Testing Apps With Real World Inputs

Tanapuch Wanwarang
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
tanapuch.wanwarang@cispa.saarland

Nataniel P. Borges Jr.
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
nataniel.borges@cispa.saarland

Leon Bettscheider
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
s8lnbett@stud.uni-saarland.de

Andreas Zeller
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
zeller@cispa.saarland

ABSTRACT

To test mobile apps, one requires *realistic* and *coherent* test inputs. The Link approach for Web testing has shown that *knowledge bases* such as DBPedia can be a reliable source of semantically coherent inputs. In this paper, we adapt and extend the Link approach towards *test generation for mobile applications:*

(1) We identify and *match* descriptive labels with input fields, based on the Gestalt principles of human perception;
(2) We then use *natural language processing techniques* to extract the *concept* associated with the label;
(3) We use this concept to *query* a knowledge base for *candidate input values;*
(4) We cluster the UI elements according to their functionality into *input* and *actions*, filling the *input* elements first and then interacting with the actions.

Our evaluation shows that leveraging knowledge bases for testing mobile apps with realistic inputs is effective. On average, our approach covered 9% more statements than randomly generated text inputs.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Dynamic analysis*; • **Human-centered computing** → Graphical user interfaces; Smartphones.

KEYWORDS

Automated Testing, Input Generation, Knowledge-Base, Android

ACM Reference Format:
Tanapuch Wanwarang, Nataniel P. Borges Jr., Leon Bettscheider, and Andreas Zeller. 2020. Testing Apps With Real World Inputs. In *AST '20: 1st IEEE/ACM International Conference on Automation of Software Test, May 25–26, 2020, Seoul, South Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Mobile applications (apps) that take complex data as input, such as travel bookings, maps, or online banking forms, are part of our everyday life. They require realistic and coherent test inputs to be tested adequately. These inputs are, however, expensive to generate manually and challenging to synthesize automatically.

AST '20, May 25–26, 2020, Seoul, South Korea
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

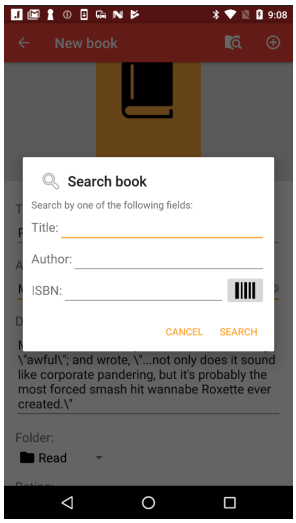


Figure 1: Book search functionality in an app. It requires syntactically and/or semantically correct values to be tested

Consider the example in Figure 1. To search for a book online, the user is required to type an author, title, or ISBN. While it is possible to input virtually any value for author and title, the ISBN must contain only numbers and must be 10 or 13 digits long to bypass syntactic validation rules.

Now consider an *automated test generator* being used to test this app. To explore the app's functionality, a test generator would systematically identify the user interface elements and interact with them. However, to find a book (and to explore the functionality associated with having found a book), it must first either input an existing ISBN or a valid combination of author name and book title. Randomly generated author and book names, or ISBNs, however, are unlikely to produce any results, and instead, fail to reach code regions located beyond input validation checks.

Even if the test generator were able to bypass the syntax validation rules, the generated values would rarely be semantically meaningful. Even though the ISBN may be valid in itself, a random valid ISBN still is unlikely to point to some existing book.

Currently, these scenarios are handled by using a *curated set of inputs*, such as dictionaries, or by manually written values for specific inputs. Both approaches are laborious, expensive, and subject

to human bias, undermining two of the main benefits of automated testing.

The past research of Mariani et al. [27] indicated that *knowledge bases* could be a reliable source of semantically coherent inputs. Their Link tool would query the *DBPedia* data collection to identify data to be used in the tests. They then manually used the extracted data to generate complex system test inputs. If a field required a "ZIP" code, for instance, Link would query for "ZIP" codes from *DBPedia*.

When Link was conceived, it targeted desktop and web applications. But would such an approach also work for mobile devices? Due to limited screen size, mobile apps have different UI design patterns [12, 14], which diverge from those used on desktops, making a Link-like approach less accurate. However, its core strategy to *query a knowledge base for input values* might be applicable for mobile devices too.

In this work, we investigate whether a Link-like approach would help to generate better tests for Android apps, and if so, in what way. Our approach consumes data from a knowledge base and uses these values during text generation in a user-like order. Specifically, we assess:

RQ1 Can semantic concepts be accurately associated with input fields?

RQ2 Can syntactically valid and semantically coherent textual inputs values be automatically extracted from a knowledge base?

RQ3 Can textual inputs automatically extracted from a knowledge base improve test generation?

The remainder of this work is organized as follows. After discussing the approaches and tools used in this work (Section 2), we make the following contributions:

(1) We present a *set of metrics* that effectively associate descriptive elements with input fields, tailored to Android-specific design guidelines, and thus *extend Link for use in mobile test generation* (Section 3).
(2) We evaluate our approach (Section 4) and show that:
   - **Concepts can be associated with input fields** with 87% precision.
   - **About three out of four queries to the knowledge base returned valid results**, 94% of which were semantically valid.
   - **Automatically generated input values used in a user-like order can improve tests**. In our experiments, we observed **an average improvement of 9% in statement coverage** compared to random tests.

After discussing limitations and threats to validity (Section 5) as well as related work (Section 6), Section 7 prescribes future work and concludes our paper.

2 BACKGROUND

Mariani *et al.* [27] proposed Link to query input values from a knowledge base. Their work exploited the metrics based on the Gestalt principles [23] of how humans perceive objects and patterns to associate descriptor labels to input fields. In this section, we describe the underlying principles or their approach, which we adapted according to the peculiarities of mobile apps.

2.1 Associating Descriptor Labels to Input Fields

Link relied on the Gestalt principles of visual perception to associate descriptor labels with input fields on desktop GUIs. It namely used the metrics of *Proximity*, *Homogeneity*, and *Closure*, as implemented by Becce *et al.* [7]. These metrics work as follows:

Proximity Humans associate elements which are close to each other. Besides, app UIs are developed to be explored from left to right and top to bottom. Therefore, this metric dictates that descriptive labels must be located on the left or top of an input field.

Homogeneity Elements should be distributed according to their semantics. On UI design, the regular distribution of UI elements is mostly done through their alignment. This metric dictates that a label should be either vertically or horizontally aligned to the input field.

Closure Semantically correlated elements should be grouped together for easier comprehension. Therefore, this metric dictates that semantically correlated UI elements should be placed in the same container.

2.2 Querying Knowledge Bases

Linked Data [8, 37] describes how to define and publish machine-readable typed links between arbitrary items on the Web so that it is interlinked and accessible through semantic queries. It is used extensively in different topical domains, including Media, Government, Publications, etc. [32]. It builds upon standard Web technologies such as Resource Description Framework (RDF) and Uniform Resource Identifiers (URIs).

RDF specification[1] describes how to connect concepts using triples containing a subject, a predicate, and an object. URIs [29] are used to describe properties in the RDF triples. For example, the URI `http://dbpedia.org/resource/London` identifies the city of London, the URI `http://dbpedia.org/page/England` denotes the country of England, and the URI `http://dbpedia.org/ontology/isPartOf` describes that a subject is a part of an object. Therefore, the RDF triple: (`http://dbpedia.org/resource/London`, `http://dbpedia.org/ontology/isPartOf`, `http://dbpedia.org/page/England`) represents the fact that Londons is a part of England.

Link exploited this structure to query for complex input values, based on the UI semantics while maintaining the semantic coherence between the inputs. It first queries DBPedia [5] classes and predicates for resource URIs whose name match those obtained in the label matching step. Link uses WordNet to search for synonyms when it cannot find any class or predicate with the exact word queried.

It then associates the discovered elements by systematically querying for resources that occur as the subject of both elements. If a resource exists, the elements are merged into a single query. Otherwise, they are kept disjoint. Finally, Link queries DBPedia for resources to obtain resources.

[1] https://www.w3.org/TR/rdf-concepts/

3 METHOD

We propose an approach with four steps, namely: *description matching*, *concept extraction*, *input value acquisition* and *input value consumption*, as illustrated in Figure 2.

Given a UI state, we start by identifying and *matching descriptive labels* with input fields, using a modified version of Becce's metrics adapted to mobile apps. We then use natural language processing (NLP) techniques [26] to *extract the concept* associated with the label. We use the extracted concepts, instead of the original labels, to *query* knowledge bases for input values. Finally, we fill all *input* elements with the queried values and randomly interact with the non-input elements.

3.1 Matching Labels

To match descriptor labels with input fields, we extend Becce's metrics to support the peculiarities of mobile apps. We reuse their metrics of *Proximity*, *Homogeneity*, and *Closure* on the Android Window Hierarchy dump [4], which is similar to the DOM structure used in Becce's original work.

Becce's metrics are, however, based on the idea that descriptor labels and input fields are distinct elements. This approach works on web and desktop apps because the hint text is frequently used to exemplify or assist the user in filling the input field, not to describe its meaning. This does not hold on mobile apps. Due to limited screen size, mobile apps frequently reuse the input element for descriptive proposes, through its hint text, as shown in Figure 3.

We thus add an *Enclosure* metric which combines the Gestalt principles of *proximity* and *closure*. Our metric is defined as follow:

Enclosure Input fields can describe themselves to mitigate UI space requirements. Therefore, a UI element describes the other if it is contained within the other.

With this new metric, we produced an algorithm, shown in Algorithm 1, to match a label with an input field on mobile apps. In principle, our matching algorithm is a map function MATCH*(field, state)* → *concept*, which receives an input field and a UI state and returns a concept. For our abstraction, we consider as a UI state the set of all UI elements on an app screen.

Algorithm 1 Matching of an input field to a concept

```
1:  function MATCH(field, state)→ concept
2:      if (field, state) ∉ memory then
3:          if hasText(field) and hasNoun(field) then
4:              label ← field                                  ▷ Enclosure
5:          else
6:              label ← becce(field, state) ▷ Proxim., Homogen. and Closure
7:          end if
8:          result ← concepts(label)
9:          memory ← ((field, state), label)
10:     else
11:         result ← memory(field, state)
12:     end if
13:     return result
14: end function
```

Our algorithm starts by checking if the input field to be matched has not already been processed (line 2). This check is necessary because our *enclosure* metric uses the hint text as a descriptor. Android does not know if an input field has been filled or not; that is, it does not differentiate between the hint text and a typed value on an input field. Moreover, it is no longer possible to determine the original label of an input field once it is filled. We thus create a *memory* with all previously encountered input fields, alongside their matched labels. This allows us to reuse the original label description in case the field gets filled in the future.

If the input field is in the memory, we simply return the previously mapped concept (line 11). Otherwise, if the input field is being processed for the first time, we match it to a concept. We first attempt to match it with our *enclosure* metric. If the input field has a text and this text contains a noun (line 3), we define this text as its label (line 4). We consider only labels that contain a noun as candidates descriptors because nouns are used to define objects. Otherwise, we apply Becce's original metrics (line 6) to search for the most relevant label descriptor. Finally, we extract the label's concept, according to Section 3.2, and add it to our cache, preventing the input field from being mapped again (line 8-9).

3.2 Extracting Concepts

Since we match input fields to both external widgets, as well as hint texts, a label can contain information such as *City* or *Name (required)* as previously shown in Figure 3, or more complex information such as *Enter your username* or *Type a location*. As a consequence, we must pre-process the label to extract a concept from it. Our approach is shown in Algorithm 2.

We employ natural language processing (NLP) techniques [26] to extract the concept of a label. We first use *part-of-speech tagging* [28] to identify all nouns of a label (line 2). We then take the first noun as the candidate concept (line 3). We use lemmatization [34] to reduce this noun to its inflectional form and query the available classes and predicates of the knowledge base using this lemma (lines 4-5). If the lemma is found in the knowledge base (line 6), we use it as the label's concept. Otherwise, we search for synonyms in a dictionary (line 7) and systematically query the knowledge base for each synonym. We repeat this process until a result is found in the knowledge base, or there are no synonyms left (lines 8-13). If we found a result for any synonym, we return the result as the label's concept (line 11). If we did not, we proceed to the next noun (line 3). If we do not find any result for any of the nouns, we return an empty label (line 18).

3.3 Obtaining Input Values

Once we extracted all input fields and their concepts, we leverage LINK to obtain semantically aware input values to use during testing. LINK consumes our concepts and uses the knowledge base to identify the largest subset of concepts which are interconnected. By identifying elements which are interconnected, it ensures the semantic coherence of part of the inputs.

Consider our motivating example, LINK can associate the concepts: *author*, *title* and *ISBN* and query for semantically coherent values, such as (Sun Tzu, The Art of War, 9781590302255). It is not, however, able to associate the term *publisher* with them. We overcome this limitation by recursively using LINK. We query the knowledge base until all concepts which exist in the knowledge

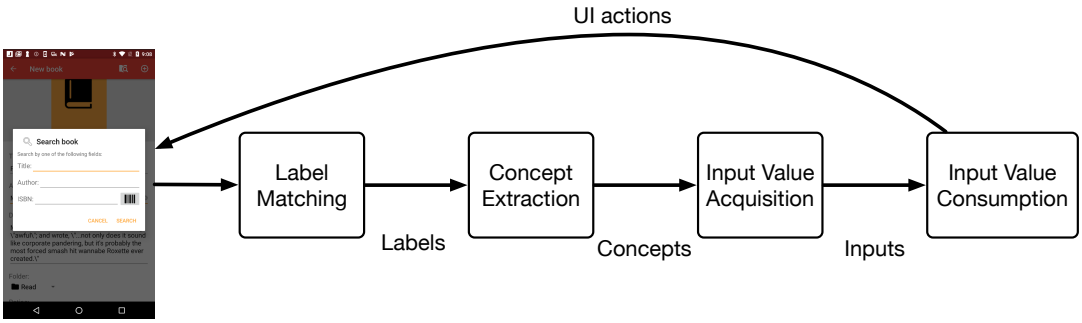


Figure 2: Approach overview diagram. Associate input fields and labels elements, extract the label's concepts and query for input values. Finally, use queried values to fill input fields during testing

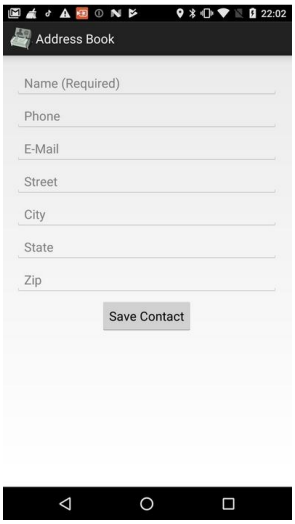


Figure 3: Self-explanatory input fields. Due to limited space, Android apps frequently use the hint text property to describe the input field

base produce input values, either interrelated to other concepts or independently. We summarize our approach to query a knowledge base for input values in Algorithm 3.

Our approach starts with a set of concepts to query, and it recursively queries the knowledge base until all concepts have been used. It first uses LINK's concept association feature to obtain the largest set of interrelated concepts (line 2). It then invokes LINK to get input values for these concepts (line 3). If there are remaining concepts to query, it recursively invokes itself, passing only the remaining concepts (lines 4-6). Finally, it returns the list of queried input values (line 7).

3.4 Consuming Input Values

Users expect app functionality to be triggered when interacting with specific types of UI elements [17]. They expect apps to trigger some functionality when they press a button or click on an image. They seldom expect anything but input validation to happen when they enter data on an input field. Moreover, app UIs are designed to guide the user towards specific flows, making the app more intuitive to use. Users, for example, fill forms sequentially, with apps guiding them to the next field after entering a value. Under these premises, we intuitively split the UI elements into two categories: input and non-input fields.

To test an app UI, we then first enter values in all input fields for which we successfully queried an input value. We fill the input fields from top to bottom and left to right, to emulate the behavior of a user. Once we have filled all possible input fields, we randomly interact with the remaining UI elements to access functionality.

Algorithm 2 Extracting the concept of a label

```
1:  function CONCEPTS(label)→ concept
2:      tagged ← part-of-speech(label)
3:      for candidate in nouns(tagged) do
4:          lemma ← lemmatization(candidate)
5:          value ← link(lemma)
6:          if |value| = 0 then
7:              synonyms ← synonyms(lemma)
8:              for synonym in synonyms do
9:                  value ← link(synonym)
10:                 if |value| > 0 then
11:                     return value
12:                 end if
13:             end for
14:         else
15:             return value
16:         end if
17:     end for
18:     return ∅
19: end function
```

Algorithm 3 Querying a knowledge base for candidate input values for a set of concepts

```
1: function QUERY(concepts)→ input values
2:     largest-set ← link-associate(concepts)
3:     values ← link(largest-set)
4:     if | (concepts \ largest-set) | > 0 then
5:         values ← QUERY(concepts \ largest-set)        ▷ Recursion
6:     end if
7:     return values
8: end function
```

4 EVALUATION AND EXPERIMENTS

In this work, we aim to gather empirical evidence that syntactically correct and semantically coherent input values can be automatically generated and used to improve automated Android testing. More specifically, we aim to answer the following research questions:

RQ1 (Associating and Extracting Concepts) Can semantic concepts be accurately associated with input fields?

RQ2 (Obtaining Input Values) Can syntactically valid and semantically coherent textual inputs values be automatically extracted from a knowledge base?

RQ3 (Consuming Input Values) Do textual inputs, automatically extracted from a knowledge base, improve test generation?

4.1 Experimental Setup

To evaluate our approach we implemented Saigen (Semantic Aware Input Generator) as DroidMate-2 [10] plug-in. DroidMate-2 is an open-source Android test input generator that can be used out of the box on Android devices or emulators executing with operating system version 6.0 or newer. To identify the UI states and elements on the app, we relied on DroidMate-2's uniqueness measurement, which allows the same UI element to be re-identified between different states.

In our experiments, we consider as input fields only UI elements of class *android.widget.TextView*, Android's native input field. We used as a knowledge base DBPedia [5]—a structured source of information gathered from Wikipedia—and we used WordNet as a synonym dictionary.

We previewed 120 Android apps on Google Play Store and F-Droid [20] from June to July 2018. We then filtered out those apps with less than 10,000 downloads and remained with 85 apps. We explored these 85 apps using DroidMate-2's random strategy for 500 actions and filtered out those in which the exploration did not reach any native Android text field. We used 500 actions as a limit as previous work [10] showed only a marginal discovery of new functionality after this point.

After these filtering steps, our dataset contained 20 apps across different domains, including travel, music, tools, books, games, business, and cars. These apps (henceforth *test set*) and their information are shown in Table 1.

We executed all experiments on a set of Google Nexus 5X and Google Pixel XL devices, running Android 7.1.2. To prevent device-dependent behavior, all tests for the same app were executed on the same device.

4.2 RQ1: Associating and Extracting Concepts

Our first research question aims to measure the accuracy of our label matching and concept extraction approaches, as they have a high impact on our remaining studies. With this goal, we re-executed DroidMate-2's default exploration strategy for 500 actions on all 20 apps from the *test set*, while recording (screenshot) all input fields found and their matched label descriptors[2]. We then manually classified each input field found according to the following rules:

[2]We ignored fields with the following concepts: username, password and email, as they are intentionally not available on the knowledge base.

Table 1: Selected Applications for the experiment (M for Millions)

Name	Domain	Downloads
Trip.com	Travel	1M+
Booking.com	Travel	100M+
Agoda	Travel	10M+
Book Catalogue	Books	100,000+
Yelp	Travel	10M+
Kayak	Travel	10M+
Arnab	Tools	100,000+
Youtube Music	Music	50M+
Lonely Planet Guides	Travel	500,000+
TripAdvisor	Travel	100M+
Airbnb	Travel	10M+
Expedia	Travel	10M+
My Books - Library	Books	50,000+
CLZ Games	Games	10,000+
Nader	Tools	50,000+
Rakesh	Tools	10,000+
Careerjet Job Search	Business	1M+
All Job Search	Business	50,000+
AnyCar	Cars	1M+
Jamendo	Music	100,000+

- **True Positive (TP)** if the label matches the correct input field;
- **False Positive (FP)** if the label does not match the correct input field, and there is a textual label for this input field on the screen.
- **True Negative (TN)** if the input field was not matched to any label, and there was no textual label matching label on the screen;
- **False Negative (FN)** if the input field was not matched to any label, however, there was a valid textual label for it on the screen.

We applied the rules according to the point of view of a human, accounting for a limitation of our implementation: we do not associate input fields with images. While it is still possible for a human to extract concepts from images, our implementation works only with textual contents. Therefore, if our approach was unable to match an input field because an image instead of a text identified it, we classified this as a true negative. Our reasoning for this choice is that the algorithm did not incorrectly associate the input field with an incorrect label.

We, however, consider as false-negative, situations in which our approach could not find a textual label for an input field because it did not contain any noun, such as *Flying to* or *Where to?* While our approach disregards labels without nouns, it is intuitive for a human to associate *Flying to* to the destination of a flight.

Since the evaluation was manual and thus subject to human bias, we performed three independent evaluations for each field-label pair and selected as the final result the one with more votes.

Our findings are shown in Table 2 and summarized in Table 3. Our experiment identified 250 unique input widgets, of which 202 were matched (≈ 81%). Overall, our matching algorithm achieved

Table 2: Per app breakdown of input field matching

Name	Input Fields	Matched	Ratio	TP	TN	FP	FN	Precision	Recall	Specificity	Accuracy
Trip.com	18	15	83%	11	3	4	0	73%	100%	43%	78%
Booking.com	14	5	36%	5	8	0	1	100 %	89 %	100 %	93 %
Agoda	2	1	50%	1	1	0	0	100 %	100 %	100 %	100 %
Book Catalogue	26	24	92%	20	1	4	1	83 %	50 %	20 %	81 %
Yelp	15	6	40%	5	9	1	0	83 %	100 %	90 %	93 %
Kayak	12	8	67%	7	4	1	0	88 %	100 %	80 %	92 %
Arnab	10	9	90%	9	1	0	0	100 %	100 %	100 %	100 %
Youtube Music	4	4	100%	3	0	1	0	75 %	-	0 %	75 %
Lonely Planet Guides	10	9	90%	8	0	1	1	89 %	0 %	0 %	80 %
TripAdvisor	12	11	92%	11	0	0	1	100 %	0 %	-	92 %
Airbnb	7	5	71%	3	2	2	0	60 %	100 %	50 %	71 %
Expedia	27	24	89%	23	1	1	2	96 %	33 %	50 %	89 %
My Books - Library	7	7	100%	5	0	2	0	71 %	-	0 %	71 %
CLZ Games	15	15	100%	14	0	1	0	93 %	-	0 %	93 %
Nader	7	7	100%	7	0	0	0	100 %	-	-	100 %
Rakesh	15	8	53%	7	7	1	0	88 %	100 %	88 %	93 %
Careerjet Job Search	9	9	100%	9	0	0	0	100 %	-	-	100 %
All Job Search	14	12	86%	9	1	3	1	75 %	50 %	25 %	71 %
AnyCar	18	16	89%	13	2	3	0	81 %	100 %	40 %	83 %
Jamendo	8	7	88%	5	1	2	0	71 %	100 %	33 %	75 %
Total	**250**	**202**	**81%**	**175**	**41**	**27**	**7**				

Table 3: Unique label descriptor to input fields matching

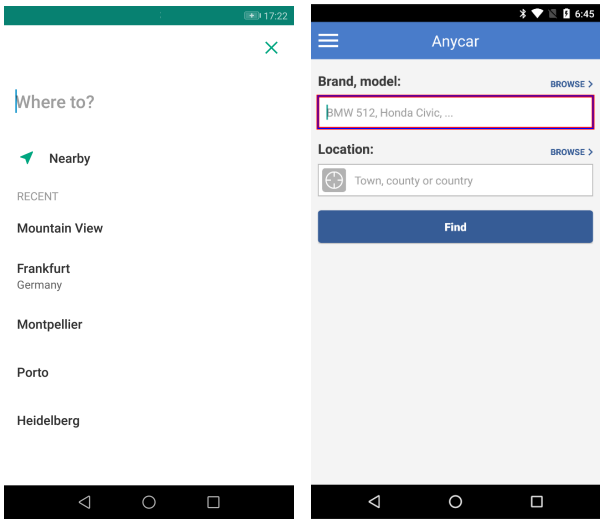
	Classified as		
Input	**True**	**False**	**Total**
True	TP = 175	FN = 7	182
False	FP = 27	TN = 41	68
Total	202	48	250

Precision = 87%
Recall = 96%
Accuracy = 86%
Specificity = 60%

87% precision and 96% recall, with 86% accuracy. These input fields represented 1.2% of the total number of unique UI elements found during the tests, which is in line with previous research [31] that demonstrated that less than 3% of the UI elements on apps are input fields.

The per-app breakdown of our results shows that most apps yielded high true-positive and true-negative values. Our manual evaluation of the false positives showed that some apps used nounless labels, such as *Where to?*, to identify the content of an input field, as shown in Figure 4a. Other false positives happened when hints were used to exemplify inputs, as shown in Figure 4b. Both false-positive examples highlight the limitations of our approach.

SAIGEN matched 81% of the input fields with 87% precision.



(a) Label without noun (b) Noun which is not a concept

Figure 4: App functionality which requires syntactically and/or semantically correct values to bypass validations

4.3 RQ2: Obtaining Input Values

The goal of our second research question is to assess the quality of querying a knowledge base for input values. With this goal, we explored the apps in our test set using DroidMate-2's default exploration strategy for 500 actions. During the exploration, we recorded the input values obtained for each input field matched from the RQ1.

We then manually assessed the syntactic and semantic quality of the inputs we obtained from the knowledge base. We considered an input as syntactically valid if it has passed all app input validation checks (no validation errors triggered).

We considered it as semantically valid if it matches the label associated with the input field. Similarly to the previous research question, we analyzed our results from a human perspective and

performed three independent analyses to mitigate the inherent bias of manual evaluations. Due to the significant manual effort involved in analyzing each input value queried from the knowledge base, we randomly chose twelve apps from our *test set* for this evaluation.

Table 4: Per app breakdown of unique labels and number of labels found on the knowledge base

Name	Unique	Found	Ratio
Trip.com	13	3	23%
Booking.com	3	2	67%
Agoda	1	1	100%
Book Catalogue	24	13	54%
Yelp	4	2	50%
Kayak	4	2	50%
Arnab	8	7	88%
Youtube Music	2	1	50%
Lonely Planet Guides	6	2	33%
TripAdvisor	9	4	44%
Airbnb	7	6	86%
Expedia	20	8	40%
Total	**101**	**51**	

Table 4 shows the number of unique labels found on each app, as well as the number of unique labels that were successfully queried. Using DBPedia, we were able to find an input for, on average, 50% of the labels found at least once during testing, with the worst app (Trip.com) finding only 23%. Note that the total number of labels is less or equal to the number of text fields found in the app (Table 2). This happens because the same label can be reused on different input fields.

Regarding the number of queries and the quality of the results, our results are shown in Table 5. We executed 360 queries for the 204 unique input fields matched (Table 2). This number is higher than the overall number of unique input fields because the same fields can be used in different queries. Considering our initial example, the concepts *Author*, *Title*, and *ISBN* are used together on the example screen, as well as alongside the label *Date* (from date published) on a different screen, resulting in two different queries. The knowledge base was able to return input values for 307 out of our 360 queries (≈ 85%), of which we classified 302 values as syntactically correct (≈ 98%) and 287 as semantically coherent (≈ 94%).

85% of Saigen's queries were able to find a result in the knowledge base. 98% of the results were syntactically valid and 94% of them were semantically valid.

4.4 RQ3: Consuming Input Values

The goal of our final research question is to measure if automatically extracted textual inputs improve Android testing. Moreover, it aims to measure the benefits of orderly interacting with the app UI. In this work, we used code coverage as an indication of the test quality as it has been shown by previous research [18] to be a good predictor. We obtained the code coverage from DroidMate-2's native code instrumentation metrics.

After instrumentation, many apps in our *test set* could no longer be used. This issue arises because many popular apps have safety verifications in place, such as certificate checks, which render the app unusable once instrumented. In the end 5 out of our 20 apps could be instrumented with DroidMate-2, namely: My Books Library, Rakesh, Kayak, Arnab, Book Catalog.

For this experiment, we explored each app five times in each of the configurations described in Table 6, to mitigate noise caused by the seed selection and by app non-determinism, resulting in 20 executions per app. In each test of each app, we executed 500 actions ($\approx$ 30*min*), as previous research [9, 10, 15] showed this to be enough to reach over 95% of the maximum test coverage. We compared the explorations by the number of actions because the overhead of querying DBPedia can be significantly reduced by hosting it locally for testing.

The results of our experiments are summarized in Figure 5. The random exploration with random textual inputs (Scenario 1) achieved an average statement coverage of 38%, with a minimum of 4% and a maximum of 74%. Scenario 2, which replaces random inputs for Saigen generated inputs but still retains DroidMate-2's random exploration strategy, achieved an average coverage of 45%, with a minimum of 7% and a maximum of 74%. These results indicate that syntactically valid and semantic coherent inputs are beneficial to the tests, with an average increase of 7%.

Using randomly generated textual inputs, alongside our sorted exploration order (Scenario 3), achieved an average coverage of 40%, with a minimum of 10% and a maximum of 74%, marginally outperforming DroidMate-2's random strategy concerning average coverage (2%) but increasing the minimum test coverage by 6%. Compared against Scenario 2, random inputs with a sorted widgets interaction achieve, on average, 5% less coverage; however, it still increases the minimum test coverage by 3%. These results indicate that orderly interacting with widgets after filling out the text fields affects the test coverage. However, it also shows that the value entered on the input field is more important than the interaction order.

Finally, in Scenario 4, when combining the sorted exploration order with Saigen, we achieve an average statement coverage of 47%, with a minimum of 11% and a maximum of 75%. These results again show the benefits of using valid input values, outperforming all our previous test scenarios. When compared to random input values and interaction order, Scenario 4 achieves, on average, 9% more coverage, with a 7% increase on the minimum coverage. When compared to Scenario 2 (Saigen inputs and random interaction order), the minimum coverage obtained is increased by 4%, and the average coverage is increased by 2%.

To ensure the statistical significance of our results, we performed a Friedman Test, which resulted in a p-value < 0.00001, meaning that the results are significant at 1%.

Filling input fields with Saigen generated values before interacting with other widgets improves code coverage by an average of 9%.

Table 5: Per app breakdown of syntactically and semantically valid inputs queried

Name	Queries	Found	Ratio	Syntactically Valid	Ratio	Semantically Valid	Ratio
Trip.com	23	13	57%	13	100%	13	100%
Booking.com	13	12	92%	12	100%	12	100%
Agoda	6	6	100%	6	100%	6	100%
Book Catalogue	44	33	75%	32	97%	32	97%
Yelp	8	6	75%	6	100%	6	100%
Kayak	19	15	79%	15	100%	15	100%
Arnab	138	137	99%	137	100%	135	99%
Youtube Music	5	4	80%	4	100%	4	100%
Lonely Planet Guides	39	35	90%	35	100%	23	66%
TripAdvisor	20	15	75%	15	100%	14	93%
Airbnb	21	19	90%	15	79%	15	79%
Expedia	24	12	50%	12	100%	12	100%
Total	**360**	**307**	**80%**	**302**	**98%**	**287**	**94%**

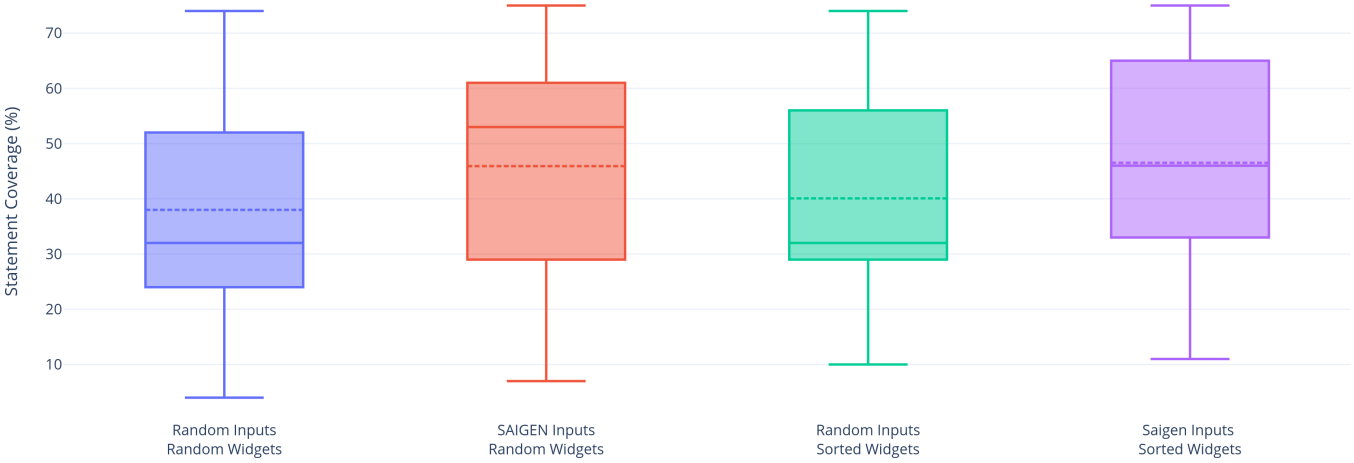


Figure 5: Coverage (%) per experimental scenario

Table 6: Coverage test scenarios

#	Textual Input generation	Widget Selection
1	Random	Random
2	Saigen	Random
3	Random	Sorted
4	Saigen	Sorted

5 LIMITATIONS AND THREATS TO VALIDITY

The presented approach and experimental evaluation present limitations and threats to validity.

Regarding external validity, our experiments have demonstrated evidence that the Saigen generated textual input values can significantly improve the test coverage with a set of benchmark apps. However, we cannot ensure that the results generalize to all apps and testing tools. We selected apps from different sizes and categories to mitigate this threat. Additionally, we added Saigen to random test generation approaches; thus, our results are limited by their constraints, such as their inherent inability to perform complex tasks. Saigen, however, can be used alongside any test generation approach to fill input fields automatically.

Regarding construct validity, in the process of extracting a concept from a label, we use a dictionary to identify synonyms for concepts that are not in the knowledge base. We observed that the synonyms acquired from the dictionary are, at times, not flexible enough for our use. Our approach is, however, abstract concerning how to obtain synonyms. Additionally, a notable limitation to our work includes the inability of Link to generate queries when multiple words are used as a query input for the knowledge base. As it stands, the approach requires a single word to be used as a query input for each concept. This can be problematic when the concepts that Saigen extract from the label descriptor contain more than one word per concept, or contains multiple concepts. This limitation causes the loss of semantics and may result in the returned values being inaccurate, as explored in [30].

Regarding internal validity, we opted for DroidMate-2's native bytecode instrumentation to acquire statement coverage, being able to test our approach with both open source and commercial apps. Our tests showed that some apps have failsafe mechanisms, such

as certificate checks, to prevent app repackaging. A more accurate coverage measurement can be obtained by using the app source code instead of its bytecode and by measuring coverage on native and JavaScript components.

6 RELATED WORK

Android test input generation is an active field of research. Testing strategies are commonly classified as *random*, *model-based*, or *explorative* strategies [13].

6.1 Random Strategies

Random strategies explore app behavior by generating random inputs. This type of strategy is used by several tools, which are mainly used to test the robustness of apps.

Monkey [16]. Google's automated random testing tool, which is a part of the Android software development kit. It is the most frequently used tool implementing random testing[3]. It generates both system-level and user events, such as clicks, touches, or gestures, using a basic random strategy. It is often used to stress-test applications and can generate reports if the app under test crashes or receives non-handled exceptions. **Dynodroid [24].** It also applies random testing. However, it attempts to explore more UI elements than Monkey by prioritizing paths that have not yet been tested. It can also generate system events. To do so, it requires instrumenting the Android framework. It checks which system events are relevant for the app under test by monitoring when the app registers listeners within the Android framework. **DroidMate [21].** It is a fully automatic GUI execution generator that runs out-of-the-box on both devices and emulators. By default, it follows Dynodroid's principle of prioritizing UI elements in which have not yet been explored, with a more accurate re-identification of UI elements. Due to its flexible architecture, we opted to use its latest version (DroidMate-2 [10]) as the base for our experiments.

6.2 Model-Based Strategies

Model-based strategies extract and use a model of the app under test to systematically generate test inputs.

DroidBot [22]. It dynamically constructs a state transition model on-the-fly and consumes it to generate test inputs. Its mains advantage is to work without instrumentation, making it useful to examine malware, because malicious apps often check their signature before triggering malicious behavior, as well as apps with safety checks against repackaging. Similar to DroidMate, users can also integrate their strategies and use them as a framework. **PUMA [19].** Features a programmable UI automation API for dynamic analysis of mobile apps. Similar to DroidBot and DroidMate-2, it can be extended with different strategies. **MobiGUITAR [3].** It is a tool that dynamically generates a model of an app during exploration, based on the run-time state of GUI widgets. The generated model can then be consumed for test generation. **SwiftHand [12].** Learns and iteratively refines, through generated inputs, a model of the app under test. It attempts to mitigate restarting an app in order to reach deeper exploration paths. It only generates touch and scroll events, no system events.

[3]https://developer.android.com/studio/test/monkey.html

Stoat [33]. Uses a combination of dynamic and static analysis to reverse-engineer a stochastic model of the app's GUI. It then consumes this model to generate events that will lead to crashes. **ORBIT [36].** It is a gray-box approach to extract an app model. It uses static analysis to extract the set of events supported by the app's GUI and then reverse-engineers an app model by systematically exercising these events. A^3E **Targeted [6].** It uses a static data flow analysis on the app byte code, to construct an activity transition graph, that captures legal transitions among activities (app screens), and then explores the graph systematically. It directs the exploration to cover all activities - especially activities that would be difficult to reach during normal use.

Our approach, without the ordering of UI elements, is built on top of a random test generation strategy. It can, however, be used alongside model-based strategies to bypass syntactic validations and improve test generation.

6.3 Explorative Strategies

The third main category of exploration strategies are *explorative strategies*.

AndroidRipper [2]. It uses a user-interface driven *ripper* to systematically traverse the app's user interface. **IntelliDroid [35].** It is a tool that attempts to trigger specific behaviors through symbolic execution. It can be configured to produce inputs specific to a dynamic analysis tool for dynamic malware analysis, and it can determine the precise order these inputs must be injected. **CuriousDroid [11]** Decomposes the app GUI on-the-fly, creating a context-based model that is tailored to the current user layout. It can be used for creating dynamic sandboxes, which is a well-known approach for detecting malicious applications. A^3E **Depth-First [6]** A depth-first variant of the *Aˆ3E* tool. It re-uses the same activity transition graphs as A^3E Targeted but explores the activities and GUI elements in a depth-first manner. It traverses the app in a slower, but more systematic way than A^3E Targeted. **Evodroid [25]** Uses a combination of program analysis and evolutionary algorithm to generate test cases that improve the overall test suite coverage.

Our approach with sorted UI element interaction is an example of an explorative approach. Saigen can, however, be used alongside other explorative test generation strategies.

7 CONCLUSION AND FUTURE WORK

We proposed an approach to obtain textual input values while testing Android apps automatically. Our results showed that even on mobile devices, it is possible to correctly identify over 95% of the labels associated with input fields; and that by entering these values on the forms from a user-perspective led to an average coverage improvement of 9%.

While our experiments were conducted in DroidMate-2, our approach is not tied to any specific test generator or strategy and can be used alongside other tools. Our approach builds upon previous research for desktop and web applications, adapting it towards Android peculiarities. There is still much room for improvements and future work:

Non-Textual UI Semantics. Our label matching approach attempts to identify a textual element to be paired to an input field. Due to size constraints, developers frequently use non-textual objects,

such as images, to describe input fields. Extracting UI semantics from non-textual elements on the GUI can allow for more accurate matching of input fields and labels.

Enhanced Concept Extraction. The concept extraction algorithm uses standard natural language processing techniques, such as part-of-speech tagging and lemmatization. It can benefit from more advanced textual concept extraction approaches that derive a context from the textual content. A brief survey on such techniques is presented in [1].

Enhanced Exploration Strategies. We explored the use of semantically aware inputs alongside a random test generation approach. The same approach can be used with model-based or explorative strategies in order to trigger more complex app functionality.

REPRODUCIBILITY

To facilitate replication and extension, all our work is available as open source. The replication package is available at:

https://drive.google.com/drive/folders/1DS7TEUIDyu8ucQ1QEFknvW8Fun35b3TO?usp=sharing

REFERENCES

[1] Mehdi Allahyari, Seyedamin Pouriyeh, Mehdi Assefi, Saied Safaei, Elizabeth D Trippe, Juan B Gutierrez, and Krys Kochut. 2017. A brief survey of text mining: Classification, clustering and extraction techniques. *arXiv preprint arXiv:1707.02919* (2017).

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. 2012. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 258–261.

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. 2015. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE software* 32, 5 (2015), 53–59.

[4] Android. 2017. UI Overview. https://developer.android.com/guide/topics/ui/overview.html

[5] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. DBpedia: A nucleus for a web of open data. In *The semantic web*. Springer, 722–735.

[6] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of Android apps. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 641–660.

[7] Giovanni Becce, Leonardo Mariani, Oliviero Riganelli, and Mauro Santoro. 2012. Extracting widget descriptions from GUIs. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 347–361.

[8] Christian Bizer, Tom Heath, and Tim Berners-Lee. 2009. Linked data—the story so far. *International journal on semantic web and information systems* 5, 3 (2009), 1–22.

[9] Nataniel P Borges Jr, Maria Gómez, and Andreas Zeller. 2018. Guiding app testing with mined interaction models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. ACM, 133–143.

[10] Nataniel P Borges Jr, Jenny Hotzkow, and Andreas Zeller. 2018. DroidMate-2: a platform for Android test generation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 916–919.

[11] Patrick Carter, Collin Mulliner, Martina Lindorfer, William Robertson, and Engin Kirda. 2016. CuriousDroid: automated user interface interaction for Android application analysis sandboxes. In *International Conference on Financial Cryptography and Data Security*. Springer, 231–249.

[12] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Acm Sigplan Notices*, Vol. 48. ACM, 623–640.

[13] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for Android: Are we there yet? (E). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 429–440.

[14] Pedro Costa, Ana CR Paiva, and Miguel Nabuco. 2014. Pattern based GUI testing for mobile applications. In *Quality of Information and Communications Technology (QUATIC), 2014 9th International Conference on the*. IEEE, 66–74.

[15] Christian Degott, Nataniel P Borges Jr, and Andreas Zeller. 2019. Learning user interface element interactions. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 296–306.

[16] Android Developers. 2012. UI/application exerciser monkey.

[17] Alan Dix. 2009. Human-computer interaction. In *Encyclopedia of database systems*. Springer, 1327–1331.

[18] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 72–82.

[19] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. 2014. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 204–217.

[20] Roberto Jacinto. 2010-2018. What is F-Droid? https://f-droid.org/en/about/. Accessed: 2018-07-28.

[21] Konrad Jamrozik and Andreas Zeller. 2016. Droidmate: A robust and extensible test generator for Android. In *Mobile Software Engineering and Systems (MOBILESoft), 2016 IEEE/ACM International Conference on*. IEEE, 293–294.

[22] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight UI-guided test input generator for Android. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*. IEEE, 23–26.

[23] Linda L Lohr. 2000. Three principles of perception for instructional interface design. *Educational Technology* 40, 1 (2000), 45–52.

[24] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 224–234.

[25] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 599–609.

[26] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. 55–60.

[27] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. 2014. Link: exploiting the web of data to generate test inputs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 373–384.

[28] Lluís Màrquez and Horacio Rodríguez. 1998. Part-of-speech tagging using decision trees. In *European Conference on Machine Learning*. Springer, 25–36.

[29] Larry Masinter, Tim Berners-Lee, and Roy T Fielding. 2005. Uniform resource identifier (URI): Generic syntax. (2005).

[30] Seyed Iman Mirrezaei, Bruno Martins, and Isabel F Cruz. 2015. The triplex approach for recognizing semantic relations from noun phrases, appositions, and adjectives. In *International Semantic Web Conference*. Springer, 230–243.

[31] Alireza Sahami Shirazi, Niels Henze, Albrecht Schmidt, Robin Goldberg, Benjamin Schmidt, and Hansjörg Schmauder. 2013. Insights into layout patterns of mobile user interfaces by an automatic analysis of Android apps. In *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, 275–284.

[32] Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. 2014. Adoption of the linked data best practices in different topical domains. In *International Semantic Web Conference*. Springer, 245–260.

[33] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 245–256.

[34] Kristina Toutanova and Colin Cherry. 2009. A global model for joint lemmatization and part-of-speech prediction. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1-Volume 1*. Association for Computational Linguistics, 486–494.

[35] Michelle Y Wong and David Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware.. In *NDSS*, Vol. 16. 21–24.

[36] Wei Yang, Mukul R Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 250–265.

[37] Amrapali Zaveri, Anisa Rula, Andrea Maurino, Ricardo Pietrobon, Jens Lehmann, and Sören Auer. 2016. Quality assessment for linked data: A survey. *Semantic Web* 7, 1 (2016), 63–93.