Trackr: Reliable UI Element Tracking for Application Refactoring Based Mobilization of Enterprise Web Applications

Uma Parthavi Moravapalle and Raghupathy Sivakumar
Georgia Institute of Technology, Atlanta, GA

ABSTRACT

Mobilizing enterprise applications enable employees to perform their job functions on their smartphones, even when away from a Desktop. Among the different strategies used to mobilize web enterprise applications, *application refactoring* based services are gaining popularity due to their ease of deployment. Refactoring involves hosting the application as-is on the enterprise cloud and allowing a user to interact with it through a smartphone-optimized UI. These services require APIs provided by the applications to accurately map the smartphone UI to the web application UI.

Of the strategies available to APIfy applications, a front-end only approach based on intelligent screen scraping is particularly attractive as it can APIfy a host of applications without any support from the applications themselves. Front-end strategies, however, rely on being able to accurately and reliably identify UI elements within the application. In this paper, we show that simple approaches which rely on graphical coordinates or the position of an element with respect to a fixed anchor in the application layout are not robust enough for APIfication. In this context, we present *Trackr*, an algorithm that relies on the notion of *quorum fingerprinting* to track elements. We then discuss several optimizations to this baseline version. We show through analysis of changes to real-world web-applications that *Trackr* has considerable benefits. We demonstrate the usage of *Trackr* through a *Dashboard* smartphone app that monitors values across different websites within one mobile app.

KEYWORDS

mobilization, application refactoring, enterprise web applications, DOM, smartphone apps

1 INTRODUCTION

Boosting employee productivity has been the focus of enterprises in the recent past as it directly translates to increased revenue. Of the several strategies employed by the enterprises towards this end, application mobilization is a game changer. Application mobilization refers to providing employees with the ability to continue to

perform their business functions even when away from their Desktops. This is inspired by the rising adoption of smartphones among employees. Strategies to mobilize Desktop applications range from building native apps, using mobile enterprise application platforms (MEAPS) and mobile back-end as a service (mBaaS), relying on application virtualization, and employing application refactoring.

Among these strategies, mobilizing web applications through *application refactoring* [1–4] is gaining popularity, as it requires minimal development effort and lower deployment and maintenance costs from the enterprises. It involves hosting the application as-is on the enterprise network and providing an optimized view of the application on the user’s smartphone through an app (see Figure 1). While the core application logic remains intact in the enterprise network, a user is provided with a smartphone native UI through which she can interact with the web application. Any user actions on the smartphone app are translated into user actions on the web application.

Application refactoring solutions rely four key components: (i) a user friendly *configuration* process to allow the users to specify the UI elements to mobilize, (ii) a *transformation* mechanism to translate web UI elements into smartphone UI elements, (iii) a robust web UI element *tracking* framework to maintain an accurate mapping between the UI elements of the smartphone app and the UI elements of the web application, even as the application layout changes, and (iv) a *presentation* system to organize the transformed UI elements into smartphone optimized layouts. While related solutions [1, 2] have tackled the configuration, transformation and presentation aspects of application refactoring, the focus of this paper is the tracking aspect. Tracking of UI elements is crucial if the user actions from the smartphone app are to be applied on the right web UI elements. This necessitates accurately identifying a web UI element among the different elements in the web application, even when the application is subject to developer or user initiated changes. Inaccurate tracking can result in user actions being applied on the wrong web UI elements and can damage data integrity, lead to unintended application behavior or result in application failures.

Some applications provide APIs that can be leveraged by mobilization services to uniquely identify UI elements and apply actions on them. However, not all web applications provide such APIs. While applications can indeed be built ground up to support such APIs, an interesting problem is how web applications not built in such manner can still be retroactively APIfied. Such a scenario occurs under two different conditions: (i) the applications are legacy applications that pre-date the APIfy movement, but still command considerable usage ; and (ii) the applications are built by a vendor who does not have any explicit business or technology motivation to expose APIs to third party developers (even if they do exist on the backend). The second issue is more pertinent as exposing APIs for a web application does come with its own costs such as ensuring

Corresponding author: Uma Parthavi Moravapalle (parthavi@gatech.edu)

This work was funded in part by the Wayne J. Holman Endowed Chair and the National Science Foundation under grants CNS-1513884, IIP-1701115, and CNS-1813242.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MobiQuitous’18, New York City, NY, USA

© 2018 ACM. 978-1-4503-6093-7/18/11...\$15.00

DOI: 10.1145/3286978.3287013

security, incurring maintenance overheads, and provisioning for scalability. A more nuanced version of the aforementioned problem is when APIs are exposed, but are limited compared to what the underlying web application offers. In other words, the problem remains if a third-party developer needs a certain functionality offered by the web application but not exposed through an API. One approach to APIfy is to rewrite the underlying software for the web application to expose APIs. However, such a strategy incurs the burden of both the redevelopment of the software, and the redeployment of the application. Hence, the rebuilding-based strategy is an expensive process and is undesirable.

A different strategy to APIfying a web application relies on front-end only techniques to create APIs. Using a combination of automated navigation, intelligent acting, and content scraping, front-end APIfying techniques create APIs without requiring any changes whatsoever to the application backend. Consider the simple example of a thermostat app (to control a smart thermostat inside a home) that requires the current temperature for a zip code. Regardless of the APIs supported by a service such as weather.com, a front-end APIfying approach can create APIs for weather.com that will provide the current temperature for a zip code purely by navigating to weather.com, entering the zip code in the search bar, and scraping the temperature information from the resultant view. The salient advantage of this strategy is the non-dependence on backend changes. This is certainly less expensive. More importantly, APIfying an application is no longer dependent on the vendor who created the application. Third party developers can as easily create APIs for it.

It is such front-end based APIfy strategies that we consider in this paper. Specifically, such strategies rely on a fundamental building block - the ability to uniquely identify and track front-end UI elements on the web application. For example, in the smart thermostat use-case, consider that the temperature UI element is uniquely identified on the resultant view on weather.com. The thermostat app will now rely on an API that reads the temperature from that specific UI element on weather.com. What happens if the weather.com changes in a manner that impacts the temperature UI element? There are indeed changes that should break the API fi! a good example would be if weather.com removes the temperature UI element. However, there are a variety of changes including the temperature UI element moving to a different location, new UI elements introduced on the page, other (non-relevant) UI elements removed from the page, attributes of UI elements such as color, size and labels change, etc., that should not break the API. This challenge is the focus of this paper.

What makes the challenge non-trivial is that UI elements within web-applications, organized in a DOM tree, *do not have distinct permanent identifiers that remain invariant across application changes*. Thus, only relative identifiers (e.g. path from DOM tree root) can be relied upon to uniquely identify UI elements. These relative identifiers are vulnerable to even minor changes to the application that impacts the DOM tree in some manner. In this context, we present *Trackr*, a UI element tracking algorithm that improves the robustness of APIs created atop web applications multi-fold. At a high level, *trackr* uses the concept of *quorum fingerprinting* that determines the identity of a target UI element based on its relative paths from other nodes in the DOM tree that have an attribute ID.

Figure 1: Application Refactoring

We then argue why such an approach by itself remains insufficient to handle the different types of possible changes to the web application. We then present multiple optimizations to the baseline quorum fingerprinting including resilient path construction, progressive patching of fingerprints, and localized fingerprints as fail safes. We show using popular web applications such as Salesforce, a PeopleSoft application, a SharePoint application, and a Sakai application that *Trackr* can improve the identification of a target UI element multifold compared to standard mechanisms. We then present a prototype smartphone app *Dashboard* that utilizes *Trackr* to monitor values across multiple web applications within a single app.

The rest of the paper is organized as follows: Section 2 presents background and motivation for *Trackr*. Section 3 outlines the *Trackr* design. Section 4 presents evaluation results and Section 5 discusses the *Dashboard* application. Finally, Section 6 discusses the integration of *Trackr* with existing mobilization solutions, related use-cases for *Trackr*, few issues with *Trackr* and key conclusions.

2 BACKGROUND AND MOTIVATION

2.1 Refactoring based Application Mobilization

To mobilize a web application with refactoring, the enterprise hosts the application along with a refactoring server in its cloud infrastructure (Figure 1). The user is provided with a smartphone client app to view and control the application on the cloud. The refactoring server transforms the UI of the application into a smartphone native UI to be displayed on the user's smartphone. Any user interactions on the native UI elements are sent to the refactoring server and translated into virtual user interactions on the web application. Subsequent UI changes within the web application are dynamically transformed to their smartphone native versions and updated on the client app.

Application refactoring based mobilization is typically achieved through four steps: (i) *Configuration*: This step involves the employees using tool (usually a browser extension or another web application) to configure the mobilization process. The configuration tool enables the users to specify the pages within the web application to mobilize, the particular UI elements to include and any authentication information related to the web application; (ii) *Tracking*: Upon the selection of UI elements to be mobilized, the next stage involves creating a unique identity for each UI element selected by the user. This allows for tracking of elements across different instances of the web application even as the application layout changes either due to the developer modifying the application source or any data related changes; (iii) *Transformation*: The web UI elements are transformed into smartphone platform native versions for optimal user experience. A mapping between the smartphone UI element and the corresponding web element's unique fingerprint is recorded. Any actions performed on the smartphone

Figure 2: Performance of related fingerprints

Figure 3: DOM Tree

```

<HTML><HEAD>
<TITLE>title</TITLE>
</HEAD><BODY>
<DIV ID=d1>
<A href=11></A>
<A href=12></A>
<A href=13></A></DIV>
<DIV ID=d2>
<A href=14></A>
<A href=15></A>
</DIV></BODY></HTML>

```

Figure 4: HTML Source

elements are mapped back to the corresponding web elements using this table; (iv) *Presentation*: Finally, the transformed UI elements are arranged in a layout and presented on the smartphone screen. The final layout can either be specified by the user in the configuration step or generated on-the-fly for the configured UI elements.

Robustness of the smartphone app relies on the ability to correctly map any actions from the smartphone native UI to actions on the original application UI. This requires reliable tracking of the UI elements in the original application even as the application changes. If the tracking is inaccurate, the mapped actions are possibly applied on the wrong element leading to failures.

2.2 Web Applications and DOM: A Primer

A web application is a collection of web pages, most of which are rendered on the browser as HTML documents. The underlying data structure for an HTML document is a tree called the Document Object Model (DOM). Each tag from the document is an element of this tree. The tree is rooted at the `<HTML>` tag. Any nested tags within a particular tag are children elements of that tag. Fig. 3 shows the DOM tree for a simple HTML document in Fig. 4. All modern browsers allow the DOM to be accessed through Javascript DOM API[5].

UI Element Identifiers: A tag can have some HTML attributes

associated with it. For example, the tag `<A href="link1">` has one attribute href. The attribute values need not be unique for the tags. One exception to this rule is the attribute ID. Therefore, the value of an HTML attribute ID is a *globally unique identifier* for that element. While such an identifier is highly desirable for an element, it is not always available. For example, in the Salesforce web application, only 19% of all elements have an ID declared. On the other hand, using the attributes contained within the tags of an element, an *attribute based identifier* can be constructed. However, this identifier is not unique as it is not necessary for an element's attributes to be unique in a DOM tree. For example, in Salesforce, only 16% of elements have a unique set of attributes. Given that the element's own attributes will not help in its identification, the next logical direction is to consider identifiers that are relative to some property. When this property is *relative to the element's local context* within the DOM, the identifier again is not unique. For example, a local identifier consisting of an element's parent, immediate siblings and children is only unique for 13% of the elements on Salesforce. On the other hand, identifiers that describe an element *relative to a unique global property* within the DOM are unique. Some examples of such IDs are - Path from the root, Path from all elements with IDs, Coordinates from the top left corner of a page, Path from the body element, etc. Such an identifier can be constructed for every element within the DOM. Also, given an identifier and any DOM tree, at most only one element can be found with the same identifier.

On the nature of changes: While an element's global relative identifiers can uniquely identify it given a DOM tree, it is not necessary that they remain constant even when the DOM tree changes. For example, when the dashboard of Salesforce application is reconfigured to add a new 'messages' section, all the elements that immediately follow this section (e.g., recently viewed) will have their global relative identifiers changed i.e. the paths to these elements in the DOM tree get altered. DOM trees not only change due to the developer modifying the application, but also because of user interactions[6]. A web application can undergo several types of changes such as layout modifications, content updates, appearance changes (either by the style attributes of elements or when a UI library is updated) and code changes. These changes affect the underlying DOM structure in one of the following ways: (i) *Local changes*: These are the changes wherein only attributes within an element are changed leaving the DOM tree intact. For example, the change in color of a link after a user clicks on it; (ii) *Insertions*: These are changes wherein a new element is inserted into the DOM tree. For example, when a user creates a new task and it is added to the list of all tasks; (iii) *Deletions*: These are the changes wherein an element is deleted from the DOM tree. Any children of this element are inserted at the element's position before deletion. For example, when container DIV is deleted and all its children are moved back into the parent DIV; (iv) *Migrations*: These occur when an element (and any descendants) moves to any other position in the tree. For example, when a user decides to reorganize a dashboard, say by moving the list of tasks to another location within the dashboard; Consider an example wherein a simplistic DOM tree shown in Fig. 3 changes to the tree shown in Fig. 5. The change in attribute ID value of element d_1 to d_3 is a local change, the addition of p_1 is an insertion, absence of a_3 and a_4 are two

deletions, and movement of a_2 is a migration. All other changes can be expressed as a combination of these categories.

2.3 Problem Definition, Scope, and Goals

In this paper, we target the problem of developing an algorithm to reliably track UI elements of a web application across several instances of the application. Note that web-applications innately do not need to have distinct permanent identifiers for the UI elements. Hence, UI elements can be identified only by a relative identifier constructed based on the underlying DOM tree. Hence, the problem involves creating a unique identity (called the *fingerprint*) for the UI elements that remains robust even as the application changes. The fingerprint of any given element should be able to identify the element in a web application, if it exists, even when the application has changed.

We only consider web applications that are rendered as HTML documents on the client browser due to their dominance in the web application ecosystem[7]. In this paper, we treat web elements as containers of content, and not as content itself. For example, in a list of recently viewed headlines, when a particular headline content originally at the top of the list moves to a different position, the web element corresponding to the top position in the list hasn't moved but the content it carries changed. On the other hand, if the list of headlines as a whole is moved to a different location on the page, we assume that the web elements have moved. Further, the framework should be able to track any element belonging to the DOM tree as a whole. Tracking parts of a node individually is beyond the scope of this paper. For example, if there is a paragraph of text declared as a node of the DOM tree, the framework should be able to track paragraph as a whole, and not individual words within it.

The problem considered in the paper can be formally stated as - *Given a web application A with a DOM tree τ , how can a unique fingerprint for any given web element $e \in \tau$ be created, such that the fingerprint can effectively be used to identify the element e in a different instance of the DOM tree τ' .*

Furthermore, any algorithm for tracking UI elements should satisfy the following goals: (i) The algorithm should be robust and withstand a wide range of changes within the DOM structure; (ii) It should be able to track elements with only the information available from a typical web-application and make no assumptions about any additional resources from the web-applications, especially from the application developers; and (iii) Finally, it should be application agnostic.

2.4 Related Approaches and Analysis

When the services relying on the application's front-end (such as the three examples above) fail due to a change in the application, they have to be reconfigured again. This can lead to increased task burden and more costs.

Prior work: The problem of reliably fingerprinting UI elements within a web application has been explored in the past in different contexts. XPath[8] is a widely adopted standard with syntax to describe elements within an XML/DOM tree. Using XPath syntax, a path for traversal within a DOM tree can be specified between two elements. However, XPath only provides a syntax and

it is upto the developer to create a fingerprint with it. Several optimizations[9, 10] have been proposed to interpret XPath. In [11], an element's path from the root of the DOM tree is used as one of its features, but in the context of enhancing mining. [12] uses the shortest path from the nearest ancestor in the DOM tree with an HTML attribute ID as a fingerprint. Here, the context is to record user actions. [13] uses path from the root in conjunction with parent and immediate siblings to identify an element for information extraction. In [14], the authors propose using subtree information for each element in a DOM path. We later show that these single-path based fingerprints do not perform well in dynamic scenarios. [15, 16], use visual features of the page to learn and extract templates for elements. However, generating fingerprints based on visual features is not feasible for a majority of secondary services as it not only requires a large amount of annotated training data but also takes a lot of time.

Performance of related approaches: We evaluate the performance of three fingerprint candidates explored in prior work - *Graphical Coordinates* of the UI element, *Path From Root* to the UI element and *Path from the nearest ancestor with an attribute ID* on four web applications - Sakai, Sharepoint, Peoplesoft and Salesforce. We first randomly select 30 elements from the DOM to track and introduce changes to the DOM (each element has a 0.5% chance of changing). We then find these elements in the modified DOM tree using the three fingerprint candidates. More details on this experimental setup are explained later in Section 4. Fig. 2 shows the ratio of elements whose fingerprint fails to find the elements within the modified tree. On an average, the error rates are 0.73, 0.44 and 0.11, for *Graphical Coordinates*, *Path From Root* and *Path From ID*, respectively. These experiments lead us to a few key insights: (i) DOM based fingerprints that leverage the application layout perform better than pixel based graphical coordinates; (ii) *Path From ID* has a much lower error rate compared to *Path From Root*. This can be attributed to the shorter length of *Path From ID* as shorter paths have a lower probability of being affected by changes; (iii) Even though *Path From ID* performs much better than other fingerprints, the error rate is still very high and unacceptable. This calls for the development of a new fingerprint algorithm.

3 TRACKR: RELIABLE TRACKING OF UI ELEMENTS FOR WEB APPLICATION MOBILIZATION

3.1 Architecture Overview

We design *Trackr* as a passive browser extension that secondary web services can rely on to track any number of UI elements from a web application. *Trackr* extension exposes two key functions - *Trackr.track(element, tname)* and *Trackr.find(tname)*. Any web service can use the *track()* function to track a certain element by passing the element's current handle (Javascript DOM object) - *element* and a name for the tracker - *tname*. *Trackr* then extracts a unique identity (*fingerprint*) for the element and adds it to a database stored in the browser's persistent storage. The fingerprints in the database are indexed by the URL of the web page from which the fingerprint was extracted and the name given to the tracker (*tname*). At every subsequent visit to the page, *Trackr* updates the fingerprint to reflect any changes within the DOM since the last

Figure 5: Changed DOM Tree

Figure 6: Quorum Tree of a_5

time the fingerprint was computed. Using *Trackr*'s *find()* function and the tracker name *tname*, the service can request a current handle to the tracked element. *Trackr* then retrieves the fingerprint from the database and uses it to find the element within the DOM tree and returns the element's handle back to the service.

3.2 Quorum Fingerprinting

In Section 2, we evaluated the performance of three simple fingerprints and observed that single-path fingerprints are insufficient to reliably track elements in dynamic web applications. Also, recall that the fingerprinting algorithm cannot assume any other information from the web applications except it's DOM structure and elements can only be identified relative to some other property of the DOM tree. Therefore, instead of just considering the position of the node in the DOM tree w.r.t. one other element (root or node with an ID), *Trackr* adds redundancy into the fingerprint by considering the position of the node with respect to all elements with an attribute ID. The key insight is that even if some portion of the DOM tree changes between two instances, a majority of the tree remains intact. Therefore, by considering position w.r.t. several anchors and using a simple majority rule to identify the element that matches most of these positions in a modified tree, *Trackr* creates a robust fingerprint. We call this principle *quorum fingerprinting*.

To construct a quorum fingerprint $Q.FP()$ for an element e in a DOM tree τ , *Trackr* reshapes the DOM tree so that it is now rooted at e (*quorum tree*). Reshaping is done by first inverting the shortest path from the element e to the HTML root, so that e is now at the root position of the new tree. *Trackr* then appends all the other elements as children to their respective parent nodes from the old tree. Fig. 6 shows the quorum tree for the node a_5 from the example shown in Fig. 3.

Algorithm 1 Baseline Algorithm

```

1: procedure match-fingerprint( $Q.FP(e)$ ,  $Q.FP(e')$ )
2:    $score \leftarrow 0$ 
3:   for  $id \in Q.FP(e)$  do
4:     if  $id \in Q.FP(e')$  then
5:        $P_1 \leftarrow$  Path corresponding to  $id$  in  $Q.FP(e)$ 
6:        $P_2 \leftarrow$  Path corresponding to  $id$  in  $Q.FP(e')$ 
7:       if  $P_1 == P_2$  then  $score \leftarrow score + 1$ 
8:     end if
9:   end for
10:  return  $score$ 
11: end procedure

```

Using the quorum tree, *Trackr* computes the shortest path from all elements with an attribute ID to the root of the quorum tree. Therefore,

$Q.FP(e) = \{(ID(a), SP(a, e)) \mid a \in \tau \text{ and } a \text{ has an attribute ID.}\}$, where $ID(a)$ is the value of attribute ID for a and $SP(a, e)$ is the shortest path between a and e in the quorum tree. Shortest path $SP(a, e)$ is computed by traversing the quorum tree up from the element with ID until its root. For each element encountered in the traversal, the element's name along with the index w.r.t. to its siblings (in the original tree) is recorded, i.e. given an element e , and it's quorum tree $Q(\tau, e)$,

$SP(a, e) = [(name(e'), index(e')) \mid e' \text{ in the traversal to } e]$, where $index(e')$ is the index of e' w.r.t. its siblings in the original tree τ . For example, the element a_5 has a quorum fingerprint $Q.FP(a_5) = (d_1, [(BODY, 2), (DIV, 2), (A, 2)]), (d_2, [(A, 2)])$.

In order to find an element in another instance of the DOM tree τ' , *Trackr* compares $Q.FP(e)$ to the quorum fingerprints of all other elements of the same type (as e) in τ' . For each element e' of the same type in the modified tree, *Trackr* uses Algorithm 1 to compute a score that reflects how many of the paths in e' 's fingerprint match with those of e . The element with the maximum non-zero score among all other elements is e 's counterpart in τ' . For example, element a_1 can be identified in the modified tree (Fig. 5) using the quorum fingerprint computed from the tree in Fig. 3. Even though it's nearest anchor $DIV d_1$'s ID has changed, the path from the other anchor element d_2 remains intact in the modified tree.

3.3 Fingerprinting Optimizations

Through the principle of quorum fingerprinting, *Trackr* increases the immunity of the fingerprint to DOM changes. We now describe five different optimizations that are progressively applied to the baseline algorithm to make it more robust.

(i) Path Resiliency:

Even though the baseline fingerprint described earlier is robust to secluded changes in the DOM tree away from the element, the presence of many changes in the vicinity of the element can still break the fingerprint. For example, element a_5 's quorum fingerprint is insufficient to find it in the modified tree, as its index w.r.t. to its siblings has changed. To counter this problem, *Trackr* adds resiliency to how paths are calculated. Instead of just using the name of an element and the index (w.r.t. its siblings) to differentiate it from its siblings, *Trackr* computes three parameters from the original tree τ - (i) l : the number of siblings to the left of the node, including the node; (ii) r : the number of siblings to the right of the node, including the node; and (iii) d : the number of children of the node. Each path is now a list of 4-tuples - $(name, l, r, d)$. The computation of $score$

Algorithm 2 Score computation with path resiliency

```

1: procedure match_paths( $P_1, P_2$ )
2:    $score \leftarrow 0, i \leftarrow 0$ 
3:   if  $names(P_1) = names(P_2)$  then  $\triangleright names()$  returns a list of names of all
     elements along the path
4:     while  $i < |P_1|$  do
5:       if  $l(P_1[i]) = l(P_2[i]) \& r(P_1[i]) = r(P_2[i])$  then
6:          $score \leftarrow score + \frac{2}{|P_1|}$ 
7:       else if  $l(P_1[i]) \neq l(P_2[i]) \& r(P_1[i]) = r(P_2[i]) \& c(P_1[i]) = c(P_2[i])$ 
         then
8:          $score \leftarrow score + \frac{l(P_1[i])}{|P_1|(l(P_1[i]) + r(P_1[i]))}$ 
9:       else if  $l(P_1[i]) = l(P_2[i]) \& r(P_1[i]) \neq r(P_2[i]) \& c(P_1[i]) = c(P_2[i])$ 
         then
10:         $score \leftarrow score + \frac{r(P_1[i])}{|P_1|(l(P_1[i]) + r(P_1[i]))}$ 
11:       else
12:         return 0
13:       end if
14:        $i \leftarrow i + 1$ 
15:     end while
16:   end if
17:   return  $score$ 
18: end procedure

```

in line 7 of Algorithm 1 is now replaced with *match_paths* from Algorithm 2. Given an anchor element with ID, a path to an element P_1 computed on the old tree, and a path to an element P_2 computed on the modified tree, *Trackr* first checks the names of all elements along these paths (line 3). For i^{th} element in P_1 and P_2 , if both l and r indices match, then the score is incremented by $\frac{2}{|P_1|}$ (lines 5-6). If only one of indices, say l , matches and the number of children d match, then the score is incremented by $\frac{r(P_1[i])}{|P_1|(l(P_1[i]) + r(P_1[i]))}$. Note that this increment is less than the increment when both l and r match i.e. there is a penalty if one of the indices doesn't match. Also note that, *Trackr* uses the number of children as an additional matching criterion in the score computation to discourage any false positives that may arise.

With this optimization in place, the fingerprint of a_5 computed from the old tree will now be sufficient to find it in the modified tree, as one of it's index (r) in the path from DIV d_2 and the number of children remain intact.

(ii) Weighted Path Matching:

Assuming uniform distribution of changes across the DOM tree, longer paths have a higher probability of breaking with time compared to shorter paths (see discussion in Section II-D). Consider a case where in there are two elements with IDs in a tree. Also consider an element whose fingerprint has two paths P_1 , and P_2 (from the two elements with ID a_1 , and a_2 , respectively). In a modified tree, it is possible that two different elements e_1 , and e_2 have the same match score from Algorithm 2. This can occur when the path from a_1 to e_1 matches completely with P_1 , and the path from a_2 to e_2 matches completely with P_2 . In such a scenario, the probability that the longer path among P_1 and P_2 points to an incorrect element is higher than it's alternate.

Based on this intuition, *Trackr* allocates more importance to matching shorter paths compared to longer paths. This is achieved by multiplying the *score* from Algorithm 2 with a weight that monotonically decreases with an increase in path length¹. *Trackr*

uses an inverse logarithm function $\frac{1}{\ln(1.25 + length)}$ to weigh scores².

(iii) Path Length Amendment:

To find whether two paths in different fingerprints lead to the same element, *Trackr* first checks if the names of elements along the paths are equal. Consider a case wherein an element is deleted along a path but the rest of the path remains intact. In this case, the names of elements will no longer match. Further, if this deletion is close to the element (say it's parent), it is highly possible that all the paths within the element's fingerprint will fail. Through this optimization, *Trackr* accounts for one possible deletion with Algorithm 3. Given a path from a fingerprint computed on the old tree P_1 and a path from a fingerprint computed on the modified tree P_2 (corresponding to the same element with ID as in P_1), if the length of P_2 is one less than that of P_1 , *Trackr* creates a set of dummy paths. For every i^{th} element along the path P_1 , *Trackr* creates a dummy path P'_1 that indicates what P_1 would look like if the i^{th} element was deleted. If the names of elements along this dummy path match to that of P_2 , *Trackr* appends this dummy path into a list of candidate paths for consideration (lines 5-12). When an element is deleted, all the element's children are appended to its parent. To account for this, if the original tree has to be traversed 'DOWN' to reach the deleted element (from the previous element in the path), the siblings count l and r of the deleted element are added to the siblings count of the next element along the dummy path. This is because the next element is a child of the deleted element. On the other hand, if the direction of movement is 'UP', the siblings count of the deleted element are added to those of the previous element along the path (lines 13-19). In the end, each candidate path is matched to P_2 using Algorithm 2, and the final *score* is set to the maximum of all scores (among the candidate paths). In addition, a penalty of 10 is added to the length of path P_1 to discourage false positives (lines 22-27).

(iv) Progressive Path Patching:

Through baseline quorum fingerprinting and the previous three optimizations, an element can be reliably identified even when the DOM tree is changed. When a web service utilizes *Trackr* to track some elements, their fingerprints are computed and stored. When the secondary web service needs to access elements tracked by *Trackr* at a later point in time, the stored fingerprints are used to identify and retrieve the elements. Over time, as the web application undergoes more changes, the paths within the old fingerprint slowly become irrelevant. To avoid this issue, *Trackr* progressively updates the fingerprint every time the user visits the same web application. To do this, *Trackr* first identifies the elements in the web application using the matching procedures outlined earlier. If any of the paths in fingerprint have since been modified, *Trackr* patches the stored fingerprint to reflect the new paths.

(v) Local Signature:

While all of the previous optimizations are designed to create resiliency in the presence of changes away from the element, when the element itself migrates to a different part of the tree either by itself, or as a part of migration of one of its ancestors all of the paths in the fingerprint can fail. However, there is still a high possibility

¹When some areas of the DOM tree are subject to more changes than other areas, the assumption on the uniform distribution of changes does not hold. In this case, more weight can be allocated to paths that do not go through change-prone areas. Finding these areas is beyond the scope of this paper

²Any monotonically decreasing function will produce the same results

Algorithm 3 Matching paths with path length amendment

```

1: procedure match_paths_weighted( $P_1, P_2$ )
2:    $score \leftarrow 0$ 
3:   if  $names(P_1) == names(P_2)$  then
4:      $score += match\_paths(P_1, P_2) \cdot \frac{1}{\ln(1.25 + |P_1|)}$ 
5:   else if  $|P_2| + 1 == |P_1|$  then
6:      $candidates \leftarrow []$ 
7:      $temp \leftarrow P_1$ 
8:     for  $i \leftarrow 0; i < |P_1|; i++$  do
9:        $temp \leftarrow temp \setminus temp[i]$ 
10:      if  $names(temp) \neq names(P_2)$  then
11:        continue
12:      end if
13:      if  $dir(P_1[i]) == 'DOWN'$  then
14:         $l(temp[i]) += l(P_1[i + 1])$ 
15:         $r(temp[i]) += r(P_1[i + 1])$ 
16:      else
17:         $l(temp[i - 1]) += l(P_1[i - 1])$ 
18:         $r(temp[i - 1]) += r(P_1[i - 1])$ 
19:      end if
20:       $candidates.add(temp)$ 
21:    end for
22:     $scores \leftarrow []$ 
23:    for  $P \in candidates$  do
24:       $scores \leftarrow match\_paths(P, P_1) / \ln(1.25 + |P| + 10)$ 
25:    end for
26:     $score \leftarrow \max(scores)$ 
27:  end if
28:  return  $score$ 
29: end procedure

```

Name	Value	Name	Value
# of iterations	50	Probability of change	0.5
# of rounds of change	7	# of tracked elements	30

Table 1: Default Experimental Parameters

that the element's surrounding context remains the same (as the element migrates with its descendants). As a fail-safe for this situation *Trackr* includes an element's local context, called its *signature* in the fingerprint. The *signature* of an element is defined as a list of tag names of the children and grandchildren of the element ordered in a depth first pattern. To find an element using its *signature*, *Trackr* matches the pattern in *signature* to all other elements in the DOM tree and looks for an exact match. As the signature has a very high rate of false positives, it is only used when the all the paths fail.

4 EVALUATION

4.1 Methodology

In this section, we evaluate the performance of *Trackr* on four different web applications: (i) Learning Management - *Sakai*[17], (ii) Human Resources Management - *Oracle Peoplesoft*[18], (iii) Collaboration and Team Management - *Microsoft Sharepoint*[19], and (iv) Customer Relationship Management - *Salesforce*[20]. For each of these websites, we first download the homepage after login and extract the DOM tree. On an average, the number of elements in the DOM tree were 191, 1356, 1357, and 1886, for Sakai, Peoplesoft, Sharepoint, and Salesforce, respectively. We then introduce several rounds of change into this DOM tree. At every change round, each DOM element undergoes a change with a probability

p_{change} (default value = 0.5%)³. Each change round represents the modifications to the DOM tree between two consecutive visits. The default number of rounds of change is set to 7. At the end of each change round, *Trackr* patches the fingerprint (III-A-iv).

For elements that are selected to change, the type of change is chosen randomly among: (i) Attribute change: The value of a randomly chosen HTML attribute is changed to a new value; (ii) Attribute insertion: A new HTML attribute is added to the element's tag; (iii) Attribute deletion: A randomly chosen attribute is deleted from the element's tag; (iv) Insertion: A new element is inserted as a child of the element at a randomly chosen index. The type of this element is randomly selected among all previously seen tags in that DOM tree; (v) Deletion: The element is deleted from the DOM tree and any children are inserted back into the deleted element's position; (vi) Migration: The element, along with its descendants, are moved to a different (randomly selected) location in the DOM tree; These changes broadly reflect the types of changes an element is subjected to in reality.

At the beginning of every iteration, we download the website, extract the DOM tree and select 30 candidate elements (at random) from the DOM tree to be tracked by *Trackr*. After completion of all change rounds, we use *Trackr* to find the candidate elements in the modified DOM tree. To establish the ground truth, at the beginning of each iteration, we add a unique dummy ID for each element in the DOM tree. At the end of the iteration, we compare this dummy-id to the dummy-id of the element returned by *Trackr*. We then compute: (i) *Type-I error* = $\frac{\# \text{ of candidates wrongly identified}}{\text{Total \# of candidates}}$ (when the dummy ID of the element returned by *Trackr* is not equal to the dummy ID of the candidate); (ii) *Type-II error* = $\frac{\# \text{ of candidates not found}}{\text{Total \# of candidates}}$ (when *Trackr* is unable to find the element in the DOM tree, but the element was not deleted); (iii) *Error* = *Type-I error* + *Type-II error*; To eliminate random bias, we repeat the experiments for 50 iterations. We also evaluate these errors for three other fingerprint candidates used in prior work: (i) Graphical coordinates (*Graphical Coordinates*), (ii) Path from the root of the DOM tree (*Path From Root*), and (iii) Path from the nearest ancestor with an attribute ID (*Path From ID*).

4.2 Macroscopic results

Figure 7 shows the errors of fingerprint candidates using the default parameters from Table 1. *Trackr* clearly outperforms all other candidates. On an average, *Trackr* is inaccurate only 4.74% of the time, where as the average error rates for *Graphical Coordinates*, *Path From Root*, and *Path From ID* are 69.74%, 45.44%, and 10.64%, respectively. *Graphical Coordinates* have the highest error rate and it performs worse compared to the fingerprints that rely on the DOM. We can also observe that *Path From ID* has a much lower error compared to *Path From Root*. This improvement can be attributed to the decrease in the path length by computing the path from an element in the vicinity of the given element. This is because the shorter paths have a lower probability of breaking. By building redundancy into the fingerprint, adding resiliency to the paths, giving more importance to shorter paths, accounting for deletions, patching fingerprints when possible, and by using the

³Even though this probability is small, given the size of a typical DOM tree, the number of changes with each round are high.

Figure 7: Performance of *Trackr* compared to Graphical (Coordinates), Path From Root, and Path From ID

Figure 8: Sensitivity to % of nodes changing in DOM

Figure 9: Sensitivity to the number of rounds of changes

local signature when all of the above fail, *Trackr* achieves a 55% improvement over *Path From ID*.

4.3 Microscopic results

We also study the improvement resulting from progressively applying optimizations to the baseline algorithm for Sakai application. When we add the path resiliency optimization to baseline quorum fingerprinting, by including two different indices in the paths, the error is reduced from 5.8% to 4.8%. By introducing weights proportional to the path lengths and including the path length amendment optimization, the error is further reduced to 4.2%. By patching the fingerprints on every visit to the web application, the error reduces to 3.9%. Finally, by using local signatures to find the elements when all the paths break, the error rate of *Trackr* is reduced to only 2.8%.

4.4 Sensitivity Analysis

In this section, we study the sensitivity of *Trackr* to different parameters for two web applications - Sakai and Peoplesoft. Unless mentioned, the experiments use the default parameters from Table 1. For relative comparison, we also show the performance of *Path From ID*. Figure 8 shows the effect on the error of changing the percentage of nodes subject to modification in each round of change. As the percentage of change increases, the error rate also increases for both *Path From ID* and *Trackr*. This is because as the

Figure 10: Sensitivity to types of changes

DOM undergoes more changes, the chances of the paths in the fingerprint breaking also increase. The increase in error is roughly linear. On an average, every 0.1% increase in probability of change results in a 1.1% increase in error for *Trackr*, and a 1.9% increase for *Path From ID*.

We also study the effect of the type of change on error (figure 10). With only insertions allowed, the average error rate is 1.7% for *Trackr* and 7.3% for *Path From ID*. When only deletions are allowed, the average error rate is 2.9% for *Trackr* and 9.9% for *Path From ID*. For only migrations, the average error rate is 4.1% for *Trackr* and 8.8% for *Path From ID*. Given that the probability of change is the same, if all changes are equal, the error should remain the same. However, these numbers indicate that *Trackr* is most resilient to insertions and most sensitive to migrations. This is because when a node migrates, all the paths in the fingerprint are broken leaving *Trackr* with only a local signature to find a match. Since the local signature is more susceptible to finding the wrong elements, the error rate for migrations is higher.

Figure 9 shows the effect of changing the number of rounds of change, the DOM tree is subject to, before the error is computed. As the number of rounds are increased from 0 to 15, the error rate also increases. On an average, per round of change, the increase in error rate is about 0.7% for *Trackr* and 1.63% for *Path From ID*. The increase in error rate is lower for *Trackr*, as the paths in the fingerprint are progressively updated after every round of change. While this number seems alarming, in reality, DOM trees change very slowly with time, and hence *Trackr* still remains robust for a long time.

5 CASE STUDY: TRACKR BASED MULTI-APPLICATION DASHBOARD

To demonstrate the usage of *Trackr*, we developed a multi-application *Dashboard*, a proof-of-concept mobilization app for Android. Using *Dashboard*, users can mobilize and monitor values within UI elements spanning across multiple web applications within one

Figure 11: Dashboard App

mobile app. Figure 11 shows a screenshot of *Dashboard* app through which four values from different websites (or web applications) are tracked. *Dashboard* prototype consists of three components: (i) Configuration chrome extension on the user's computer: To monitor a value through *Dashboard*, the user installs this extension and selects the value to be tracked through a dropdown menu on the UI element containing this value. The extension uses *Trackr* to generate a fingerprint for that element; (ii) Python *Dashboard* server on Amazon EC2: This server uses the fingerprint generated by the chrome extension to track UI elements and periodically monitor the value contained within them. If there is a change in value, the server pushes the new value to the *Dashboard* app (see Figure 12); and (iii) Android *Dashboard* app on the user's smartphone: This app is responsible for displaying the configured values to the user and updating these values upon a push notification from the server.

6 DISCUSSION AND CONCLUSIONS

6.1 Integration with application refactoring based mobilization services

Application refactoring based mobilization services like Capriz [1] allow the users to select elements from the original application UI, customize them and add them to the smartphone UI. For these selected elements, Capriz creates unique fingerprints and associates them to their smartphone native versions. When the user performs an action (say taps a button on the smartphone), these mappings are used to find the corresponding UI element of the original application and execute actions. When the layout of the original application changes, it is possible that the fingerprints generated by Capriz at the time of micro-app creation fail. In this case, the user will now have to recreate the original micro-app from the modified application UI. When the user selects the elements from the Designer, Capriz can use *Trackr* to initiate trackers for them and map these trackers to their corresponding native UI elements. When the user performs an action on the native UI, the tracker names can be used to obtain a handle to the element in the original UI and perform the corresponding action on it.

Figure 12: Architecture of *Dashboard*

6.2 Other Applications of Trackr:

Automation: Automation services like Selenium[21] help users programmatically perform a sequence of tasks within a web application, so to eliminate the task burden of performing them manually. To automate a workflow with Selenium, the developer has to first obtain a handle for the UI element using DOM access methods and specify the type of action with any required parameters in a script. The burden of obtaining the handle for an element rests with the developer. If the application changes after the automation scripts have been written, Selenium will not be able to perform the specified actions on the desired element. The developer has to manually update the scripts with methods to access the correct handle for the elements within the modified DOM tree. For web applications that frequently change, this is impractical. *Trackr* can alleviate the problem of re-coding handles for elements every time the application changes, by allowing the developers to create a robust fingerprint for the elements. By including the *Trackr* browser extension through *add_extension()* method of selenium, and calling *Tracker.track()* on the element's current handle, a tracker for the element can be initialized. At a later point in time, the correct handle to the element can be obtained by passing the tracker's name to *Tracker.find()* method. The following pseudo code demonstrates the usage of *Trackr* in Selenium.

```

\\Adding trackr extension
options = webdriver.ChromeOptions()
options.add_extension('trackr.crx')
driver = webdriver.Chrome(chrome_options=options)

...
\\ Track a list of elements
\\ elem: a handle for the element to be tracked.
\\ name: a name for the tracker
js='return Trackr.track(arguments);'
fp = driver.execute_script('js',elem, name)

...
\\ Get a handle for tracked element
js = 'return Trackr.find(arguments)'
elem = driver.execute_script('js',name)

...

```

Macro-Creation: With services like IFTTT (If this then that) [22] users can create macros to observe certain variables within a web application, create triggers when the variables satisfy some conditions and perform specific actions on a different web service. The burden of providing the triggers for IFTTT is on the web application and therefore, the users are restricted to only those applications that expose an IFTTT compatible API. However, given that very small percentage of applications provide an API, the benefits of macros are severely limited. On the other hand, expecting all web services to provide a functional API to monitor variables and perform tasks

is impractical. With simple extensions to *Trackr*, users can be allowed to create their own triggers even from web applications that do not currently provide an API to integrate with IFTTT. To support this feature, *Trackr* browser extension can be extended to allow users to select a web element to be monitored by right-clicking on it in a web page and selecting an option from the context menu. The users can then be asked to also enter a condition for the monitored value. *Trackr* can then periodically monitors the element from the application. When the trigger conditions are met, *Trackr* can embed the value in a JSON object and send a response back to IFTTT as an HTTP POST message indicating that the trigger is activated. These steps are demonstrated in the following pseudo code.

```
elem, condition <- get from user
// Start a tracker to monitor element
fp = trackr.Track(elem,name);

//periodically monitor value of elem
while(1):
  load_web_service() // Load the web service
  // Get the monitored value
  value = Trackr.find(name).value
  if value satisfies condition:
    //Create a IFTTT JSON response
    response_json= {
      "trigger_identity": "92429d82a41e93048",
      "triggerFields": {"monitored_value": value},
      "ifttt_source":
        {"url": "https://example.com/trigger"},
    }
    // post the response back to IFTTT
    post(response_json);
    sleep(period) // polling frequency
```

6.3 Issues

The following questions could be raised on the approach taken by *Trackr* to track elements: (i) *Can Object tracking algorithms from image processing research [23] be used to track elements?* Object tracking algorithms assume that between two consecutive video frames, the object does not move by a lot. However, given that the web elements are containers of content, their appearance can change drastically between two instances. Therefore, pixel based object tracking methods are not applicable to our problem; (ii) *Can the developers of web applications be forced to declare attribute IDs for all web elements?* It requires remodeling the large body of legacy web applications and is impractical; (iii) *Can trackers be embedded within elements by the secondary web services?* This would require a change at the web application's end to honor the trackers, and is therefore impractical;

The following are some issues with the design choices made for *Trackr*. (i) *Software design choice*: In this paper, we designed *Trackr* to be a browser extension. However, the principles of *Trackr* are not restricted to this design choice. Alternatively, *Trackr* can also be implemented as a javascript library that the web applications can include to avail fingerprinting services; (ii) *Reactive vs. Proactive updates*: *Trackr* updates the stored fingerprints reactively upon every subsequent visit to the web application by the user. While this approach could work well if the pages are frequently visited by the user, a reactive approach where in *Trackr* periodically updates the fingerprint is more suited for infrequently accessed pages; (iii) *Identification of the web page*: *Trackr* stores the fingerprints in a database indexed by the name of the tracker and a URL of the web page. However, it is possible for some web pages to have a dynamic URL e.g., news articles. In this case, a better indexing mechanism

would be to create a fingerprint for the page itself, independent of the URL. One way to achieve this is to select a subset of elements whose presence definitively identifies the web page. We plan to address these issues in the future; (iv) *Complexity*: To find an element, *Trackr* uses DOM access methods to retrieve all elements of the same type and compares their fingerprints. The worst case complexity is proportional to the size of the DOM. However, our observations from experiments indicate that the retrieval does not add any noticeable delays. We plan on performing a more formal study on the complexity in the future.

7 CONCLUSIONS

To conclude, in this paper, we proposed *Trackr*, an algorithm to reliably track UI elements within a web application for robust API creation. We introduced the principle of quorum fingerprinting used by *Trackr* to create unique identities for the tracked elements and presented optimizations designed to increase its robustness. We evaluated *Trackr* over four popular web applications to show attractive benefits. Finally, we discussed *Trackr*'s application through a case study.

REFERENCES

- [1] "Capriza," <https://www.capriza.com/>.
- [2] U. P. Moravapalle and R. Sivakumar, "Just do it: Fast and easy mobilization of web-based enterprise applications," in *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*, ser. HotMobile '18, 2018.
- [3] R. Sivakumar, S. Kakumanu, and C.-L. Tsao, "Rapid view mobilization for enterprise applications," 2018, uS Patent 9,880,796.
- [4] "Powwow mobile," <https://www.powwowmobile.com/>.
- [5] "Javascript html dom," https://www.w3schools.com/js/js_htmldom.asp.
- [6] E. Adar, J. Teevan, S. T. Dumais, and J. L. Elsas, "The web changes everything: Understanding the dynamics of web content," in *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, ser. WSDM '09, 2009, pp. 282–291.
- [7] "Google trends on web platforms," <https://goo.gl/qy558>.
- [8] W. W. W. C. (W3C), "Xml path language (xpath) version 3.1," <https://www.w3.org/TR/xpath-31/>, 2017.
- [9] G. Gottlob and et.al., "Efficient algorithms for processing xpath queries," *ACM Trans. Database Syst.*, vol. 30, no. 2, pp. 444–491, '05.
- [10] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi, "Efficient filtering of xml documents with xpath expressions," *The VLDB Journal*, vol. 11, no. 4, pp. 354–379, Dec. 2002.
- [11] L. Yi and B. Liu, "Web page cleaning for web mining through feature weighting," in *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, ser. IJCAI'03, 2003, pp. 43–48.
- [12] S. Sanadhya, "Ultra-mobile computing: adapting network protocol and algorithms for smartphones and tablets," Ph.D. dissertation, Georgia Institute of Technology, 2013.
- [13] L. Zhang, M. Li, N. Dong, and Y. Wang, "An improved dom-based algorithm for web information extraction," *JOURNAL OF INFORMATION & COMPUTATIONAL SCIENCE*, vol. 8, no. 7, pp. 1113–1121, 2011.
- [14] J. P. Cohen, W. Ding, and A. Bagherjeiran, "Semi-supervised web wrapper repair via recursive tree matching," *CoRR*, 2015.
- [15] S. Zheng, R. Song, and J.-R. Wen, "Template-independent news extraction based on visual consistency," in *AAAI*, vol. 7, 2007, pp. 1507–1513.
- [16] D. Cai, S. Yu, J.-R. Wen, and W.-Y. Ma, "Vips: a vision-based page segmentation algorithm," Tech. Rep., November 2003. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/vips-a-vision-based-page-segmentation-algorithm/>
- [17] "Sakai," <https://sakaiproject.org/>.
- [18] "Peoplesoft applications overview," <http://www.oracle.com/us/products/applications/peoplesoft-enterprise/overview/index.html>.
- [19] "Sharepoint 2016. team collaboration software tools," <https://products.office.com/en-us/sharepoint/collaboration>.
- [20] "Salesforce," <http://www.salesforce.com/>.
- [21] "Selenium - web browser automation framework," <http://www.seleniumhq.org/>.
- [22] "ifttt," <https://ifttt.com/>.
- [23] A. Yilmaz, O. Javed, and M. Shah, "Object tracking: A survey," *Acm computing surveys (CSUR)*, vol. 38, no. 4, p. 13, 2006.