple information systems. *The VLDB JournalThe International Journal on Very Large Data Bases*, 9(1):pages 56–75, 2000.

[12] Tim Berners-Lee, Larry Masinter, and Mark McCahill. Uniform resource locators (url). Technical report, 1994.

[13] Aurelien Bouteiller, George Bosilca, and Jack Dongarra. *Retrospect: Deterministic replay of MPI applications for interactive distributed debugging*, pages 297–306. Springer, 2007.

[14] Alex G Büchner and Maurice D Mulvenna. Discovering internet marketing intelligence through online analytical web usage mining. *ACM Sigmod Record*, 27(4):pages 54–61, 1998.

[15] Brian Burg, Richard Bailey, Andrew J Ko, and Michael D Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 473–484. ACM, 2013.

[16] Franco Callegati, Walter Cerroni, and Marco Ramilli. Man-in-the-middle attack to the https protocol. *IEEE Security and Privacy*, 7(1):pages 78–81, 2009.

[17] Calin Cascaval, Seth Fowler, Pablo Montesinos-Ortego, Wayne Piekarski, Mehrdad Reshadi, Behnam Robatmili, Michael Weber, and Vrajesh Bhavsar. Zoomm: a parallel web browser engine for multicore mobile devices. In *ACM SIGPLAN Notices*, volume 48, pages 271–280. ACM, 2013.

[18] Lara D Catledge and James E Pitkow. Characterizing browsing strategies in the world-wide web. *Computer Networks and ISDN systems*, 27(6):pages 1065–1073, 1995.

[19] Brian Chess, Yekaterina Tsipenyuk ONeil, and Jacob West. Javascript hijacking. `https://j11y.io/wp-content/uploads/2009/03/javascript_hijacking.pdf`, 2007. Accessed: 2017-06-08.

[20] Jong-Deok Choi and Harini Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59. ACM, 1998.

[21] Brian Clifton. *Advanced web metrics with Google Analytics*. John Wiley & Sons, 2012.

[22] Robert Cooley, Bamshad Mobasher, and Jaideep Srivastava. Data preparation for mining world wide web browsing patterns. *Knowledge and information systems*, 1(1):pages 5–32, 1999.

[23] John R Corbin. *The art of distributed applications: programming techniques for remote procedure calls*. Springer Science & Business Media, 2012.

[24] Frank Cornelis, Andy Georges, Mark Christiaens, Michiel Ronsse, Tom Ghesquiere, and KD Bosschere. A taxonomy of execution replay systems. In *Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003.

[25] Robert F Dell, Pablo E Román, and Juan D Velásquez. Web user session reconstruction using integer programming. In *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, volume 1, pages 385–388. IEEE Computer Society, 2008.

[26] Mustafa Emre Dincturk. *Model-based crawling-an approach to design efficient crawling strategies for rich internet applications*. PhD thesis, EECS-University of Ottawa, 2013. `http://ssrg.eecs.uottawa.ca/docs/Dincturk_MustafaEmre_2013_thesis.pdf`.

[27] Mahendra Pratap Singh Dohare, Premnarayan Arya, and Aruna Bajpai. Novel web usage mining for web mining techniques. *International Journal of Emerging Technology and Advanced Engineering*, 2(1):pages 253–262, 2012.

[28] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):pages 211–224, 2002.

[29] Mohammadreza Barouni Ebrahimi, Obidul Islam, and Iosif V Onut. Clustering repetitive structure of asynchronous web application content, August 15 2017. US Patent 9,734,147.

[30] Kobra Etminani, Amin Rezaeian Delui, Noorali Raeeji Yanehsari, and Modjitaba Rouhani. Web usage mining: Discovery of the users' navigational patterns using SOM. In *Proceedings of the first international conference on Networked Digital Technologies (NDT'09)*, pages 224–249. IEEE, 2009.

[31] Amin Milani Fard and Ali Mesbah. Feedback-directed exploration of web applications to derive test models. In *Proceedings of the 24th International Symposium on Software Reliability Engineering (ISSRE)*, volume 13, pages 278–287, 2013.

[32] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1. `http://www.w3.org/Protocols/rfc2616/rfc2616.html`, 1999. Accessed: 2017-09-08.

[33] David Flanagan. *JavaScript: the definitive guide*. O'Reilly Media, Inc., 2006.

[34] Piero Fraternali, Gustavo Rossi, and Fernando Sánchez-Figueroa. Rich Internet Applications. *Internet Computing, IEEE*, 14(3):pages 9–12, 2010.

[35] Alan Freier, Philip Karlton, and Paul Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. `https://tools.ietf.org/html/rfc6101`, 2011. Accessed: 2016-01-12.

[36] Jesse James Garrett. Ajax: A new approach to web applications. `http://adaptivepath.org/ideas/ajax-new-approach-web-applications`, 2005. Accessed: 2017-06-08.

[37] Dennis Michael Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. *Replay debugging for distributed applications*. PhD thesis, University of California, Berkeley, 2006.

[38] David Gourley and Brian Totty. *HTTP: the definitive guide*. O'Reilly Media, Inc., 2002.

[39] Ilya Grigorik. HTTP caching. `http://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching?hl=en`, 2017. Accessed: 2017-09-05.

[40] A. Grosskurth and M. W. Godfrey. A reference architecture for web browsers. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 661–664, 2005.

[41] Seung-Sun Hong and Shyhtsun Felix Wu. On interactive internet traffic replay. In *International Workshop on Recent Advances in Intrusion Detection (RAID)*, volume 3858, pages 247–264. Springer, 2005.

[42] Tasawar Hussain, Sohail Asghar, and Nayyer Masood. Web usage mining: A survey on preprocessing of web log file. In *Proceedings of the International Conference on Information and Emerging Technologies (ICIET)*, pages 1–6. IEEE, 2010.

[43] Sunghwan Ihm and Vivek S Pai. Towards understanding modern web traffic. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 295–312. ACM, 2011.

[44] Staffs Keele et al. Guidelines for performing systematic literature reviews in software engineering. In *Technical report, Ver. 2.3 EBSE Technical Report. EBSE*. sn, 2007.

[45] Mohamed Koutheair Khrib, Mohamed Jemn, and Olfa Nasraoui. Automatic recommendations for e-learning personalization based on web usage mining techniques and information retrieval. In *Proceedings of the Eighth IEEE International Conference on Advanced Learning Technologies (ICALT'08)*, pages 241–245. IEEE, 2008.

[46] Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. Case studies for method and tool evaluation. *IEEE software*, 12(4):pages 52–62, 1995.

[47] Yan Li, Boqin Feng, and Qinjiao Mao. Research on path completion technique in web usage mining. In *Proceedings of the International Symposium on Computer Science and Computational Technology (ISCSCT'08)*, volume 1, pages 554–559. IEEE, 2008.

[48] Bing Liu. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data (Data-Centric Systems and Applications)*. Springer, 2006.

[49] Bruce A Mah. An empirical model of http network traffic. In *Proceedings of the Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution. (INFOCOM'97)*, volume 2, pages 592–600. IEEE, 1997.

[50] Roy Fielding Mark Nottingham and Julian Reschke. Hypertext transfer protocol (http/1.1): Caching. `https://tools.ietf.org/html/rfc7234`, 2014. Accessed: 2017-09-08.

[51] Ali Mesbah, Engin Bozdag, and Arie Van Deursen. Crawling ajax by inferring user interface state changes. In *Proceedings of the Eighth International Conference on Web Engineering (ICWE'08)*, pages 122–134. IEEE, 2008.

[52] James W Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for javascript applications. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI '10)*, volume 10, pages pages 159–174, 2010.

[53] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[54] Bamshad Mobasher, Robert Cooley, and Jaideep Srivastava. Automatic personalization based on web usage mining. *Communications of the ACM*, 43(8):pages 142–151, 2000.

[55] Ali Moosavi Byooki. Component-based crawling of complex rich internet applications, 2014. `https://ruor.uottawa.ca/bitstream/10393/30636/3/Moosavi_Byooki_Seyed_Ali_2014_thesis.pdf`.

[56] Michal Munk, Jozef Kapusta, and Peter Švec. Data preprocessing evaluation for web log mining: reconstruction of activities of a web visitor. *Procedia Computer Science*, 1(1):pages 2273–2280, 2010.

[57] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Recording application-level execution for deterministic replay debugging. *IEEE Micro*, (1):pages 100–109, 2006.

[58] Christopher Neasbitt, Roberto , Kang Li, and Terry Nelms. Clickminer: Towards forensic reconstruction of user-browser interactions from network traces. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1244–1255. ACM, 2014.

[59] JinSeok Oh, Jin-woo Kwon, Hyukwoo Park, and Soo-Mook Moon. Migration of web applications with seamless execution. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 173–185. ACM, 2015.

[60] JinSeok Oh and Soo-Mook Moon. Snapshot-based loading-time acceleration for web applications. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 179–189. IEEE Computer Society, 2015.

[61] Junghoon Oh, Seungbong Lee, and Sangjin Lee. Advanced evidence collection and analysis of web browser activity. *digital investigation*, 8:pages S62–S70, 2011.

[62] Saroj K Pani, L Panigrahy, VH Sankar, Bikram Keshari Ratha, AK Mandal, and SK Padhi. Web usage mining: a survey on pattern extraction from web logs. *International Journal of Instrumentation, Control & Automation*, 1(1):pages 15–23, 2011.

[63] Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral clustering of HTTP-based malware and signature generation using malicious network traces. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI '10)*, volume 10, pages 391–404, 2010.

[64] Emmanuel S Pilli, Ramesh C Joshi, and Rajdeep Niyogi. Network forensic frameworks: Survey and research challenges. *Digital Investigation*, 7(1):pages 14–27, 2010.

[65] Sriram Raghavan and SV Raghavan. Reconstructing tabbed browser sessions using metadata associations for multi-threaded browser implementation. In *Proceedings of the Twelfth Annual IFIP WG 11.9 International Conference on Digital Forensics*, pages 165–188. Springer, 2016.

[66] Yasushi Saito. Jockey: a user-space library for record-replay debugging. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, pages 69–76. ACM, 2005.

[67] Sreedevi Sampath. Advances in user-session-based testing of web applications. *Advances in Computers*, 86:pages 87–108, 2012.

[68] Davide Sangiorgi and David Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge university press, 2003.

[69] Fabian Schneider, Anja Feldmann, Balachander Krishnamurthy, and Walter Willinger. Understanding online social network usage from a network perspective. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 35–48. ACM, 2009.

[70] Hongzhou Sha, Tingwen Liu, Peng Qin, Yong Sun, and Qingyun Liu. Eplogcleaner: improving data quality of enterprise proxy logs for efficient web usage mining. *Procedia Computer Science*, 17:pages 812–818, 2013.

[71] F Donelson Smith, Félix Hernández Campos, Kevin Jeffay, and David Ott. What tcp/ip protocol headers can tell us about the web. In *ACM SIGMETRICS Performance Evaluation Review*, volume 29, pages 245–256. ACM, 2001.

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

[73] Jaideep Srivastava, Robert Cooley, Mukund Deshpande, and Pang-Ning Tan. Web usage mining: Discovery and applications of usage patterns from web data. *ACM SIGKDD Explorations Newsletter*, 1(2):pages 12–23, 2000.

[74] Mitali Srivastava, Rakhi Garg, and PK Mishra. Analysis of data extraction and data cleaning in web usage mining. In *Proceedings of the 2015 International Conference on Advanced Research in Computer Science Engineering & Technology (ICARCSET 2015)*, pages 13:1–13:6. ACM, 2015.

[75] K Vikram, Abhishek Prateek, and Benjamin Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 173–186. ACM, 2009.

[76] Claes Wohlin, Martin Höst, and Kennet Henningsson. Empirical research methods in web and software engineering. *Web engineering*, pages pages 409–430, 2006.

[77] Guowu Xie, Marios Iliofotou, Thomas Karagiannis, Michalis Faloutsos, and Yaohui Jin. Resurf: Reconstructing web-surfing activity from network traffic. In *2013 IFIP Networking Conference*, pages 1–9. IEEE, 2013.

[78] Min Xu, Rastislav Bodik, and Mark D Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 122–133. IEEE, 2003.

[79] Robert K Yin. *Case study research: Design and methods*. Sage publications, 2013.