Near-Duplicate Detection in Web App Model Inference

Rahulkirsha Yandrapally
University of British Columbia
Vancouver, BC, Canada
rahulyk@ece.ubc.ca

Andrea Stocco∗
Università della Svizzera italiana
Lugano, Switzerland
andrea.stocco@usi.ch

Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

ABSTRACT
Automated web testing techniques infer models from a given web app, which are used for test generation. From a testing viewpoint, such an inferred model should contain the minimal set of states that are distinct, yet, adequately cover the app’s main functionalities. In practice, models inferred automatically are affected by near-duplicates, i.e., replicas of the same functional webpage differing only by small insignificant changes. We present the first study of near-duplicate detection algorithms used in within app model inference. We first characterize functional near-duplicates by classifying a random sample of state-pairs, from 493k pairs of webpages obtained from over 6,000 websites, into three categories, namely clone, near-duplicate, and distinct. We systematically compute thresholds that define the boundaries of these categories for each detection technique. We then use these thresholds to evaluate 10 near-duplicate detection techniques from three different domains, namely, information retrieval, web testing, and computer vision on nine open-source web apps. Our study highlights the challenges posed in automatically inferring a model for any given web app. Our findings show that even with the best thresholds, no algorithm is able to accurately detect all functional near-duplicates within apps, without sacrificing coverage.

CCS CONCEPTS
• Software and its engineering → Software testing and debugging.

KEYWORDS
near-duplicate detection, reverse engineering, model-based testing

1 INTRODUCTION
Automated techniques such as web app crawlers are widely used to reverse-engineer state-based models as a viable vehicle for various analysis and testing tasks such as automated test generation. The state in such models represents the dynamic webpage of the app, as represented by the Document Object Model (DOM) in the browser. Crawlers are capable of efficiently exploring a large state space of any given web app. However, an adequate model should contain only the minimal set of distinct states that represent the web app functionalities, while discarding insignificant states that do not contribute to exposing new functionality to the end user. Instances of such states are pages only differing by small cosmetic changes, which are also referred to as near-duplicates in the literature [23, 24, 29, 34]. To discard such near-duplicate webpages, crawlers have adopted state abstraction functions over the DOM [26, 36, 37, 45] as a proxy for the similarity of webpages. The downside of these abstractions is that minimal changes to the DOM can result in duplicate states in the model, even if such DOM changes are not reflected on the final UI visually, and therefore might not be representative of a new webpage functionality. From an end-to-end (E2E) testing perspective, clone and near-duplicate states in web app models negatively impact their accuracy and completeness, undermining the quality of the test suites generated from such models in terms of size, runtime, and coverage.

Clone and near-duplicate detection across different web apps has been an active research topic in many fields [23, 24, 29, 34]. In information retrieval, the content of a webpage has been the primary focus, because the purpose of web search engines is to index and retrieve information from webpages through search queries. Computer vision techniques have been employed to detect visually similar webpages, for instance in phishing detection [2, 21]. Other approaches leverage state abstractions based on the similarity of URLs, textual content and the DOM [17, 44, 53]. Detecting near-duplicate pages is a challenging problem as there is no generally accepted definition of near-duplicate states and there is no unified standard against which a technique can be assessed [28, 29]. A second challenge pertains to the selection of similarity thresholds that such techniques need as input to determine when two pages are similar. These thresholds are usually educated guesses, as no systematic means have been proposed so far to estimate them automatically.

In this work, we are interested in detecting distinct states in web app models in the context of functional E2E web testing. Our aim is to study the nature of duplicate states occurring within a web app, and provide a systematic approach to selecting thresholds for inferring an optimal model, i.e., having the lowest number of (near-)duplicate states. To this end, we evaluate the capability of 10 near-duplicate detection algorithms in identifying clone, near-duplicate, and distinct web app states. We adopt techniques from three different domains—information retrieval, web testing, and computer vision—where the textual content, the DOM tree, and the
visual screenshot of the page are used to measure the similarity between states. Our goal is to assess whether textual, structural, or visual features are related with semantic properties of webpages and provide meaningful means to understanding their degree of functional relatedness from an E2E testing perspective.

To select the similarity thresholds for fine-tuning such techniques, we first crawled 6k websites randomly selected from Alexa’s top million URLs. We retrieved 493k pairs of states belonging to the same application, and computed the similarity distance between these pairs using each near-duplicate algorithm. We then manually classified 1k random state-pairs into three categories of clone, near-duplicate, or distinct. We used our empirical data of distances to choose thresholds for each algorithm through statistical and optimization search methods. We evaluated their accuracy in automatically classifying clones and near-duplicates in the remaining unlabelled portion of the dataset. Further, we evaluated these configured algorithms on a subject set of nine unseen web apps, for which manual ground truth models were previously created.

Our work makes the following novel contributions:

- The first study of 10 different near-duplicate detection techniques applied in the context of web app model inference.
- A classification of different categories of near-duplicates occurring within a web app.
- Systematic ways of threshold selection for near-duplicate detection as well as an empirical evaluation of their effectiveness in test models.
- The toolset comprising the 10 near-duplicate detection algorithms, which is available for download [5].
- A dataset of 99k manually classified pairs of webpages, of which (1) 1.5k pairs are randomly sampled from 6k websites, and (2) 97.5k from nine real-size web apps. Our dataset can be used by others to conduct similar near-duplicate detection studies and is also available for download publicly [5].

Our results show that even with the best thresholds, no algorithm is able to accurately detect all functional near-duplicates within apps. In practice, existing near-duplicate detection techniques are not designed to find functional similarity in a way that human testers regularly assess while testing web apps. For certain types of near-duplicates, we observed that the model deteriorates over time as the crawl progresses. For instance, although RTED was able to achieve a high accuracy $F_1$ score of 0.95 initially, the final produced model had only an $F_1$ of 0.45. This deterioration is due to the accumulation of numerous near-duplicates to the model, which decreases precision. Our results underline the need for further research in devising techniques geared specifically toward web test models, i.e., that can distinguish between different types of near-duplicates such as those found in our study.

2 REDUNDANCIES IN WEB APP MODELS

In practice, web testing is often performed in an end-to-end (E2E) fashion, by verifying the correctness of the web app state in response to user events and interactions with the GUI (e.g., clicks, and forms submissions). This task is performed either manually by testers, or by writing test scripts with test automation tools such as Selenium [46].

Automated techniques, on the other hand, generate web test cases from models that are inferred through reverse-engineering techniques. A popular method to model construction for modern web apps is automated state exploration, also known as web app crawling [35, 54]. Such techniques dynamically analyze the web app under test by automatically firing events and checking the webpage for changes. When new state changes are detected, the model is updated to reflect the event causing the new state. Generated models can be represented in various formats such as UML state diagrams, Finite State Machines (FSM), or State-Flow Graphs (SFG) [35, 42, 54].

To avoid redundancies in the model, states that are identical or highly similar to previously encountered states should be discarded. For instance, let us consider Figure 1, a web app in which the homepage shows a list of phones. When the user clicks on any of the phones in that list, she is redirected to another webpage displaying the detailed characteristics of the selected phone. From a functional testing viewpoint, however, a page containing a list of 20 phones is conceptually the same as one listing the same 20 phones plus one extra phone.

The problem of detecting already visited states can be cast as an equivalence problem: given two web page states $p_1$ and $p_2$ explored by the crawler, a state abstraction function determines whether $p_1 \approx p_2$. More formally:

**Definition 1 (State Abstraction Function).** A state abstraction function (SAF) $\mathcal{A}$ is a pair $(\text{dist}, t)$, where $\text{dist}$ is a similarity function, and $t$ is a threshold defined over the values of $\text{dist}$. Given two web pages $p_1, p_2$, $\mathcal{A}$ determines whether the distance between $p_1$ and $p_2$ falls below $t$.

$$\mathcal{A}(\text{dist}, p_1, p_2, t) = \begin{cases} \text{true} & : \text{dist}(p_1, p_2) < t \\ \text{false} & : \text{otherwise} \end{cases}$$

In practice, $\mathcal{A}$ is defined based on the similarity of some abstracted notion of the web pages such as their URLs, textual content, DOM structure, or screenshot image. However, the amount and nature of changes occurring in a web page with respect to the functionality of the app is not always directly proportional to the amount of changes in the DOM tree, textual content, or visual aspects of the page.

Let us consider using a crawler equipped with a SAF based on DOM content similarity on our sample web app of Figure 1. This SAF is less tolerant to content (textual) changes occurring in web pages. Therefore, each page displaying a new phone’s

![Figure 1: Example of near-duplicate web pages.](image)
characteristics might be considered a different state and many functionally similar occurrences of already modelled pages (i.e., near-duplicates) would be included in the model. If we use this inflated model to generate test cases, the overall functional coverage does not change when the generated tests exercise the phone details page multiple times, thus potentially wasting precious testing time and resources.

On the other hand, another “better” SAF, for instance based on the DOM tree similarity with a proper threshold value, would consider all such phone detail pages as the same, providing a more concise model for the web application of our example. However, a high threshold value might cause other relevant functionality to be abstracted away as well, resulting in an incomplete model.

Near-duplicate detection techniques have been studied for reducing the occurrence of redundant similar pages across web apps, e.g., in web search engines [20] or phishing detection [38]. An understanding of whether such techniques apply also in detecting functional near-duplicates within the same web app is missing in the literature. Despite its prevalence and importance, this problem is understudied, because it is hard to define a notion of equivalence for two arbitrary webpages. Moreover, in the general case, deciding a priori which abstraction function and which threshold would work best for a given web app is a challenging task, as it requires substantial domain-specific knowledge of the web app under test.

3 NEAR-DUPLICATE ALGORITHMS

In this work, we study 10 near-duplicate detection algorithms from three different domains, namely, information retrieval, web testing, and computer vision. Table 1 presents the techniques, along with the domain they belong to, the input types, a short description, and their distance output.

### 3.1 Information Retrieval

Near-duplicate detection has been applied to index the massive volume of web pages continuously retrieved by web crawlers for search engines. The overall goal is to select only a relevant set of pages based on the provided user search string. In this setting, performance is the most important factor, therefore hashing mechanisms have been adopted due to their design simplicity and speed of comparison. As an input, the web page content is typically the primary focus when designing algorithms used in this domain.

We chose two content hashing algorithms from this domain: (1) simhash [20], a popular and effective web page fingerprinting technique adopted by Google to index web pages [29], and (2) Trend Locality Sensitive Hash (TLSH) [38], a hashing technique for fingerprinting source code, employed for malware detection [55].

### 3.2 Web Testing

In the web testing domain, researchers have studied DOM-based abstractions to compare webpages during the crawling of the application under test. The assumption is that two web pages sharing similarities among their DOMs are likely to represent pages having analogous functionalities, hence it is worthwhile to consider them the same. The DOM can be treated either as a tree-like structure, or as a simple string of characters.

We chose three different similarity algorithms over the DOM that have been employed as state abstraction functions in prior web testing research [36, 37, 49]: (1) tree edit distance with the RTED algorithm [39], (2) Levenshtein distance [30] over the string represented by the DOM, and (3) string equality between two DOM strings, which we use as baseline.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Algorithm</th>
<th>Input</th>
<th>Description</th>
<th>Distance Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information Retrieval</td>
<td>Web Search</td>
<td>simhash [20]</td>
<td>DOM (content)</td>
<td>64 bit fingerprinting technique that uses features extracted from the web page content</td>
</tr>
<tr>
<td>Malware detection</td>
<td>TLSH [38]</td>
<td>DOM (content)</td>
<td>Locality-sensitive 256 bit hashing scheme that is robust to minor variations of the input</td>
<td>Hamming distance of two 256 bit digests</td>
</tr>
<tr>
<td>Web Testing</td>
<td>RTED [37, 39]</td>
<td>DOM (Tree)</td>
<td>Minimum-cost sequence of node edit operations that transform one DOM tree into another</td>
<td>Tree edit distance value normalized by the sum of nodes in the two trees</td>
</tr>
<tr>
<td></td>
<td>Levenshtein [30, 36]</td>
<td>DOM (String)</td>
<td>Minimum number of single-character edits required to transform one string into another</td>
<td>Edit distance value normalized by the sum of the string lengths</td>
</tr>
<tr>
<td></td>
<td>String Equality (baseline)</td>
<td>DOM (String)</td>
<td>String equality comparison</td>
<td>Boolean value</td>
</tr>
<tr>
<td>Computer Vision</td>
<td>PHash [39]</td>
<td>Screenshot</td>
<td>128 bit perceptual hash that represent the lowest frequencies of pixel brightness, to which discrete cosine transform (DCT) is applied to retrieve a brightness matrix</td>
<td>Hamming distance of two 128 bit digests</td>
</tr>
<tr>
<td></td>
<td>Block-mean [57]</td>
<td>Screenshot</td>
<td>256 bit perceptual hash obtained by dividing the image into non-overlapping blocks, which are encrypted with a secret key and normalized median value is calculated</td>
<td>Hamming distance of two 256 bit digests</td>
</tr>
<tr>
<td>Whole Image Comparison</td>
<td>Histogram [52]</td>
<td>Screenshot</td>
<td>Color distribution of a digital image</td>
<td>Number of different pixels normalized by the maximum number of pixels in the two images</td>
</tr>
<tr>
<td></td>
<td>PDiff [58]</td>
<td>Screenshot</td>
<td>A human-like concept of similarity that uses spatial, luminance, and color sensitivity</td>
<td>Normalized structural distortion value</td>
</tr>
<tr>
<td>Structural Similarity</td>
<td>SSIM [4]</td>
<td>Screenshot</td>
<td>Simulates the high sensitivity of human visual system to structural distortions while compensating for non-structural distortions</td>
<td>Number of different key-points normalized by the maximum number of key-points in both images</td>
</tr>
<tr>
<td>Feature Detection</td>
<td>SIFT [32]</td>
<td>Screenshot</td>
<td>Computes local feature vectors and image descriptors which are invariant to geometric affine transformations like scaling/rotation</td>
<td>Number of different key-points normalized by the maximum number of key-points in both images</td>
</tr>
</tbody>
</table>

3.1 Information Retrieval

Near-duplicate detection has been applied to index the massive volume of web pages continuously retrieved by web crawlers for search engines. The overall goal is to select only a relevant set of pages based on the provided user search string. In this setting, performance is the most important factor, therefore hashing mechanisms have been adopted due to their design simplicity and speed of comparison. As an input, the web page content is typically the primary focus when designing algorithms used in this domain.

We chose two content hashing algorithms from this domain: (1) simhash [20], a popular and effective web page fingerprinting technique adopted by Google to index web pages [29], and (2) Trend Locality Sensitive Hash (TLSH) [38], a hashing technique for fingerprinting source code, employed for malware detection [55].

3.2 Web Testing

In the web testing domain, researchers have studied DOM-based abstractions to compare webpages during the crawling of the application under test. The assumption is that two web pages sharing similarities among their DOMs are likely to represent pages having analogous functionalities, hence it is worthwhile to consider them the same. The DOM can be treated either as a tree-like structure, or as a simple string of characters.

We chose three different similarity algorithms over the DOM that have been employed as state abstraction functions in prior web testing research [36, 37, 49]: (1) tree edit distance with the RTED algorithm [39], (2) Levenshtein distance [30] over the string represented by the DOM, and (3) string equality between two DOM strings, which we use as baseline.
3.3 Computer Vision

Image similarity is one of the main topics in computer vision. Many techniques have been proposed and studied, at different levels of granularity, ranging from low-level pixel matching up to high-level feature-based matching. These techniques are applied in indexing and searching, summarization, object detection and tracking, facial recognition, and also copyright image detection. We consider different classes of image-based algorithms.

Image hashing techniques map visually identical or nearly-identical images to the same (or similar) digest called image hash. We chose two image hashing algorithms: (1) block-mean hash [57] and (2) perceptual hash (PHash) [59], which have been used in multimedia security for image retrieval, authentication, indexing and copy detection. Whole image matching techniques focus instead on individual pixels composing the image. Color-Histogram [43] and Perceptual Diff (PDiff) [58] have been successfully applied in previous web testing work for detecting cross-browser incompatibilities [33]. A downside of those techniques is that they are affected by changes in coordinates of web elements common in responsive web layouts. Structural similarity techniques quantify image quality degradation. For instance, Structural Similarity Index (SSIM) [4] has been shown to be effective due to the highly structured nature of web apps' GUls [21]. Lastly, feature detection techniques have been widely employed for near-duplicate image detection. For instance, Scale Invariant Feature Transform (SIFT) [32] has been applied to aid web test repair [51] and phishing detection [2].

To the best of our knowledge, this work is the first to consider and evaluate visual image similarity as a near-duplicate detection technique for web application crawling.

4 EMPIRICAL STUDY DESIGN

The goal of our study is to determine how existing near-duplicate detection techniques can be employed to obtain an optimal model of a web application that can be used for E2E testing.

RQ1: What type of functional near-duplicates exist within apps?
RQ2: How well can functional near-duplicates be detected?
RQ3: What is the impact of near-duplicates and detection techniques in inferring a web-app model?

First, in Section 5, we randomly sample 1,000 within-app state-pairs from a dataset created by crawling 6k randomly selected URLs. We characterize the changes occurring between states within an app and identify how they lead to different classes of functional near-duplicates (RQ1). We label these 1,000 pairs as either clones, near-duplicates or distinct states, and compute the distance between them for all the ten near duplicate techniques described in Section 3.

In Section 6, using these labelled pairs, we compute statistical and optimal thresholds to fine-tune each near duplicate technique. Through this, we aim to determine whether such randomly sampled distances from a large dataset can be used to automatically classify state-pairs in unseen web apps and detect near-duplicates (RQ2).

In Section 7, we determine the best near-duplicate detection techniques and application-specific thresholds to infer web app models for nine open-source web apps covering the different near-duplicate categories. Finally, we analyze these models to determine how different kinds of near-duplicates impact model inference (RQ3).

5 RQ1: NEAR-DUPLICATES IN WEB APPS

In order to determine what kinds of functional near-duplicates occur within apps, we first create a dataset of within app state-pairs and their calculated distances for each near-duplicate detection algorithm. Then, we manually characterize the nature of differences between pairs of pages and classify them in a random sample.

5.1 Dataset Creation

First, we crawl randomly selected website URLs from the top one million as provided by Alexa, a popular website that ranks sites based on their global popularity for a week using CRAWLJAX [36], an event-driven crawler for exploring highly dynamic web apps. We configured CRAWLJAX to run using the Chrome browser, with its default simple state abstraction function, namely string equality (see Table 1), and a runtime limit of five minutes for each crawl.

To account for network communication errors and the tool’s exploration limitations, e.g., on sites that require login credentials, we filtered out sites for which the crawl models obtained contained less than 10 states. After this filtering stage, we retained 1,064 different sites accounting for 30,202 states from the original 6,359 web crawls. We then created all possible 677,415 pair-wise combinations of states within each crawl, which we call state-pairs.

5.1.1 Computing Distances. We computed the distance for each state-pair using each of the 10 algorithms presented in Table 1. We normalized all distances computed by each algorithm, as described in the Distance Output column of Table 1, but we never compare outputs of different techniques.

5.1.2 Distance Normalization. The raw distances which quantify the difference between two given pages have different output spaces based on the page characteristic used by the technique. As an example, given a state-pair of web pages, PDiff outputs the number of perceptually different pixels between their screenshots, whereas BlockHash returns the hamming distance between image hashes. For the sake of comprehensibility, we normalized all distances computed by each algorithm, as described in the Distance Output column of Table 1, but we never compare outputs of different techniques.

5.2 Classification of Changes

To gain a better understanding of what changes within web pages characterize near-duplicates, we classify the differences of the state-pairs in our dataset from the point of view of a human tester who is interested in functionality coverage.

5.2.1 Procedure. Manually examining state-pairs is a time consuming task requiring familiarity with the functionality of the application. Therefore, we randomly sampled a set, called RS, of 1,000 state-pairs from our final dataset of 493,088 state-pairs.
which allows us to have a confidence level of 99% with a 4% 
margin of error in deriving a representative statistic. For each 
state-pair \((p_i, p_j) \in S\), the authors of the paper visually analyzed, 
in isolation, the screenshot images (and the original web pages 
where necessary) of the two web app states from a functional 
testing perspective, to obtain a set \(D\) of differences. Each 
difference in \(D\) is defined as \(\Delta(p_i, p_j) = \{ \delta(e_i, e_j) \}\) where \(\delta(e_i, e_j)\) 
is a pair of non-identical web elements in which \(e_i \in p_i\) and 
\(e_j \in p_j\). Finally, each author assigned a descriptive label to each 
detected difference.

5.2.2 Difference Categorization. After enumerating all differences 
across the 1,000 state-pairs in \(RS\), the authors reviewed them to-
together to resolve conflicts and reached consensus on equivalence 
classes of differences. Our study revealed the following categories.

Definition 2 (Unrelated (\(U\))). Given a difference \(\delta(e_i, e_j)\), neither 
of \(e_i\) or \(e_j\) are related to any functionality offered by the web app.

Examples of these differences include changes in background 
images, or GUI widgets related to advertisement (see red ovals 
in Figure 2a).

Definition 3 (Duplicated (\(D\))). Given a difference \(\delta(e_i, e_j)\), \(e_i\) and 
\(e_j\) replace each other in the original pages \(p_i\) and \(p_j\) without adding 
any new functionality to either page.

Two distinct subcategories of duplicated differences emerged:

- **Replacement** (\(D_1\)): \(D_1 : e_i \equiv e_j\) meaning the difference rep-
  resents a functionality or content that is equivalent. For in-
  stance, in Figure 2b, the red ovals highlight equivalent con-
tent.

- **Addition** (\(D_2\)): \(D_2 : e_i = \emptyset \land \exists e'_j \in p_1 : e'_j = e_j \lor \delta(e'_j, e_j) \|=
  D_1\) meaning the non-empty \(e_x\) in \(\delta\) has a duplicate \(e_y\) in 
the same page, and therefore its addition does not affect the 
overall functionality of the page. For example, in Figure 2c, 
the oval identifies a duplication of an existing functionality.

Definition 4 (New (\(N\))). Given a difference \(\delta(e_i, e_j)\), \(\delta\) represents 
a new functionality or a semantically different content, i.e.: 
\(\delta(e_i, e_j) \equiv N : (e_i = \emptyset ) \land (\exists e'_j \in p_1 : e'_j = e_j \lor \delta(e'_j, e_j) \|=
D_1\).

For example, the search box in Figure 1 is absent in phone de-
description pages and is an example of new functionality.

5.2.3 State-Pair Classification. Following the classification of 
differences described above, we classified state-pairs from a 
functional point of view, in three distinct categories defined as 
follows.

Definition 5 (Functional Clone (\(Cl\))). Given two web pages \(p_1\) 
and \(p_2\), the state-pair \((p_1, p_2)\) is a functional clone (\(Cl\)) if there are 
no semantic, functional or perceptible differences between them, 
defined as \(Cl : \Delta(p_1, p_2) = \emptyset\).

Definition 6 (Functional Distinct (\(Di\))). Given two web pages \(p_1\) 
and \(p_2\), \(p_1\) is functionally distinct from \(p_2\) if there is any semantic or 
functional difference between the two pages, \(Di : \exists \delta(e_1, e_2) \| N\).

Definition 7 (Functional Near-Duplicate (\(Nd\))). Given two 
web pages \(p_1\) and \(p_2\), \(p_1\) is a functional near-duplicate of \(p_2\) if the 
changes between the states do not change the overall functionality being exposed: \(Nd : \Delta \not\subset Cl \land \beta(\delta(e_1, e_2) \| N) \in \Delta\).

We further observed three fine-grained subclasses of near-duplicates in our dataset.

Cosmetic (\(Nd_1\)) when changes related to the aesthetics of the 
webpage such as advertisements or background images occur, which leave the functionalities unaltered (see Figure 2a): \(Nd_1 : \Delta(p_1, p_2) \supset \delta(e_1, e_2) \| U\)

Dynamic data (\(Nd_2\)) when both states of the pair are generated 
from the same template and populated with dynamic data, 
according to a user query or app business logic (see Figure 
2b): \(Nd_2 : \Delta(p_1, p_2) \supset \delta(e_1, e_2) \|= D_1 \cup U\)

Duplication (\(Nd_3\)) when there are additional web elements in a 
page the functionality and semantics of content of which is 
entirely represented within the other page (see Figure 2c): \(Nd_3 : \exists \delta(e_1, e_2) \|= D_2 \in \Delta(p_1, p_2)\)

Following these definitions, we manually labelled the 1,000 
state-pairs in \(RS\), and found 441 clones, 275 near-duplicates (45 
\(Nd_1\), 219 \(Nd_2\), 11 \(Nd_3\)), and 284 distinct pairs.

6 RQ2: CLASSIFICATION OF STATE-PAIRS

6.1 Subject Systems

To address RQ2 (and later RQ3), we need to infer models with dif-
f erent algorithms and thresholds numerous times, which requires 
web apps with deterministic behaviours.
We set out to create manually labelled models for each subject, which we can use as ground truths for comparison of techniques.

To this aim, we selected nine open-source web apps (Table 2) used in previous research of web testing [15, 16, 49, 50], as subjects: Claroline (v. 1.1.5) [7], Addressbook (v. 8.2.5) [40], PPMA (v. 0.6.0) [11], MRBS (v. 1.4.9) [12] and MantisBT (v. 1.1.8) [13] are open-source PHP-based applications while Dimeshif (commit 261166d) [8], Pagekit (v. 1.0.16) [9], Phoenix (v. 1.1.0) [10] and PetClinic (commit 6010d5) [6] are web apps that cover popular JavaScript frameworks Backbone.js, Vue.js, Phoenix/React and AngularJS, respectively.

Note that these nine subject apps are not part of the dataset DS.

### 6.2 Manual Classification (Ground Truth)

We set out to create manually labelled models for each subject, which we can use as ground truths for comparison of techniques.

First, we use CRAWLJAX to create a master crawl model with default depth-first exploration strategy, default state abstraction function based on DOM string equality, and a maximum time budget of one hour, which allow us to capture a large portion of each app’s state space.

Next, we created state-pairs from the states in each model, as follows. The authors of this paper manually classified each state-pair into a clone, near-duplicate (with subcategories) or distinct category, following the same procedure described in Section 5.2. In addition, we also assigned each state to a bin that represents a part of the application’s state space devoted to a certain functionality. As such, each bin is a logical container for all dynamically generated concrete webpages upon crawling (e.g., all webpages related to login). We consider the first concrete instance of a bin B to be a coverage of B by that crawl model. Additional concrete instances of a bin are considered clones or near-duplicates of the bin B.

Table 2 shows the master crawl characteristics for each web app as well as our classification outcome. In the rest of the paper, we refer to the nine master crawls with manually classified 97.5k state-pairs of the nine apps as subject set (SS), and to our manual classification and identified bins as ground truth. Our classification of the subject-set did not find any near-duplicates of category ND2 in SS as the subjects did not feature unrelated changes (U) such as advertisements, commonly found in other kind of websites. MantisBT has the most bins (53), representing a space of five times bigger than that of Phoenix, which has the smallest number of bins (10). Addressbook, PageKit and Phoenix have a high number of near-duplicates of category ND3, differently from the other six.

To study how different near-duplicate categories impact web-app model inference, we group these three subjects referring to them as ND3-Apps and the other six as ND2-Apps.

Table 3 compares the subjects webpage characteristics in terms of DOM size, complexity, and image size to DS. For example, the content of a web page in DS on an average is almost eight times that of the web pages in SS.

### 6.3 Threshold-Based Classification

We aim to evaluate the effectiveness of the near-duplicate detection algorithms in classifying a given pair as either clone, near-duplicate, or distinct. Essentially, this is a multi-class classification problem, which we propose to solve using a classification function \( \Gamma \). Function \( \Gamma \) takes as inputs a near-duplicate detection algorithm \( f \) and computes the distance between two given states in a state-pair \((p_1, p_2)\), classifying the pair to a category according to a threshold-pair \((t_c, t_n)\), as follows:

\[
\begin{align*}
\text{Cl} & : f(p_1, p_2) < t_c \\
\text{D} & : f(p_1, p_2) > t_n \\
\text{Nd} & : \text{otherwise}
\end{align*}
\]

To evaluate \( \Gamma \), we need to find appropriate threshold values for each algorithm that maximize the classification scores.

#### 6.3.1 Threshold Determination

We employ two different approaches, namely, statistical and optimization, to find a suitable threshold-pair \((t_c, t_n)\) for each algorithm. In the statistical approach, we follow a data-based approach in which we use the distance distributions of different classes (Figure 3). In the optimization approach, instead, we determine the thresholds that maximize the classification score on a given labelled set, a commonly adopted strategy in machine learning for hyper-parameters selection of predictive models [47].

**Definition 8 (Statistical Threshold Pair \((St_c, St_n)\)).** Threshold \(St_c\) is the 3rd quartile \((Q_3)\) of the distances calculated by a technique on a given set of clone state-pairs, whereas, threshold \(St_n\) is the median distance on a given set of near-duplicate state-pairs.

**Definition 9 (Optimal Threshold Pair \((O_c, O_n)\)).** Given a labelled set of clones, near-duplicates and distinct state-pairs, the optimal thresholds \(O_c\) and \(O_n\) are retrieved by a Bayesian optimization search that maximizes the average \(F_1\) classification score for \(\Gamma\) over all three classes.

Figure 3 shows the distribution of distance values across the three classes, for each considered algorithm. As the box-plots show,
Table 4: Estimated statistical (\(S_t\)) and optimal (\(O\)) thresholds for clone (c) and near-duplicate (\(n\)) bounds, in dataset DS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>(S_{t,DS})</th>
<th>(S_{n,DS})</th>
<th>(O_{c,DS})</th>
<th>(O_{n,DS})</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLSH</td>
<td>0.00794</td>
<td>0.00794</td>
<td>0.01742</td>
<td>0.07052</td>
</tr>
<tr>
<td>Levenshtein</td>
<td>0.00638</td>
<td>0.01089</td>
<td>0.00704</td>
<td>0.07029</td>
</tr>
<tr>
<td>RTED</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00007</td>
<td>0.04099</td>
</tr>
<tr>
<td>SimHash</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00044</td>
<td>0.00108</td>
</tr>
<tr>
<td>BlockHash</td>
<td>0.00000</td>
<td>0.04082</td>
<td>0.00301</td>
<td>0.13371</td>
</tr>
<tr>
<td>HYST</td>
<td>6.52E-11</td>
<td>1.29E-09</td>
<td>1.15E-09</td>
<td>1.49E-08</td>
</tr>
<tr>
<td>PHASH</td>
<td>0.01754</td>
<td>0.17544</td>
<td>0.04018</td>
<td>0.32232</td>
</tr>
<tr>
<td>SIFT</td>
<td>0.16691</td>
<td>0.27993</td>
<td>0.10192</td>
<td>0.61876</td>
</tr>
<tr>
<td>SSIM</td>
<td>0.01000</td>
<td>0.08800</td>
<td>0.02020</td>
<td>0.15560</td>
</tr>
</tbody>
</table>

Table 5: \(F_1\) Measure for Statistical and Optimal threshold sets

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Statistical ((S_{t,DS},S_{n,DS}))</th>
<th>Optimal ((O_{c,DS},O_{n,DS}))</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(T_S)</td>
<td>(S_S)</td>
<td>Avg</td>
</tr>
<tr>
<td>TLSH</td>
<td>0.50</td>
<td>0.40</td>
<td>0.45</td>
</tr>
<tr>
<td>Levenshtein</td>
<td>0.54</td>
<td>0.46</td>
<td>0.50</td>
</tr>
<tr>
<td>RTED</td>
<td>0.50</td>
<td>0.45</td>
<td>0.47</td>
</tr>
<tr>
<td>SimHash</td>
<td>0.48</td>
<td>0.17</td>
<td>0.33</td>
</tr>
<tr>
<td>BlockHash</td>
<td>0.62</td>
<td>0.54</td>
<td>0.58</td>
</tr>
<tr>
<td>HYST</td>
<td>0.52</td>
<td>0.37</td>
<td>0.44</td>
</tr>
<tr>
<td>PHASH</td>
<td>0.63</td>
<td>0.57</td>
<td>0.60</td>
</tr>
<tr>
<td>SIFT</td>
<td>0.59</td>
<td>0.43</td>
<td>0.51</td>
</tr>
<tr>
<td>SSIM</td>
<td>0.62</td>
<td>0.53</td>
<td>0.57</td>
</tr>
<tr>
<td>Average</td>
<td>0.56</td>
<td>0.44</td>
<td>0.50</td>
</tr>
</tbody>
</table>

While the scores on \(T_S\) can validate these thresholds, scores on \(S_S\) assess the viability of discovering universal thresholds for a near-duplicate detection algorithm for unseen web apps.

6.3.3 Findings (RQ2). Table 5 shows the \(F_1\) classification scores for all techniques on the two labelled sets, \(T_S\) and \(S_S\). As a baseline to compare the techniques, we use a stratified-random-classifier [1] that classifies each state-pair randomly based on proportions of classes in the labelled set.

All evaluated techniques perform better on \(T_S\) than \(S_S\) when universal thresholds are used (+15% on average). This result is not surprising as \(T_S\) is sampled from DS, as well as RS from which we derived these thresholds. \(S_S\), on the other hand, is completely disjoint and different from DS (Table 3).

Although statistical and optimal thresholds have similar overall average \(F_1\) scores (0.50, 0.51), it is important to notice that optimal thresholds perform worse than statistical thresholds on \(S_S\), contrary to expectation.

These two findings essentially indicate that, the distance thresholds for optimal classification of state-pairs can vary based on the characteristics of the particular web app. The thresholds obtained from a labelled data such as RS are therefore, not necessarily applicable for a random unseen web app. Hence, universal thresholds that can classify any given state-pair may not be feasible.

Amongst the techniques, SimHash has the lowest average \(F_1\) score (0.17) on \(S_S\), almost 90% worse than the random baseline.
The results concur with findings of a previous study [29], which points to the fact that the algorithm is poor at distinguishing states that belong to the same app.

On average, five out of top six techniques belong to the computer vision domain. PDIFF is the best with a classification $F_1$ score of $0.60$, $>85\%$ better than the baseline and $>13\%$, $>20\%$ better than Levenshtein and TLSH, the best techniques in DOM and IR categories, respectively. On average, most visual techniques outperform DOM and IR techniques (with the exception of PHash and color-histogram). On SS, PDIFF again outperforms all techniques while BlockHash and SSIM, both visual, are the only other techniques that have an $F_1$ score of more than $0.50$.

7 RQ3: IMPACT ON INFERRED MODELS

With RQ3, we evaluate the impact of the near-duplicate detection algorithms in automated web app model inference.

**RQ3.1:** How can classification thresholds be applied to state abstraction functions (SAFs)?

**RQ3.2:** Can domain knowledge be employed to improve the obtained models?

**RQ3.3:** How does efficiency of SAFs impact the obtained models?

Specifically, we evaluate the quality of crawl models inferred using each of the near-duplicate detection algorithms as state abstract function (SAF) (see Definition 1) along with the determined thresholds. CRAWLJAX already includes all DOM-based algorithms described in Section 3.2; we added the computer vision and information retrieval near-duplicate algorithms within CRAWLJAX as SAFs. More specifically, we integrated the implementations of PDiff, SIFT, and SSIM from the open-source computer-vision library OpenCV, and the publicly available versions of TLSH$^2$ and simhash.$^3$

Since we need to run and analyze many crawl sessions (i.e., nine apps, 10 algorithms, different threshold sets), we limit the crawl session with a maximum runtime of five minutes.

**Model Quality:** We measure the quality of a generated model through its $F_1$ score, the harmonic mean of $Pr$ and $Re$. Lower precision ($Pr$) denotes a greater redundancy in the model and is computed as the ratio of unique states (bins) covered by the model to the total number of states in the model. Recall ($Re$) quantifies the application state coverage achieved in the model and is computed as the number of bins covered by the model to the total number of bins identified by humans, for the corresponding app, in the ground truth (see Section 6.2).

The recall $Re$ of a crawl model is highly dependent on the ability of the SAF to reliably distinguish the distinct state-pairs and its precision $Pr$ on its ability to exclude near-duplicates and clones of states already present from the model. Crawlers, however, typically expect one single similarity threshold for deciding if a state is new to be added to the model; i.e., they do not distinguish between clone/near-duplicate. Therefore, we frame the problem of finding optimal thresholds for a SAF as maximizing the $F_1$ score of its distinct-pair detection.

### Table 6: Distinct pair ($Pr$, $Re$, $F_1$) on existing datasets

<table>
<thead>
<tr>
<th></th>
<th>$TS$</th>
<th></th>
<th></th>
<th>$SS$</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$Pr$</td>
<td>$Re$</td>
<td>$F_1$</td>
<td>$Pr$</td>
<td>$Re$</td>
<td>$F_1$</td>
</tr>
<tr>
<td>$O_n$</td>
<td>0.61</td>
<td>0.81</td>
<td>0.80</td>
<td>0.89</td>
<td>0.53</td>
<td>0.64</td>
</tr>
<tr>
<td>$SS_n$</td>
<td>0.65</td>
<td>0.90</td>
<td>0.75</td>
<td>0.87</td>
<td>0.76</td>
<td>0.78</td>
</tr>
</tbody>
</table>

#### 7.1 Thresholds for SAFs (RQ3.1)

Before we employ the near-duplicate techniques as SAFs in crawling and evaluate the generated models, which is a manual and time consuming process, we assess the techniques and the universal thresholds based on the $F_1$ score of the distinct-pair detection, which indicates the applicability of the techniques as SAFs.

**Findings (RQ3.1).** In the distinct state-pair detection scores from RQ2 shown in Table 6, scores on $TS$ allow us to assess the ability of a technique to distinguish distinct state-pairs in the wild, while $SS$ lets us simulate each technique as a SAF on generated models captured in our subject-set. In contrast to the RQ2 results, where both the threshold sets had better average classification $F_1$ on $TS$ compared to $SS$, Table 6 shows that statistical threshold had better distinct state-pair detection $F_1$ of $0.78$ on $SS$ than $0.73$ in $TS$. Optimal threshold $O_{n, DS}$, which is higher/stricter than $SS_{n, DS}$, in terms of actual threshold value, as shown in Table 4, has a poor recall on $SS$ (53%) compared to $TS$ (81%). Also in $TS$ statistical threshold has the highest recall, but by sacrificing precision; the optimal threshold emerges with a better overall $F_1$ score through a 25% better precision on $TS$. The same threshold, however, could not improve precision in $SS$ but has 50% lower recall.

As we optimized our threshold to be stricter to fit the distribution in $DS$, we ended up misclassifying distinct pairs to be near-duplicates in $SS$ because of the differences in the distributions between the two data-sets. As we pointed out in RQ2, these results show the infeasibility of finding universal thresholds as the distances for state-pairs are highly influenced by the intrinsic characteristics of the web app they belong to.

#### 7.2 Using Application Knowledge (RQ3.2)

These results for universal thresholds prompted us to investigate whether having knowledge of the web app characteristics helps in selecting better thresholds to improve the detection rates of the techniques.

We use the manually labelled models (see Section 6.2) in the subject-set ($SS$) for each app to represent application knowledge. In order to use this application knowledge, we apply the near-duplicate threshold definitions in Definition 8 and Definition 9 to each subject in $SS$ to derive $SS_{n, SS}$ and $O_{n, SS}$ respectively. In addition to these two thresholds, through initial experiments, we have observed that category $Nd_3$ near-duplicates overlap with distinct (Di) pairs and it is not possible to design a threshold that can distinguish them. We therefore created a new threshold definition that sacrifices the precision of distinct pair detection by allowing misclassification of $Nd_3$ near-duplicates as $Di$ for better recall ($Re$).

---

$^2$https://github.com/albertjuhe/charikars_algorithm

$^3$https://github.com/idealista/tlah
Table 7: Inferred model \(F_1\) score

<table>
<thead>
<tr>
<th></th>
<th>Universal</th>
<th>App-Specific</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(S_{\text{DS}})</td>
<td>(O_{\text{DS}})</td>
</tr>
<tr>
<td>AddressBook</td>
<td>0.33</td>
<td>0.27</td>
</tr>
<tr>
<td>PetClinic</td>
<td>0.36</td>
<td>0.25</td>
</tr>
<tr>
<td>Chat</td>
<td>0.30</td>
<td>0.18</td>
</tr>
<tr>
<td>DimEmit</td>
<td>0.31</td>
<td>0.22</td>
</tr>
<tr>
<td>PageKit</td>
<td>0.30</td>
<td>0.27</td>
</tr>
<tr>
<td>Phoenix</td>
<td>0.44</td>
<td>0.29</td>
</tr>
<tr>
<td>PPMA</td>
<td>0.31</td>
<td>0.19</td>
</tr>
<tr>
<td>MTRB</td>
<td>0.37</td>
<td>0.35</td>
</tr>
<tr>
<td>ManimBT</td>
<td>0.24</td>
<td>0.18</td>
</tr>
<tr>
<td>Average</td>
<td>0.33</td>
<td>0.24</td>
</tr>
</tbody>
</table>

Table 8: Inferred model \(F_1\) for each algorithm for selected thresholds

|          | All\(\_\text{DS}\) | \(O_{\text{DS}}\) | \(S_{\text{SS}}\) | \(O_{\text{SS}}\) | \(S_{\text{TLSH}}\) | \(O_{\text{TLSH}}\) | \(S_{\text{PHASH}}\) | \(O_{\text{PHASH}}\) | \(S_{\text{RTED}}\) | \(O_{\text{RTED}}\) | \(S_{\text{BlockHash}}\) | \(O_{\text{BlockHash}}\) | \(S_{\text{Hash}}\) | \(O_{\text{Hash}}\) | \(S_{\text{SimHash}}\) | \(O_{\text{SimHash}}\) | \(S_{\text{SSIM}}\) | \(O_{\text{SSIM}}\) |
|----------|-------------------|------------|------------|------------|-------------------|------------|-------------------|------------|-------------------|------------|-------------------|------------|-------------------|------------|-------------------|------------|-------------------|------------|-------------------|
| AddressBook | 0.10 | 0.05 | 0.07 | 0.04 | 0.06 | 0.04 | 0.09 | 0.06 | 0.13 | 0.11 | 0.13 | 0.11 | 0.11 | 0.11 | 0.11 | 0.11 | 0.11 | 0.11 |
| PetClinic | 0.16 | 0.07 | 0.07 | 0.05 | 0.08 | 0.05 | 0.08 | 0.05 | 0.10 | 0.09 | 0.10 | 0.09 | 0.10 | 0.09 | 0.10 | 0.09 | 0.10 | 0.09 |
| Chat | 0.10 | 0.04 | 0.04 | 0.03 | 0.04 | 0.03 | 0.04 | 0.03 | 0.05 | 0.04 | 0.05 | 0.04 | 0.05 | 0.04 | 0.05 | 0.04 | 0.05 | 0.04 |
| DimEmit | 0.09 | 0.02 | 0.02 | 0.01 | 0.02 | 0.01 | 0.02 | 0.01 | 0.04 | 0.03 | 0.04 | 0.03 | 0.04 | 0.03 | 0.04 | 0.03 | 0.04 | 0.03 |
| PageKit | 0.11 | 0.04 | 0.04 | 0.03 | 0.04 | 0.03 | 0.04 | 0.03 | 0.05 | 0.04 | 0.05 | 0.04 | 0.05 | 0.04 | 0.05 | 0.04 | 0.05 | 0.04 |
| Phoenix | 0.10 | 0.02 | 0.02 | 0.01 | 0.02 | 0.01 | 0.02 | 0.01 | 0.04 | 0.03 | 0.04 | 0.03 | 0.04 | 0.03 | 0.04 | 0.03 | 0.04 | 0.03 |
| PPMA | 0.11 | 0.04 | 0.04 | 0.03 | 0.04 | 0.03 | 0.04 | 0.03 | 0.05 | 0.04 | 0.05 | 0.04 | 0.05 | 0.04 | 0.05 | 0.04 | 0.05 | 0.04 |
| MTRB | 0.11 | 0.04 | 0.04 | 0.03 | 0.04 | 0.03 | 0.04 | 0.03 | 0.05 | 0.04 | 0.05 | 0.04 | 0.05 | 0.04 | 0.05 | 0.04 | 0.05 | 0.04 |
| ManimBT | 0.11 | 0.04 | 0.04 | 0.03 | 0.04 | 0.03 | 0.04 | 0.03 | 0.05 | 0.04 | 0.05 | 0.04 | 0.05 | 0.04 | 0.05 | 0.04 | 0.05 | 0.04 |

Table 7 shows the average \(F_1\) scores for each algorithm for five minute crawls on our subjects. **RTED** consistently outperforms other techniques with an \(F_1\) score of 0.62 averaged over all five thresholds. It is 29% better than Levenshtein, the next best algorithm.

The results for visual techniques in Table 8 are contrary to our expectation, given that, in RQ2, they convincingly outperformed the DOM and IR techniques in state-pair classification using \(\Gamma\). Apart from being slow compared to DOM based algorithms as shown in Table 9, visual techniques, rely on characteristics that cannot directly capture differences corresponding to web elements (e.g., SIFT keypoints). Techniques such as RTED, which use a DOM characteristic on the other hand, can reliably capture differences in individual web elements between given two web pages, essential to be able to classify states similar to a human tester.

In IR techniques, SimHash is not able to distinguish even two completely different states in our subject-set as already seen in RQ2. TLSH on the other hand, fails to calculate digests for app states of our subjects due to lack of enough complexity as shown in Table 3 — the content in our subjects is 1/9th of the content size in the wild. Therefore, we exclude SimHash and TLSH from further analysis.

7.3 Impact of Efficiency (RQ3.3)

An analysis of visited states per minute or **speed** of the algorithms, shown in Table 9, seems to suggest that faster algorithms such as **RTED** (25 states per minute) could explore more states in a given crawl time and improve its **Re** whereas, slower algorithms such as **PDiff**, which could only explore **four** states per minute on an average are at a clear disadvantage.

Table 8 shows that for all remaining eight techniques with the exception of SIFT, \(O_{\text{SS}}\) for Nd2-Apps and \(S_{\text{n3 SS}}\) for Nd3-Apps is the best threshold configuration.

Table 9 shows the statistics of the 5-min crawls for each technique with their best threshold configuration. Coverage (**Re**) data suggests that 5 minutes was not enough to cover all of the app state-space. Therefore, we experiment with a longer crawl time, i.e., 30 minutes. Given the exponential nature of increase in manual effort to analyze larger crawl models, we limit this experiment to the best performing techniques tuned with thresholds from the best 5-minute crawls presented in Table 9. We select the top four techniques based on \(F_1\) scores, however, as discussed before, since the slower algorithms were placed at a disadvantage in the 5-minute crawls, we also include PDiff and SSIM that produced models with the best precision (**Pr**) scores of 0.91 and 0.85 (respectively 12% and 6% better than RTED which has the best \(F_1\) score of 0.66).

Findings (RQ3.3). Average \(F_1\) scores shown in Table 10 for 30 minute crawls indicate that, when tuned correctly and given enough time, Histogram, BlockHash, RTED and Levenshtein can all perform well on Nd2-Apps meaning that they managed to discard near-duplicates of type \(N_{d_{2}}\) reasonably well. However, it is surprising to see that PDiff and SSIM score higher than all of them on Nd3-Apps. Thus, we decided to analyze how \(F_1\) has changed over the 30 minutes for Nd3-Apps as opposed to the Nd2-Apps.
This threat selecting real-world web apps with different sizes, pertaining to different domains, and adopted in previous web testing work [15, 16, 49]. Another threat concerns the selection of thresholds for near-duplicate detection techniques, whose results may not generalize to other algorithms. We mitigated this threat by selecting 10 techniques from three different domains: web testing, computer vision and information retrieval. Internal validity threats concern uncontrolled factors that may have affected our results. A possible threat is represented by the manually created ground truth, which was unavoidable because no automated method could provide us with the ideal classification of web pages. To minimize this threat, the authors of this paper created, in isolation, a ground truth. Then, the two established a discussion to produce a single ground truth for each web app.

For reproducibility of the results, we made our tool, datasets and used subject systems available [5], along with required instructions.

9 RELATED WORK

A large body of research has addressed the analysis of web sites structure via clustering for clone detection and duplicate removal of web pages [18, 19, 22–25, 29, 34, 41, 56].

Henzinger [29] performed an evaluation of two near-duplicate detection algorithms based on shingling on a large dataset of 1.6B web pages. Manku et al. [34] followed up on the work using simhash to detect near-duplicates for web information retrieval, data extraction, plagiarism and spam detection with promising results. Fetterly et al. [23] study the evolution of near-duplicate web pages over time and conclude that near-duplicates have little variability over time, and two pages that have been found to be near-duplicates of one another will continue to be so for the foreseeable future.

Our study is different from the above work as we aim to detect near-duplicates within web apps and not across different web apps. Regarding detection of within app near-duplicates, Calefato et al. [19] propose a method to identify near-duplicates as well as functional clone web pages based on a manual visual inspection of the GUI. Crescenzi et al. [22] propose a structural abstraction for web pages as well as a clustering algorithm that groups web pages based on this abstraction. Di Lucca et al. [24, 25] evaluate the Levenshtein distance and the tag frequency methods for detecting near-duplicate web pages. Eyk et al. apply simhash and broders near-duplicate detection within Crawljax [27].

THREATS TO VALIDITY

External validity threats concern the generalization of our findings. We considered only nine web apps and experiments with other subject systems are necessary to fully confirm the generalizability of our results, and corroborate our findings. We tried to mitigate

| Table 9: Techniques Speed and Inferred model (Re, Pr, F₁) for best 5-minute crawls |
|----------------|----------------|----------------|----------------|----------------|----------------|
|               | Levenshtein    | RTED           | BlockHash      | PHASH          | HIST           |
| Speed         | 11             | 23             | 17             | 16             | 16             |
| Recall        | 0.42           | 0.61           | 0.49           | 0.49           | 0.55           |
| Precision     | 0.84           | 0.79           | 0.75           | 0.79           | 0.72           |
| F₁            | 0.54           | 0.66           | 0.54           | 0.52           | 0.58           |

| Table 10: Inferred model F₁ for 30-Minute crawls |
|----------------|----------------|----------------|----------------|----------------|
|               | BlockHash      | HIST           | Levenshtein    | PMHF           | RTED           | SSIM           |
| Apps          |                |                |                |                |                |
| All           | 0.51           | 0.57           | 0.53           | 0.52           | 0.62           | 0.56           |
| Nd₂           | 0.57           | 0.62           | 0.59           | 0.51           | 0.66           | 0.52           |
| Nd₃           | 0.39           | 0.47           | 0.42           | 0.36           | 0.52           | 0.64           |

A plot of F₁ of the model over its states percentage for RTED crawls is shown in Figure 4. The figure highlights that for Nd₃-Apps, the model deteriorates as states being added are near-duplicates, mostly of type Nd₃, while, the models of Nd₂-Apps seem to stabilize as Nd₃ near-duplicates are being detected and discarded. During the manual analyses of models, we observed that the Nd₃ near-duplicates are dynamically created, typically through user-interactions that result in addition/removal of web elements whose functionality already exists in the state (e.g., addition/deletion of new rows in a table). Not only is this newly created state a near-duplicate that will eat into precious testing time, but each time the crawler revisits this state, it may invoke the same creation path adding even more duplicates resulting in a never-ending loop.

Efficiency may negatively impact the generated model in time-limited crawls for Nd₃ apps.

Given that RTED is the best algorithm and was fine-tuned to produce best model for each application, this surprising revelation points to the limitation of existing crawlers and threshold based SAFs and shows that threshold based crawling may never produce an accurate and complete model of modern web apps with dynamic Nd₃ near-duplicates. We therefore think that future SAFs should incorporate characteristics that represent functionality and crawlers should be designed to utilize near-duplicate detection to establish the nature of duplication instead of quantifying the computed differences to actively guide the exploration to discover newer functionality.

8 THREATS TO VALIDITY

External validity threats concern the generalization of our findings. We considered only nine web apps and experiments with other subject systems are necessary to fully confirm the generalizability of our results, and corroborate our findings. We tried to mitigate this threat selecting real-world web apps with different sizes, pertaining to different domains, and adopted in previous web testing work [15, 16, 49]. Another threat concerns the selection of thresholds for near-duplicate detection techniques, whose results may not generalize to other algorithms. We mitigated this threat by selecting 10 techniques from three different domains: web testing, computer vision and information retrieval. Internal validity threats concern uncontrolled factors that may have affected our results. A possible threat is represented by the manually created ground truth, which was unavoidable because no automated method could provide us with the ideal classification of web pages. To minimize this threat, the authors of this paper created, in isolation, a ground truth. Then, the two established a discussion to produce a single ground truth for each web app.

For reproducibility of the results, we made our tool, datasets and used subject systems available [5], along with required instructions.
In mobile testing research, researchers [3, 14] used mobile GUI widget hierarchies in order to design optimal state abstractions. Our study did not consider such techniques as they are not directly applicable for web applications.

To the best of our knowledge, our work is the first one to study different near-duplication detection algorithms (from different fields) as SAFs in a web crawler. This paper is the first to propose a systematic categorization of near-duplicates in web apps, from a functional E2E testing perspective and to study the impact of near-duplicate detection on generated web application models and web testing. Moreover, our paper is the first to discuss selection of thresholds for near-duplicate detection, an important first step.

### 10 CONCLUSIONS AND FUTURE WORK

Automatically asserting the equality of two complex web pages is a difficult problem, which the state abstraction function of a crawler needs to solve at runtime during the exploration. The problem is further complicated by the presence of near-duplicates that need to be detected and mapped to the logical pages in order to produce meaningful crawl maps.

We study ten existing near-duplicate detection techniques from three different domains and compare their effectiveness as state abstraction functions in a crawler. Our results show that near-duplicates characterized by dynamic data, as categorized in the study, are detectable when application knowledge is employed. However, near-duplicates characterized by duplication of web elements, that are often a by-product of state exploration, cannot be handled by threshold-based model inference.

Future work includes devising novel types of abstraction functions, incorporating both page structural and visual characteristics in a single hybrid solution to detect different kinds of near-duplicates.

### REFERENCES


