Trackr: Reliable Tracking of UI Elements within Web-Applications to Enable Robust APIfication

ABSTRACT

APIfication of web applications to expose their functionalities for the benefit of *secondary services* is a trend that is now popular. 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.

1 INTRODUCTION

A relatively recent trend in the domain of web applications is to APIfy applications so that evolutionary secondary services may be built upon them seamlessly. A simple example of APIfication of web applications is Google Maps. While Google Maps is itself a popular application used by users to obtain navigation information, other applications can also leverage the API exposed by Google Maps such as those for directions, distance, elevation, geolocation, roads, and time zones. The APIs can be used by any application over HTTP, allowing for faster integration of mapping and navigation intelligence into those applications. Well known applications such as AirBnb, Expedia, Allstate Goodhome, NYTimes, 7-Eleven, and Runstatic all rely on Google Maps APIs [1]. Most popular web-based applications such as Gmail, Salesforce, Twitter, etc., have their own APIs that other applications can leverage.

While applications can indeed be built ground up to support 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 wherein APIfication will have tremendous value; 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 security, incurring maintenance overheads, facilitating monitoring and monetizing, and provisioning for scalability. A more nuanced version of the aforementioned problem is when 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 quite 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 web application (that could 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 application 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 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. 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 three different use-cases that rely on APIfied web applications and discuss how they benefit from *Trackr*.

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 use-cases where *Trackr* can be used to deliver better performance. Finally, Section 6 discusses a few issues with *Trackr* and presents key conclusions.

2 BACKGROUND AND MOTIVATION

2.1 Application Mobilization

Application mobilization services allow enterprise employees to use their smartphones to complete the tasks that were originally performed on a desktop. Among the different mobilization strategies used today, mobilization with application refactoring has minimum development and deployment costs. This involves hosting the application as-is on a cloud and presenting the users with an optimized native UI on their smartphones. Any actions on the native UI are then executed on the original UI in the cloud. Capriza [2], Powwow[3], [**?**] and StarMobile[**?**] are some examples of refactoring based mobilization services.

Refactoring based mobilization is typically achieved through four stages: (i) *Configuration*: This stage involves the employees using a tool to configure various parameters related to the mobilization process. The configuration tool allows 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 accurately tracking the UI elements across different instances of the web application. The tracking process should be able to identify the UI elements 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 then transformed into smartphone platform native versions for optimal user experience and the corresponding transformation is noted in a mapping table. Any actions performed by the user on the transformed elements are mapped back to the tracked original web elements; (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 selected UI elements.

A critical step in refactoring is to 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 performed on the wrong element leading to failures.

2.2 Web Applications and DOM Trees: 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 tree to be accessed through Javascript DOM API[4].

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

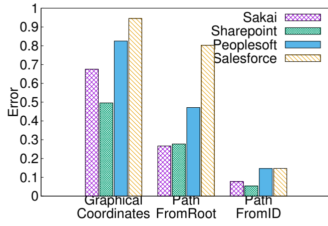


Figure 1: Performance of related fingerprints

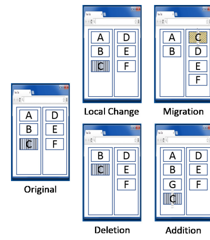


Figure 2: Possible changes in a web application

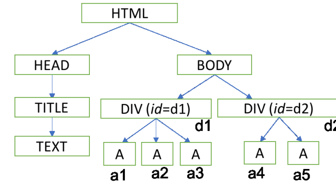


Figure 3: DOM Tree

```
<HTML><HEAD>
<TITLE>title</TITLE>
</HEAD><BODY>
<DIV ID=d1>
<A href=l1></A>
<A href=l2></A>
<A href=l3></A></DIV>
<DIV ID=d2>
<A href=l4></A>
<A href=l5></A>
</DIV></BODY></HTML>
```

Figure 4: HTML Source

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[5]. 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 (see Fig. 2): (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 some property of 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.

We only consider web applications that are rendered as HTML documents on the client browser due to their dominance in the web application ecosystem[6]. 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.

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 Problem Relevance and Significance

Given the rising popularity of web applications, there are several secondary web services available that extend the functionalities provided by the (primary) applications. A common goal among these secondary services is to observe some variables from the web application(s) and act on the them to provide the necessary functions. To explain the relevance of the problem considered in the paper, we discuss three such secondary service use-cases that rely on accurate and reliable tracking of UI elements within a web application.

(i) Automation: Automation services like Selenium[7] 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 particular action on a web element using Selenium, a user has to write a script that declares how to access the web element using simple Javascript DOM access methods and specifies the type of action to be performed. However, as the application and the corresponding DOM tree change, it is possible that the access methods mentioned in the automation scripts fail to access the correct element. In this case, the user has to manually rewrite the automation scripts to access the elements in the modified DOM. This can be burdensome. An accurate element tracking algorithm can effectively eliminate this burden.

(ii) Macro-Creation: With services like IFTTT (If this then that) [8] 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. For example, a user can create a macro that tracks a package and emails a public transit schedule to reach home in time to collect the package. IFTTT relies on APIs provided by the web applications to create triggers and perform actions on their data. However, given that a vast majority of applications do not expose a compatible API, the users are restricted to using only a limited number of web applications. With the availability of an accurate tracking algorithm, third-party services like IFTTT can reliably access application data from their DOM structure, independent of any support from the application itself.

(iii) Application Mobilization: Application mobilization services allow the employees to use their smartphones to complete the tasks that were originally performed on a desktop. Of all mobilization strategies, application refactoring has minimum development and deployment costs. This involves hosting the application as-is on a cloud and presenting the users with an optimized native UI on their smartphones. Any actions on the native UI are then executed on the original UI in the cloud. Capriza [2] is an example of a refactoring based mobilization service. A critical step in refactoring is to 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 performed on the wrong element leading to failures.

2.5 Related Approaches and Performance 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[9] 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[10, 11] have been proposed to interpret XPath. In [12], 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. [13] 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. [14] uses path from the root in conjunction with parent and immediate siblings to identify an element for information extraction. In [15], 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. [16, 17], 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. 1 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

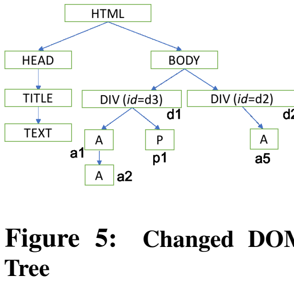


Figure 5: Changed DOM Tree

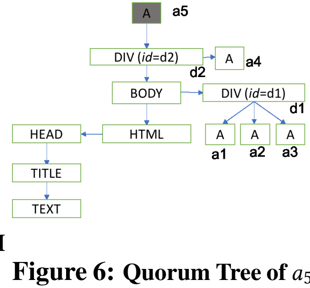


Figure 6: Quorum Tree of a_5

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.

3 TRACKR: RELIABLE TRACKING OF UI ELEMENTS WITHIN WEB APPLICATIONS

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 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.

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.

Using the quorum tree, *Trackr* computes the shortest path from all elements with an attribute ID to the root of the quorum tree. Therefore, $Q.FP(e) = (ID(a), SP(a, e)) \forall a \in \tau$and a has an attribut 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 upwards from the element with ID until its root is reached. 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 $Q(\tau, e)$, $SP(a, e) = [(name(e'), index(e')) \forall e'$encountered in the traversal to where $index(e')$ is the index of e' w.r.t it's 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

Algorithm 1 Baseline Algorithm

```
1:  procedure match_fingerprint(Q.FP(e), Q.FP(e'))
2:      score ← 0
3:      for id ∈ Q.FP(e) do
4:          if id ∈ Q.FP(e') then
5:              P1 ← Path corresponding to id in Q.FP(e)
6:              P2 ← Path corresponding to id in Q.FP(e')
7:              if P1 == P2 then score ← score + 1
8:              end if
9:          end if
10:     end for
11:     return score
12: end procedure
```

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* 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

Algorithm 2 Score computation with path resiliency

```
1:  procedure match_paths(P1, P2)
2:      score ← 0, i ← 0
3:      if names(P1) = names(P2) then          ▷ names() returns a list of names of all elements along the path
4:          while i < |P1| do
5:              if l(P1[i]) = l(P2[i])&r(P1[i]) = r(P2[i]) then
6:                  score ← score + 2/|P1|
7:              else if l(P1[i]) ≠ l(P2[i])&r(P1[i]) = r(P2[i])&c(P1[i]) = c(P2[i]) then
8:                  score ← score + l(P1[i]) / (|P1|(l(P1[i])+r(P1[i])))
9:              else if l(P1[i]) = l(P2[i])&r(P1[i]) ≠ r(P2[i])&c(P1[i]) = c(P2[i]) then
10:                 score ← score + r(P1[i]) / (|P1|(l(P1[i])+r(P1[i])))
11:             else
12:                 return 0
13:             end if
14:             i ← i + 1
15:         end while
16:     end if
17:     return score
18: end procedure
```

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 length[1]. *Trackr* uses an inverse logarithm function $\frac{1}{\ln(1.25+length)}$ to weigh scores[2].

(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 ith 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

[1]When 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

[2]Any monotonically decreasing function will produce the same results

Algorithm 3 Matching paths with path length amendment

```
1: procedure match_paths_weighted(P1, P2)
2:     score ← 0
3:     if names(P1) == names(P2) then
4:         score+ = match_paths(P1, P2). 1/ln(1.25+|P1|)
5:     else if |P2| + 1 == |P1| then
6:         candidates ← []
7:         temp ← P1
8:         for i ← 0; i < |P1|; i + + do
9:             temp ← temp\temp[i]
10:            if names(temp) ≠ names(P2) then
11:                continue
12:            end if
13:            if dir(P1[i]) == 'DOWN' then
14:                l(temp[i])+ = l(P1[i + 1])
15:                r(temp[i])+ = r(P1[i + 1])
16:            else
17:                l(temp[i − 1])+ = l(P1[i − 1])
18:                r(temp[i − 1])+ = r(P1[i − 1])
19:            end if
20:            candidates.add(temp)
21:        end for
22:        scores ← []
23:        for P ∈ candidates do
24:            scores ← match_paths(P, P1)/ln(1.25 + |P| + 10)
25:        end for
26:        score ← max(scores)
27:    end if
28:    return score
29: end procedure
```

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. 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 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.

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

4 EVALUATION

Methodology: In this section, we evaluate the performance of *Trackr* on four different web applications: (i) Learning Management - *Sakai[18]*, (ii) Human Resources Management - *Oracle Peoplesoft[19]*, (iii) Collaboration and Team Management - *Microsoft Sharepoint[20]*, and (iv) Customer Relationship Management - *Salesforce[21]*. 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

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

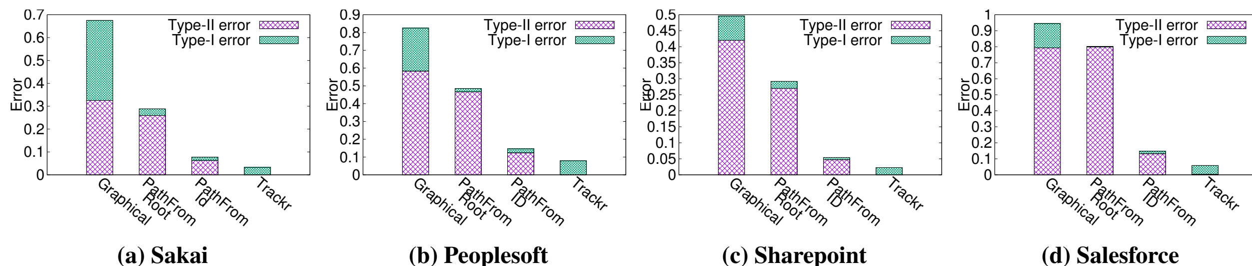


Figure 7: Performance of *Trackr* compared to Graphical (Coordinates), Path From Root, and Path From ID

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* = $\frac{\textit{\# of candidates wrongly identified}}{\textit{Total \# of candidates}}$ (when the dummy ID of the element returned by *Trackr* is not equal to the dummy ID of the candidate); (ii) *Type-II error* = $\frac{\textit{\# of candidates not found}}{\textit{Total \# of candidates}}$ (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*).

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

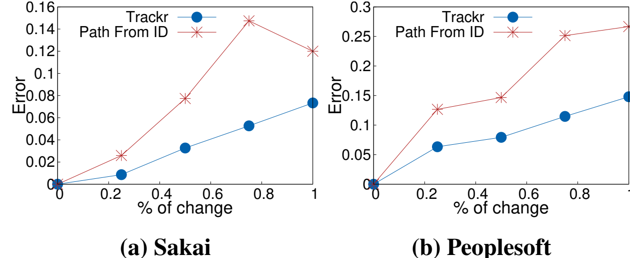
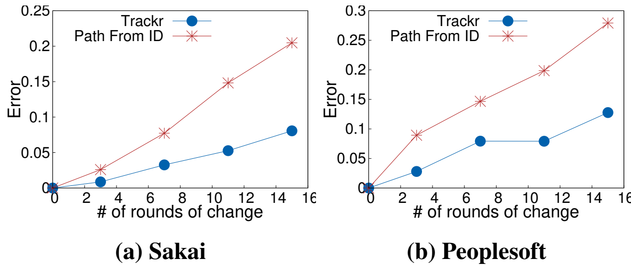


Figure 8: Sensitivity to % of nodes changing in DOM

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 by computing paths from many elements, adding resiliency to the paths, giving more importance to shorter paths, accounting for deletions, patching fingerprints when possible, and by using the local signature when all of the above fail, *Trackr* achieves a 55% improvement over *Path From ID*.

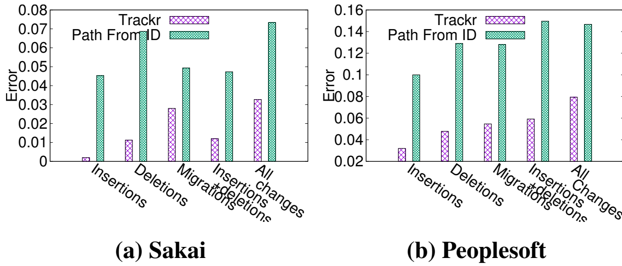
Microscopic results: We also study the improvement resulting from progressively applying optimizations to the baseline algorithm for Sakai application. When we add the path resiliency optimization to baseline quorum fingerprinting, by including two different indices in the paths, the error is reduced 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%.

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 DOM undergoes more changes, the chances of the paths in the fingerprint breaking also increase. The increase in error is



(a) Sakai **(b) Peoplesoft**

Figure 9: Sensitivity to the number of rounds of changes



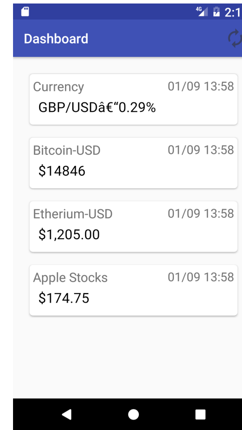
(a) Sakai **(b) Peoplesoft**

Figure 10: Sensitivity to types of changes

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 the 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 local signature to find a match. However, 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 period of time.



1. GBP-USD gains www.nytimes.com
2. Bitcoin price www.bitcoin.com
3. Etherium price www.coinbase.com
4. Apple stock price www.nasdaq.com

Figure 11: Dashboard App

5 USE CASES FOR TRACKR

5.1 Prototype:

To demonstrate the usage of *Trackr*, we developed *Dashboard*, a proof-of-concept Android mobilization app. Using *Dashboard*, users can mobilize and monitor values within UI elements spanning across multiple web applications within one mobile app. Figure **??** shows In this section we discuss how *Trackr*'s accurate and reliable fingerprinting of UI elements can be integrated with the three secondary service use cases we introduced in Section 2.

Automation: 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. For example, to enter text in a text box, the developer has to script how to access the text box element, say through xpath expressions, and use the method *find_element_by_xpath()* to obtain a handle. Text can be inserted by calling the method *send_keys()* on the handle. The burden of obtaining the right handle for an element rests with the developer. If the web 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 then 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 burdensome and 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: In IFTTT, macros can be configured through a GUI, wherein the users can select from a list of available triggers and actions. 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
```

Application Mobilization: Mobilizing enterprise applications with refactoring involves hosting the application on a cloud server and providing a highly optimized native UI for the users to interact with the application on their smartphones. Through Capriza [2], users can create micro-apps that perform specific workflows on traditional enterprise applications through a simple GUI tool called the Designer. It allows the users to select elements from the original application UI, customize them and add them to the smartphone UI. For these selected elements, Capriza 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. For the created mobile app to function correctly, the actions have to be executed on the correct elements in the original UI. When the layout of the original application changes, it is possible that the fingerprints generated by Capriza at the time of micro-app creation fail. In this case, the user will not be able to perform the intended workflow on the micro-app. 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, Capriza 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.

6 ISSUES AND CONCLUSIONS

The following questions could be raised on the approach taken by *Trackr* to track elements: *(i) Can Object tracking algorithms from image processing research [22] 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;

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 three uses cases.

REFERENCES

[1] "Google maps apis," https://developers.google.com/maps/showcase/,.

[2] "Capriza," https://www.capriza.com/.

[3] "Powwow mobile," https://www.powwowmobile.com/.

[4] "Javascript html dom," https://www.w3schools.com/js/js_htmldom.asp.

[5] 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.

[6] "Google trends on web platforms," https://goo.gl/qqy558.

[7] "Selenium - web browser automation framework," http://www.seleniumhq.org/.

[8] "Ifttt," https://ifttt.com/.

[9] W. W. W. C. (W3C), "Xml path language (xpath) version 3.1," https://www.w3.org/TR/xpath-31/, 2017.

[10] G. Gottlob and et.al., "Efficient algorithms for processing xpath queries," *ACM Trans. Database Syst.*, vol. 30, no. 2, pp. 444–491, '05.

[11] 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.

[12] 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.

[13] S. Sanadhya, "Ultra-mobile computing: adapting network protocol and algorithms for smartphones and tablets," Ph.D. dissertation, Georgia Institute of Technology, 2013.

[14] 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.

[15] J. P. Cohen, W. Ding, and A. Bagherjeiran, "Semi-supervised web wrapper repair via recursive tree matching," *CoRR*, 2015.

[16] S. Zheng, R. Song, and J.-R. Wen, "Template-independent news extraction based on visual consistency," in *AAAI*, vol. 7, 2007, pp. 1507–1513.

[17] 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/

[18] "Sakai," https://sakaiproject.org/.

[19] "Peoplesoft applications overview," http://www.oracle.com/us/products/applications/peoplesoft-enterprise/overview/index.html.

[20] "Sharepoint 2016. team collaboration software tools," https://products.office.com/en-us/sharepoint/collaboration.

[21] "Salesforce," http://www.salesforce.com/.

[22] A. Yilmaz, O. Javed, and M. Shah, "Object tracking: A survey," *Acm computing surveys (CSUR)*, vol. 38, no. 4, p. 13, 2006.