WaRR: A Tool for High-Fidelity Web Application Record and Replay

Silviu Andrica and George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{silviu.andrica, george.candea}@epfl.ch

***Abstract*—We introduce WaRR, a tool that records and replays with high fidelity the interaction between users and modern web applications. WaRR consists of two independent components: the WaRR Recorder and the WaRR Replayer.**

The WaRR Recorder is embedded in a web browser, thus having access to user actions, and provides a complete interaction trace—this confers high recording fidelity. The WaRR Replayer uses an enhanced, developer-specific web browser that enables realistic simulation of user interaction—this confers high replaying fidelity.

We describe two usage scenarios for WaRR that help developers improve the dependability of web applications: testing web applications against realistic human errors and generating user experience reports. WaRR helped us discover bugs in widely-used web applications, such as Google Sites, and offers higher recording fidelity compared to current tools.

***Keywords*-web applications; record & replay; testing;**

I. Introduction

Web applications are becoming pervasive as users rely increasingly more on applications that are accessed through a web interface rather than on shrink-wrapped software. One of the most widely used web applications is e-mail, such as GMail [1]. With Google Docs, even application suites that traditionally have been running on a user's computer are moved to the cloud and accessed through web browsers.

Testing and debugging modern web applications requires a holistic approach, to include client-side code in addition to server-side code. Modern web applications are distributed across back-end servers and front-end clients, with some of the functionality being offloaded to clients. Alas, tools that focus on client-side code are missing from an adequate set of tools available to web application developers.

In this paper, we present a tool that helps developers test and debug client-side code easily and efficiently. For server-side code, there exist several promising techniques for testing, such as symbolic execution [2], [3] and execution synthesis [4], and for debugging, such as output-deterministic replay [5]. Finally, we do not focus on bugs triggered by browser differences or network errors.

Testing and debugging a web application with realistic usage scenarios requires high-fidelity record and replay of the interactions between users and the application, because users drive its behavior. Recording fidelity quantifies recorded interactions, and high-fidelity recording requires that all interactions be recorded. Replaying fidelity quantifies correctly played back interactions, and high-fidelity replaying requires that all interactions be realistically simulated.

High-fidelity record and replay is challenging because the client-side code can dynamically change the content of a web page. Modern web applications achieve a high level of sophistication, by using complex client-side JavaScript code that vastly extends the range of possible interactions between users and web applications, beyond merely clicking on links. Nowadays, the HTML pages of a web application are mere containers whose contents change in reaction to user events (i.e., the HTML code of a page pointed to by a URL can dramatically differ in time). Moreover, record and replay tools need always be recording, without hurting user experience, so that users can submit complete bug reports.

Current record-and-replay tools that target web applications, such as Selenium IDE [6], have low fidelity. Selenium IDE yields incomplete user interaction traces, fails to trigger event handlers associated to a user action, and must be explicitly installed by users. Since users do not expect bugs, they are unlikely to use Selenium IDE at all times, so bug-triggering user interactions will be missed. An incomplete trace precludes developers from reproducing and fixing the bug and, ultimately, hurts a web application's dependability.

We introduce WaRR, an "always-on," high-fidelity record-and-replay tool for interactions between users and modern web applications. Throughout this paper, a user interaction with a web application (or simply, a user action) denotes a mouse click, a UI-element drag, or a keystroke performed by a user. To the best of our knowledge, WaRR is the first tool that can be used to record and replay interactions between users and complex web applications, such as drag-and-drop, writing emails in GMail, editing spreadsheets in Google Docs, or editing web pages in Google Sites.

WaRR uses a novel architecture, with the recording functionality being an integral part of a web browser. This design decision brings five advantages. First, WaRR has access to a user's every click and keystroke, thus providing high-fidelity recording. Second, WaRR's recorder requires no modification to web applications, being easy to employ by developers. Third, the recorder has access to the actual HTML code that will be rendered, after code has been dynamically loaded. Fourth, it can easily be extended to

978-1-4244-9233-6/11/$26.00 ©2011 IEEE

record various sources of nondeterminism (e.g., timers). Fifth, since the recorder is based on the web browser engine WebKit [7], which is used by a plethora of browsers, it can record user interactions across a large number of platforms.

WaRR's replaying functionality leverages the observation that a web browser used for web application debugging can be less restricted than one for regular users. Thus, it is reasonable to expect developers' browsers to have additional features compared to users' browsers. For example, while normal WebKit-based web browsers prevent setting certain properties of JavaScript events, this restriction can be lifted during testing and debugging. Hence, WaRR's replayer can correctly trigger JavaScript events (e.g., `onKeyPress`) and ensure that the associated event handlers run correctly.

We illustrate how WaRR helps web application developers improve their application's dependability with two case studies: human error testing and automatic generation of user experience reports. For testing, we use known error models to inject realistic human errors into traces gathered by WaRR and then replay the generated traces and observe how the web application handles the injected errors.

As a standalone tool, WaRR can reproduce bugs triggered by a sequence of user actions, but it is more useful to pair WaRR with server-side debugging aids, when a bug involves nondeterminism (e.g., wrongful handling of concurrent clients' session data). Existing server-side debugging aids, such as [8], can complement WaRR and together provide a solution to debug modern web applications.

We implement WaRR, evaluate its record-and-replay fidelity, explore its bug-finding abilities, and measure the impact WaRR's recording functionality has on user experience. Our evaluation shows that: (1) WaRR has a higher recording fidelity than Selenium IDE, (2) by using WaRR, we were able to find a bug in Google Sites, and (3) WaRR's recording functionality induces an overhead below human perception levels and can, therefore, be kept running continuously.

The rest of the paper is organized as follows: Section II reviews previous approaches to record and replay, Section III presents WaRR's design, and Section IV details its implementation. Sections V and VI describe two usage scenarios for WaRR. Section VII concludes the paper.

II. Background

The field of record and replay has received a lot of attention, both in the research community and in industry. Below, we describe the main approaches to recording user interaction and subsequently replaying it.

One way to record user interaction with a web application is to log all network traffic that occurs between a web browser and an application server, as in Fiddler [9], a proxy that logs HTTP(S) traffic. One can then replay recorded traffic. Alas, when analyzing recorded traffic, one cannot distinguish between requests made in response to user interaction versus requests made by a web page while loading. Disambiguating these two types of requests is difficult. Therefore, such tools are of little help in debugging client-side code. WaRR, on the other hand, records the user actions that cause network traffic, not the network traffic itself.

One can use proxies to inject JavaScript code into HTML pages to track user interaction, as in Mugshot [10] and UsaProxy [11]. These approaches have two limitations. First, they can instrument only HTML pages, because they cannot identify HTML or JavaScript code in non-HTML server responses. Second, using proxies requires breaking the end-to-end security enforced by HTTPS, because proxies need to intercept server responses, thus creating the possibility of leaking private information. In contrast, WaRR has access to the processed and decrypted HTML code of a web application and logs user actions on the user's machine.

By modifying the operating system, tools such as RUI [12] and AppMonitor [13] can record a user's every keystroke and mouse click. Recording accurate traces requires application support to precisely identify the UI element a user acted upon. Removing such support can hinder replay accuracy. Finally, such tools cannot easily be ported to various operating systems (OSes). Since WaRR is based on a browser engine, it has enough information about a user action's target to provide accurate traces without OS support.

Virtual machines can be used to record and replay user interaction. In this approach, an entire OS runs inside a virtual machine, which captures an execution and enables developers to replay it later [14], [8]. However, the incurred performance overhead hurts user experience. When debugging, this approach is time-consuming, because developers must step through machine-level instructions. WaRR is different because it enables stepping through user-level actions.

There exist tools that run as browser plug-ins, [15], [16], [6], [17], and can record a user's clicks and keystrokes, but lack fidelity or are bound to a platform. Selenium [18] is a testing framework designed for web applications and offers record-and-replay functionality, but misses user actions when recording complex web pages [19]. WET [20] is a web automation testing tool, bound to Internet Explorer, that uses Watir [17] to drive browser interaction. WaRR, on the other hand, can record the interaction between users and arbitrarily complex web applications and run on multiple platforms.

There exist commercial solutions, like LoadRunner [21] and SilkPerformer [22], for testing distributed systems, but they require special software to be installed on the client's computer. Such tools offer recording, load generation, monitoring, and diagnostics for both client-side and server-side code. In contrast, WaRR targets client-side code only and requires users to use a fully-functional, but custom browser.

WaRR extends the relevancy of tools like DoDOM [23]. DoDOM infers DOM (Document Object Model) invariants and uses them in tests to detect errors, but is limited to web applications that use HTTP. WaRR can aid DoDOM test also HTTPS applications, because WaRR can replay the

interaction between a user and any type of web application.

III. WaRR Design

WaRR's design goal is to record and replay interactions between users and web applications with low overhead and sufficient fidelity to make WaRR suitable for testing and debugging modern web applications.

WaRR targets both developers and web application users. Users only require recording, to submit comprehensive bug reports, while developers use recording for testing, and replaying for testing and debugging.

WaRR consists of two independent components: a recorder, targeting users and developers, and a replayer, targeting developers only. The WaRR Recorder is a modified web browser that captures the interactions between users and web applications. The WaRR Replayer uses a different, custom browser and a browser interaction driver to simulate user interaction, based on recorded traces. Figure 1 provides an overview of WaRR's architecture.

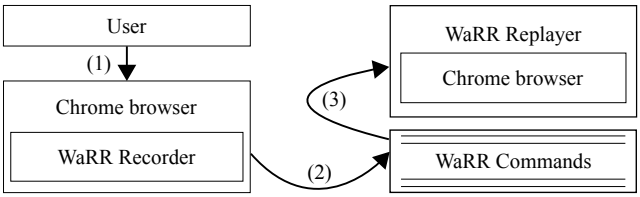

Figure 1: WaRR architecture. The WaRR Recorder captures user actions (1), logs them as WaRR Commands (2), and the WaRR Replayer plays back the recorded commands (3).

In the rest of this section we describe the two WaRR components and highlight key design decisions.

A. The WaRR Recorder

The WaRR Recorder is meant to be a recording solution that offers high fidelity, is lightweight, always-on, and does not require user setup. High fidelity requires that all user actions be recorded, such that the interaction trace is complete. A lightweight solution does not affect user experience. An always-on, no setup solution ensures that, if a bug manifests, the bug-triggering interaction is always available.

Our solution is to embed the recording logic deep inside a web browser. Alternative designs, such as logging HTTP traffic, are being made obsolete by increased HTTPS deployment, and there even exist proposals to replace parts of HTTP and augment it with SPDY [24]. HTML usage, on the other hand, will likely become more widespread, especially if we consider the increased usage of handheld devices whose power consumption requirements impede the use of competing technologies, such as Flash.

The WaRR Recorder extends WebKit and is embedded into the Chrome web browser. Figure 2 shows Chrome's architecture for displaying web pages. As described in [25], *WebKit* is the rendering engine, *Renderer* proxies messages across process boundaries, *Tab* represents a web page, and *Browser window* contains all the opened Tabs.

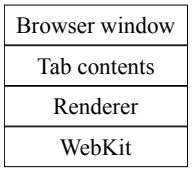

Figure 2: Simplified Chrome architecture.

Even though our design requires browser changes, this brings an important advantage. Being based on WebKit, the WaRR Recorder can capture user interactions on more platforms than any other web application record-and-replay tool, because WebKit is used for desktop browsers like Chrome and Safari, and for the default browsers of mobile platforms such as iOS, Android, and WebOS. Thus, WaRR enables developers to test web applications with realistic usage scenarios, originating from varied usage contexts.

The WaRR Recorder outputs a sequence of WaRR Commands, where each command is a user action. Section IV-B further describes WaRR Commands.

B. The WaRR Replayer

The WaRR Replayer is the counterpart to the WaRR Recorder and simulates a user interacting with a web application as specified by WaRR Commands.

The WaRR Replayer has two main components: a browser interaction driver and a browser. The driver reads WaRR Commands and converts them into commands sent to the browser. This design enables the driver to use any browser, given a suitable API to drive interaction with that browser. Section IV-C provides implementation details.

IV. WaRR Implementation

This section describes the implementation of the WaRR Recorder and Replayer, the format of WaRR Commands, and discusses some of WaRR's limitations. We present the main challenges we faced and how we addressed them.

A. The WaRR Recorder

The recorder is located at Chrome's WebKit layer, because it provides the ideal opportunity to record user actions: when a mouse button is clicked or a key is pressed, this event arrives at the WebKit layer to be dispatched to the appropriate HTML element. Figure 3 shows parts of the stack trace when handling such events.

The WaRR Recorder captures three types of user actions: mouse clicks, UI-element drags, and keystrokes. We implement the WaRR Recorder by adding calls to the recorder's logging functions in three methods of the `WebCore::EventHandler` class: `handleMousePressEvent`, `handleDrag`, and `keyEvent`. The changes amount to less than 200 lines of C++ code.

```
WebCore::EventHandler::handleMousePressEvent
WebKit::WebViewImpl::handleInputEvent
RenderView::OnMessageReceived
IPC::ChannelProxy::Context::OnDispatchMessage
DispatchToMethod
MessageLoop::Run
ChromeMain
main
```

Figure 3: Fragment of the stack trace generated when performing a mouse click in Chrome.

A benefit of our choice of implementation layer is that, if necessary, other events of interest can easily be monitored, requiring only slight modifications to the WaRR Recorder. For example, we initially did not record drag events, but adding support for them took one person less than one day.

As we show in Section VI, our implementation incurs negligible overhead and does not affect user experience.

The recorder exports each interaction between a user and a web application as a WaRR Command, described next.

B. WaRR Commands

A WaRR Command contains the type of an action (i.e., `click`, `doubleclick`, `drag`, and `type`), an identifier of the HTML element that was acted upon, information specific to the action's type, and the time elapsed since the previous action. Figure 4 shows a sequence of WaRR Commands that have been slightly edited for readability.

```
click //div/span[@id="start"] 82,44  1
type  //td/div[@id="content"] [H,72] 3
type  //td/div[@id="content"] [e,69] 4
type  //td/div[@id="content"] [l,76] 7
type  //td/div[@id="content"] [l,76] 9
type  //td/div[@id="content"] [o,79] 11
type  //td/div[@id="content"] [ ,32] 12
type  //td/div[@id="content"] [w,87] 15
type  //td/div[@id="content"] [o,79] 17
type  //td/div[@id="content"] [r,82] 19
type  //td/div[@id="content"] [l,76] 23
type  //td/div[@id="content"] [d,68] 29
type  //td/div[@id="content"] [!,49] 31
click //td/div[text()="Save"] 74,51  37
```

Figure 4: Fragment of the sequence of WaRR Commands recorded by WaRR while editing a Google Sites web page.

HTML elements that are the target of an action are identified by XPath [26] expressions. XPath is a language for locating an element in an XML/HTML document, by specifying a set of properties of that element or by specifying how to reach it from one of its ancestors. For example, `//td/div[@id="content"]` denotes an element of type `div` that is a child of an element of type `td` and has the property `id` set to `content`. For a single HTML element, there can be multiple XPath expressions, and various HTML elements may correspond to the same XPath expression.

Click-related WaRR Commands indicate the position in the web browser window where a click originated, as backup element identification information. The `drag` command indicates the difference in the dragged element's position.

The `type` command provides a string representation of a typed key and its ASCII code. When typing capital letters using the `Shift` key in Chrome, the browser registers two keystrokes: one for the `Shift` key and one for the printable key. Logging the event of pressing `Shift` is unnecessary, so we only log the combined effect. However, since other control keys, such as `Control`, do not always lead to new characters being typed, we log their ASCII codes.

C. The WaRR Replayer

The WaRR Replayer is based on WebDriver [27] and ChromeDriver [28], and the Chrome browser.

High-fidelity replay is hard to achieve in browsers based on WebKit, because they make certain properties of JavaScript events read-only. This prevents event handlers associated to such events from running with correct parameters, thus damaging replay fidelity. Since the WaRR Replayer is targeted at developers, its browser need not obey such restrictions. We modify Chrome to enable setting properties of the `KeyboardEvent` JavaScript event, making such events practically indistinguishable from those generated by users.

WebDriver is a browser interaction automation tool that controls various browsers through a common API, while ChromeDriver is a WebDriver implementation tailored to Chrome. WebDriver provides functionality for clicking, dragging, and entering text. Therefore, one should be able to use ChromeDriver to successfully replay WaRR Commands.

Chrome is controlled through a ChromeDriver plug-in composed of a master and multiple ChromeDriver clients, one for each `iframe` in an HTML document. The master acts as a proxy between the ChromeDriver clients and the rest of ChromeDriver/WebDriver. Clients receive commands and execute them on the `iframe` they are responsible for. At any point in time, only one client executes commands.

When replaying WaRR Commands, the main challenge we faced was the replay of interactions with web applications where an HTML element's properties differ between record time and replay time. The recorded XPath expression became invalid, the WaRR Replayer failed to find the required element, and the corresponding WaRR command could not be replayed. For example, whenever GMail loaded, it generated new `id` properties for HTML elements.

To mitigate this problem, the WaRR Replayer employs an automatic, application-independent, and progressive relaxation of an element's XPath expression. However, the tool first assumes an application's HTML structure is constant, to provide timing-accurate interaction replay, and tries to use the recorded XPath. If this expression is invalid, WaRR progressively simplifies the expression to find a matching element. This automatic simplification is guided by heuristics that remove XPath attributes (e.g., `id`), maintain only certain attributes (e.g., only `name`), or discard a prefix of an

XPath expression (e.g., changing `//td/div[@id="id1"]` to `//div[@id="id1"]`).

For WaRR, a web application's DOM is free to extensively change between the time of recording and that of replay. To replay a user action on an HTML element, the WaRR Recorder only requires some of the DOM properties in close vicinity of that element to be preserved.

The next major challenge we faced was ChromeDriver's incomplete functionality.

First, ChromeDriver lacks support for double clicks. It is important for WaRR to be able to replay double clicks, because web applications that use them, such as Google Docs, are increasingly popular. We add double clicking support by using JavaScript to create and trigger the necessary events.

Second, ChromeDriver does not handle text input properly. When simulating keystrokes into an HTML element, ChromeDriver sets that element's `value` property. This property exists for `input` and `textarea` elements, but not for other elements (e.g., `div`, a container-like element). We fix this issue by setting the correct property (e.g., `textContent` for `div` elements) as the target of received keystrokes and triggering the required events.

The third major replay challenge we encountered was improper support for `iframe`s, and it involved Chrome and ChromeDriver. An `iframe` allows an HTML document to embed another one. First, one cannot execute commands on `iframe`s that lack the `src` property, because Chrome does not load ChromeDriver clients for them. We solve this issue by having the ChromeDriver client of the parent HTML document execute commands on such `iframe`s. Second, ChromeDriver provides no means to switch back to an `iframe`. We use a custom `iframe` name to signal a change to the default `iframe` and implement the necessary logic.

The last major challenge was ChromeDriver becoming unresponsive, when a user changed web pages. Chrome unloads the ChromeDriver clients corresponding to the `iframe`s of the old page and loads new clients for the new page's `iframe`s. The ChromeDriver master keeps track of the active client, the one executing commands, and when it is unloaded, a new active client is selected. The selection is based on an assumed order of loads and unloads, but Chrome does not ensure this order, and a new active client may not be chosen. Therefore, new commands will not be executed, and the replay will halt. We fix this issue by ensuring that unloads do not prevent selecting a new active client.

D. WaRR Limitations

WaRR records all keystrokes, therefore also potentially sensitive information, such as passwords and usernames. While this paper does not address privacy issues, we envision a solution in which users share recorded traces with a web application's developers after they removed sensitive information. If concerns still arise, one can take an approach similar to [29] to generate anonymized user interaction traces that lead the application along the same execution path. To prevent traces from being used to exploit an application's vulnerabilities, one can encrypt them with the developers' public key, so that only developers can access the traces.

WaRR cannot handle pop-ups because user interaction events that happen on such widgets are not routed through to WebKit. A solution we are considering is to insert logging functionality in the browser code that handles pop-ups.

WaRR offers a single user's perspective of how a bug was triggered, but this can be insufficient for reproducing a bug involving concurrent clients. However, if users use WaRR, developers have access to all the actions users performed. Alas, the traces do not contain the timing dependencies between various users' actions.

WaRR cannot control the environment it runs in and, therefore, cannot ensure that event handlers triggered by user actions finish in the same amount of time, during replay, as they did during recording, possibly hurting replay accuracy.

For a complete debugging solution, WaRR should be coupled with server-side debugging aids. As a standalone tool, WaRR can help developers debug bugs that are always triggered by a predefined sequence of user actions. To aid debugging bugs triggered by events other than user actions or ones that manifest only in a particular state of the entire web application, WaRR must be complemented by server-side aids. Such aids span from simple logs to sophisticated techniques such as execution synthesis [4] and output-deterministic replay [5].

V. WebErr: Testing Web Applications Against Human Errors

After describing WaRR, we now describe two tools we built on top of it. The first one, named WebErr, tests web applications against realistic human errors. The second one, called AUsER, automatically generates user experience reports. We begin by describing WebErr.

Bugs triggered by human errors have high chances of manifesting in production, because human error is pervasive. Studies show users commit more than 14 errors in one web application interaction session [30]. WebErr is a tool that tests web applications against such human errors.

Figure 5 depicts how WebErr works: it records the interaction between a user and a web application as a trace (Step 1), then injects realistic human errors into this trace (Steps 2 and 3), and then uses the WaRR Replayer to test that web application against the modified interaction traces (Step 4).

To simulate realistic human errors, we use models of how users interact with web applications and what type of errors they commit. We focus on two categories of user errors, navigation errors and timing errors, that together cover a significant part of the observed errors [30]. Navigation errors lead to incorrect interaction sequences [31], while timing errors occur when users interact with a web application "at a bad time" (e.g., before the application finished loading).

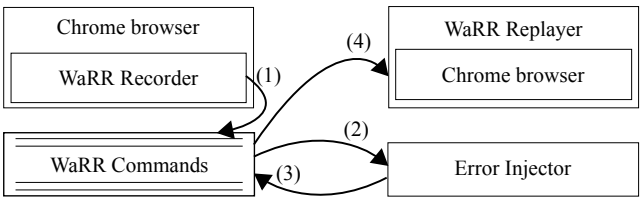

Figure 5: Testing web applications against human errors.

A. Testing Web Applications Against Navigation Errors

Navigation errors manifest as deviations from a correct pattern of interaction with a web application. If we consider a pattern of interaction to be a sequence of steps, then the errors we are interested in are: forgetting, reordering, and substitution of steps. Typos, clicking the wrong button or link, and selecting another item from a drop-down list are examples of such errors that are common in practice.

One approach to test web applications against navigation errors is to apply all possible combinations of the above errors to a trace and test with the resulting traces. Although able to detect all bugs, this approach is impractical, because of the large number of possible tests and the low probability of discovering bugs by injecting errors into unrelated steps.

For example, from a trace of 100 WaRR Commands corresponding to filling in two text fields, one can generate $permutations(100) = 100!$ new traces, considering only step-reordering errors, yet tests that alternatively fill in letters of each field have low bug-detection power.

We employ an approach that uses a grammar expressing a correct pattern of interaction, confines error injection to a reduced number of this grammar's rules, and never performs cross-rule error injection. We view an interaction step as a grammar rule and simulate forgetting a step by making a rule have no productions, step reordering by reordering a rule's right-hand side productions, and substitution of steps by substituting a rule's right-hand side productions with others.

For example, suppose that editing a web page is defined by the grammar $EditSite \rightarrow Authenticate\ Edit$, where the right-hand side productions are expressed by other rules. After injecting a step-reordering error into this grammar, we obtain the erroneous grammar $EditSite' \rightarrow Edit\ Authenticate$.

After obtaining erroneous grammars, we generate erroneous user interaction traces, by recursively applying the rules of these grammars, and replay these traces to test web applications. Our approach requires an oracle to conclude whether the application behaved correctly, a common practice in automated testing and debugging techniques [32].

We now describe how we define a user interaction grammar. We follow the process of how humans solve tasks: an initial task is split into subtasks, and these subtasks are then performed one by one [33]. Subtasks are recursively split into other subtasks until they can be performed directly in a web application (e.g., click on a particular link, type a key). Doing so yields a task tree expressible by a grammar. Figure 6 depicts such a tree for the case of editing a website.

Since user interaction grammars do not readily exist, and we have no semantic information for WaRR Commands, we face the challenge of having to infer such grammars given only a sequence of WaRR Commands. We aim to cluster WaRR Commands in a way that reconstructs, as much as possible, the task tree followed by the user.

We employ an algorithm to cluster WaRR Commands based on web page similarity. The insight is that different web pages denote different subtasks, and when consecutive web pages differ, a subtask finished and another one started. For each WaRR Command, the algorithm compares the web page generated by replaying that command against web pages generated by previous commands. The command that generated the most similar web page becomes the parent of the current WaRR command. Computing the similarity of web pages is based on their DOM shape, taking into account the type of the HTML elements and their `id` property.

As described so far, the algorithm generates a tree with three levels: one for the initial WaRR Command, one for commands that change the URL, and one for the rest of the commands. Alas, this tree does not clearly distinguish between unrelated subtasks, leading to the generation of tests that have little bug-finding power. To add more depth, we tune the algorithm to spawn new tree nodes whenever the interaction changes from one HTML element to another one.

Since a deep task tree can still generate an impractically large number of interaction traces, we propose two heuristics to reduce this number. First, if a trace cannot be successfully replayed, we remove all traces that have as prefix the WaRR Commands replayed so far, because neither them can be successfully replayed. Second, we focus error injection toward only some of the grammar rules.

B. Testing Web Applications Against Timing Errors

Timing errors are caused by users who interact with web applications while the latter are not yet ready to handle user interaction. These errors occur because, although applications display wait messages, users disregard them. Hence, we consider them to be user errors. The advent of Asynchronous JavaScript And XML (AJAX) [34], which enables asynchronous browser-server communication, made web applications more vulnerable to timing errors.

To simulate timing errors, we modify the delay between replaying consecutive WaRR Commands. We stress test web applications by replaying commands with no wait time.

Since web applications are dynamically loaded, one can envision more sophisticated testing techniques, such as replaying events before and after new code has been downloaded. We leave such techniques to future work.

C. WebErr In Practice

We applied WebErr to two usage scenarios: First, we tested how well web search engines detect and fix typos

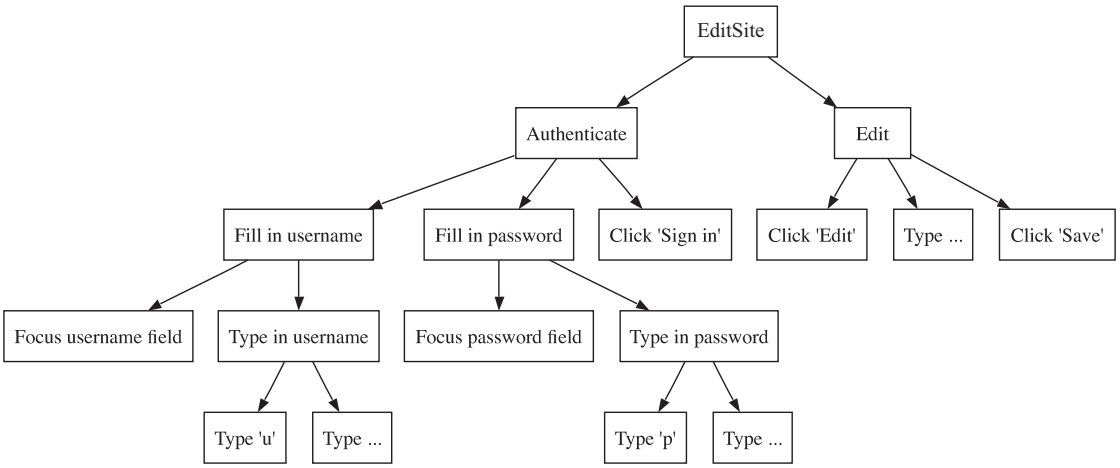

Figure 6: A task tree for editing a website.

present in search queries. Second, we injected timing errors while editing a website using Google Sites.

Web search engines are widely used web applications that must tolerate one of the most common user errors, typos in search queries. We want to test how well three web search engines, Google, Bing, and Yahoo!, handle such typos, so we choose 186 frequent queries, from New York Times's top search keywords and Google Trends's list of top searches. Next, we inject a typo into each search query, perform the searches, and measure the number of errors detected by each search application. Table I presents the results.

Search engine	**Google**	**Bing**	**Yahoo!**
Percentage	100%	59.1%	84.4%

Table I: The percentage of query typos detected and fixed by the Google, Bing, and Yahoo! web search engines.

We tested Google Sites, a web hosting solution that enables users to edit web sites using a rich web application, against timing errors and found a bug. When editing a Google Sites website, one has to wait for the editing functionality to load. In our experiment, we simulated impatient users who do not wait long enough and perform their changes right away. In doing so, we caused Google Sites to use an uninitialized JavaScript variable, an obvious bug.

VI. AUsER: Automatic User Experience Reports

AUsER is a tool that automatically generates user experience reports. If a user experiences a bug while using a web application, she presses a button in AUsER, and the developers of that application receive the sequence of WaRR Commands she performed. Being based on WaRR, AUsER offers high-fidelity recording and replaying, thereby reducing developer effort when reproducing the problem.

Since most bugs are hard to detect automatically [35], AUsER allows users to provide a textual description of the bug and a snapshot of the final web page in which the bug manifests. AUsER allows users to send developers only a part of the snapshot, such as the button that has the wrong name, leaving out private details displayed on the web page.

In order to be practical, AUsER must not hinder a user's interaction with web applications. The runtime overhead introduced by the WaRR Recorder must be below the 100 ms human perception threshold [36]. We run an experiment, consisting of writing an email in GMail, to compute the time required by the WaRR Recorder to log each user action. The average required time is on the order of hundreds of microseconds and does not hinder user experience.

High-fidelity recording is critical for AUsER, so we want to compare the recording fidelity of the WaRR Recorder and Selenium IDE. For our experiment, we focus on four widely used web applications: Google Sites, GMail, the Yahoo! web portal, and Google Docs. We choose these applications because they are representative of modern web applications. Results are presented in Table II and show that the WaRR Recorder offers higher fidelity than Selenium IDE.

Application	**Scenario**	**WaRR Recorder**	**Selenium IDE**
Google Sites	Edit site	C	P
GMail	Compose email	C	P
Yahoo	Authenticate	C	C
Google Docs	Edit spreadsheet	C	P

Table II: The completeness of recording user actions using the WaRR Recorder and Selenium IDE. In this table, C stands for Complete, and P stands for partial.

VII. Conclusions

We presented WaRR, an "always-on" tool that records and replays with high fidelity the interactions between a user and a modern web application. WaRR achieves high recording

fidelity due to the interaction-recording functionality being deeply embedded in the web browser, thus having direct access to user keystrokes and clicks.

We envision two usage scenarios for WaRR: testing web applications against realistic human errors and generating user experience reports. We expect WaRR to help developers overcome the challenges associated with writing dependable modern web applications.

Our evaluation shows that WaRR incurs low runtime overhead during recording, offers higher fidelity than similar tools, and can find bugs in real modern web applications.

REFERENCES

[1] "Top ranking applications (Wakoopa)," http://www.favbrowser.com/top-ranking-applications-wakoopa/, 2009.

[2] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Symp. on Operating Systems Design and Implementation*, 2008.

[3] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *ACM SIGOPS/EuroSys European Conf. on Computer Systems*, 2011.

[4] C. Zamfir and G. Candea, "Execution synthesis: A technique for automated debugging," in *ACM SIGOPS/EuroSys European Conf. on Computer Systems*, 2010.

[5] G. Altekar and I. Stoica, "ODR: Output-deterministic replay for multicore programs," in *Symp. on Operating Systems Principles*, 2009.

[6] "Selenium IDE," http://seleniumhq.org/projects/ide.

[7] "WebKit," http://www.webkit.org/.

[8] G. W. Dunlap, D. Lucchetti, P. M. Chen, and M. Fetterman, "Execution replay on multiprocessor virtual machines," in *Intl. Conf. on Virtual Execution Environments*, 2008.

[9] "Fiddler," http://www.fiddler2.com/fiddler2/.

[10] J. Mickens, J. Elson, and J. Howell, "Mugshot: Deterministic capture and replay for JavaScript applications," in *Symp. on Networked Systems Design and Implementation*, 2010.

[11] R. Atterer, M. Wnuk, and A. Schmidt, "Knowing the user's every move - user activity tracking for website usability evaluation and implicit interaction," in *Intl. World Wide Web Conference*, 2006.

[12] U. Kukreja, W. Stevenson, and F. Ritter, "RUI: Recording user input from interfaces under Windows and Mac OS X," in *Behavior Research Methods*, 2006.

[13] J. Alexander, A. Cockburn, and R. Lobb, "AppMonitor: A tool for recording user actions in unmodified windows applications," in *Behavior Research Methods*, 2008.

[14] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen, "ReVirt: Enabling intrusion analysis through virtual-machine logging and replay," in *Symp. on Operating Systems Design and Implementation*, 2002.

[15] "iMacros," www.iopus.com/iMacros/.

[16] "Test4Gen," https://addons.mozilla.org/en-US/firefox/addon/testgen4web/.

[17] "Watir," http://watir.com.

[18] "Selenium," http://seleniumhq.org/.

[19] "Selenium FAQ," http://web.archive.org/web/20080822230502/http://wiki.openqa.org/display/SIDE/FAQ.

[20] "WET," http://www.wet.qantom.org/.

[21] "HP LoadRunner," http://learnloadrunner.com/.

[22] "SilkPerformer," http://www.borland.com/us/products/silk/silkperformer/.

[23] K. Pattabiraman and B. Zorn, "DoDOM: Leveraging DOM invariants for Web 2.0 application reliability," Microsoft Research, Tech. Rep. MSR-TR-2009-176, 2009.

[24] "SPDY," http://www.chromium.org/spdy.

[25] "Chrome's rendering architecture," http://www.chromium.org/developers/design-documents/displaying-a-web-page-in-chrome.

[26] "XML path language (XPath)," http://w3.org/TR/xpath.

[27] "WebDriver," http://google-opensource.blogspot.com/2009/05/introducing-webdriver.html.

[28] "ChromeDriver," http://code.google.com/p/selenium/wiki/ChromeDriver.

[29] M. Castro, M. Costa, and J.-P. Martin, "Better bug reporting with better privacy," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[30] S. D. Wood and D. E. Kieras, "Modeling human error for experimentation, training, and error-tolerant design," in *The Interservice/Industry Training, Simulation & Education Conference*, 2002.

[31] S. Hallé, T. Ettema, C. Bunch, and T. Bultan, "Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines," in *Intl. Conf. on Automated Software Engineering*, 2010.

[32] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, 2002.

[33] S. K. Card, T. P. Moran, and A. Newell, *The psychology of human-computer interaction*. Lawrence Erlbaum Associates, 1983.

[34] "AJAX," http://www.w3schools.com/ajax/.

[35] P. Godefroid and N. Nagappan, "Concurrency at Microsoft – An exploratory survey," in *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[36] J. V. Forrester, *The eye: basic sciences in practice*. Elsevier Health Sciences, 2002.