Extracting information types from Android layout code using sequence to sequence learning

Mitra Bokaei Hosseini, Xue Qin, Xiaoyin Wang, and Jianwei Niu
University of Texas at San Antonio, San Antonio, TX, USA

Abstract

Android mobile applications collect information in various ways to provide users with functionalities and services. An Android app's permission manifest and privacy policy are documents that provide users with guidelines about what information type is being collected. However, the information types mentioned in these files are often abstract and does not include the fine grained information types being collected through user input fields in applications. Existing approaches focus on API calls in the application code and are able to reveal what information types are being collected. However, they are unable to identify the information types based on direct user input as a major source of private information. In this paper, we propose to direct apply natural language processing approach to Android layout code to identify information types associated with input fields in applications.

Introduction

Mobile apps are being used widely in domains where a lot of privacy information is involved. According to a latest report in May 2017, 58.23% of mobile phone users had downloaded a health-related mobile app by 2015 (Krebs and Duncan 2015), which can collect information on body measurements, diet, exercise, and medical treatment, among others. Similarly, Mint, one of the most popular personal finance apps, serves more than 20 million users and 73% of them pay their balances every month through the Mint. (min). To protect privacy, mobile app users should better understand how their personal information is collected, used and shared by their apps.

With increased access to personal information and the scale of mobile app deployment, the need for tools to help developers to protect user privacy is increasingly important. Google encourages app developers to provide users with privacy policies that describe how personal information is collected from users (Slavin et al. 2016). These policies are written in natural language and describe the data practices. Such policies are also meant to fulfill legal requirements to protect privacy, such as the GDPR in Europe, or FTC Act in the US. However, innovation and competition among mobile app developers challenges identifying the trace links between privacy polices and app code.

Therefore, there is a need for automatic extraction of information types being collected through application code that can be used for checking the consistency between the code and the data practices in privacy policies.

Prior work by Slavin et al. (Slavin et al. 2016) and Zimmeck et al. (Zimmeck et al. 2017) attempt to identify platform information types collected through API calls with static analysis. These API calls concern personal data that is automatically collected from the device, such as sensor data. However, these works are not focused on addressing personal data that *users provide directly through an app's user interface.* Figure **??** shows an example where sensitive data is provided to the app via the interface and is thus disconnected from any API call. These user-based inputs are difficult to identify as they are both context-sensitive and can vary in implementation from developer to developer, so that they bring two new technical challenges as follows:

C1: Vague and Unbounded Information Types for User Input Data. The information types automatically collected through platform API methods are constrained to Android API which is described by comprehensive documents and information collected is well defined. These constraints limit the terminological space to only a few general category names (e.g., location, voice, etc.) In contrast, developers can design novel user interfaces that ask users to provide potentially *any kind of information*, which includes unstructured and semi-structured personal information in different formats and language types.

C2: Varying User Interface Structures. Unlike platform API method calls that can be detected by scanning the app byte code, user interfaces are implemented as Android layouts using static declarations in XML code. The XML code can be very complicated, with various view types, attributes, and nested structures.

To address both challenges we propose an approach that identifies the information types associated with user input fields automatically. Our approach is based on the assumption on the naturalness of Android layout code, so that it is possible to directly apply natural language processing technique to the layout code, and extract the information types, just as extracting semantics from natural language texts. Specifically, given a decompiled Android app, first we extract the static layout files and we construct a context sequence for EditText view elements by analyzing the preced-

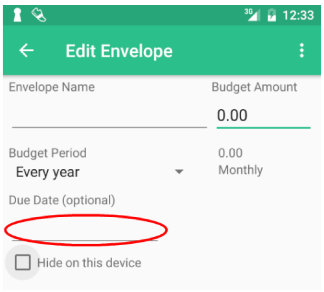

Figure 1: User Interface Screenshots from Good Budget

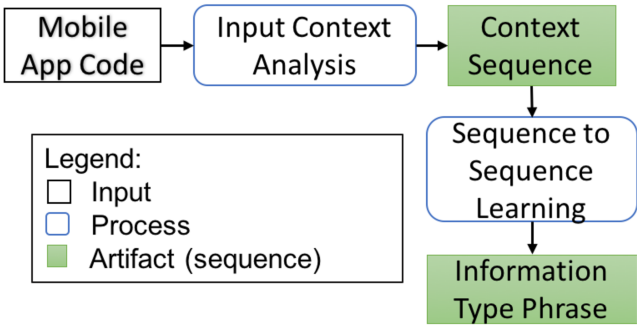

Figure 2: Identifying information type phrases from UI context analysis

ing views in the graphic user interface (GUI). Second, we establish a ground truth by asking human subjects to identify information types related input fields in GUI. The context sequence generated for each EditText element is then paired with human interpretations of the user input field. This data is used to train a sequence-to-sequence Long Short-Term Memory (LSTM) model. Finally, given a context sequence from user interface static layout, our trained sequence-to-sequence model is used to identify the information type.

This paper is organized as follows: First, we provide a motivating example on user provided information through input fields; second, we discuss our proposed approach for automatic extraction of information types from static layouts; and finally, we provide our proposed experiment setup.

Motivating Example

In this section, we give a real example showing how GUI views, especially user input views can be constructed from layout file. In the Android framework, a *layout* defines the visual structure of the GUI, such as locations for views, buttons, windows, and widgets.

```
1  <LinearLayout android:id=
2    "@id/trans_message_header">
3   <TextView android:id=
4     "@id/trans_message_text"/>
5   ...
6   <TextView android:id="@id/name_label"
7     android:text=
8     "@string/edit_envelope_envelope_name"/>
9   <TextView android:id="@id/amount_label"
10    android:text="@string/edit_envelope_budget"/>
11  ...
12  <EEBAAutoCompleteTextView
13    android:id="@id/name"/>
14  <EditText android:id="@id/amount"
15    android:hint="@string/amount_hint"/>
16  ...
17  <TextView android:id="@id/period_label"
18    android:text="@string/envelope_period_label"/>
19  <TextView android:id="@id/helper_text_amount"
20    android:text="0.00" />
21  ...
22  <Spinner android:id="@id/period"
23    android:prompt="@string/period_prompt"/>
24  <TextView android:id="@id/helper_text_period"
25    android:text="@string/period_text_monthly"/>
26 </LinearLayout>
```

```
27  <LinearLayout android:id="@id/extra_fields">
28   <TextView android:id="@id/due_date_label"
29      android:text="@string/due_date_label"/>
30   <LinearLayout>
31    <EditText android:id="@id/due_date" android:text=""/>
32   ...
33   <CheckBox android:id="@id/local_hidden"/>
34   <TextView android:text="@string/edit_envelope_hide"/>
35   ...
```

Listing 1: Partial Code from edit_envelope.xml

Layouts Layouts allow developers to pre-draw the GUIs and reduce the overhead at runtime which can be extracted by decompiling the app's APK files. Static layout files contain the structure of pre-drawn GUIs, view ids as well as all the text labels we can see from the GUIs. Listing 1 shows the partial code of `edit_ envelope.xml`, which is the static layout file of Figure 1.

In listing 1, lines 6-8 refer to a `TextView` element that corresponds to "Envelope Name" field label in Figure 1. This element also has two attributes: `android:id` for identfication purposes; and `android:text` which contains a reference to `string.xml` file including the actual text user observes on the GUIs. Lines 28-29 refer to a `TextView` View corresponding to "Due Date" field label in Figure 1. This View also has two attributes: `android:id` for identification purposes; and `android:text` which contains a reference to `string.xml` file including the actual text user observes on the GUIs.

GUI Context. Just like natural language text, input views can only be well understood with neighboring/ancestor views. For the circled input field in figure 1 which relates to line 31 in listing 1, without considering the context "envelope" only "due date" can be inferred as the information type. If the privacy policy contains the collection of "bill information" or "envelope information", the automatic consistency checkers fail to trace "due date" to "envelope information" without further context information. Therefore, GUI context is essential in understanding user input information types. In this paper, we propose a learning model on GUI context to infer the proper information type for user input fields.

Proposed Approach

We present the overview of our approach in figure 2 which consists of two main steps: (1) given a mobile app decompiled code, the graphical user interface (GUI) analysis extracts the layout XML code and constructs a *context sequence* for each input field (`TextView`) which includes the id, text, and hint attributes of the `TextView`, and the id, text, and hint attributes of all views preceding the `TextView` in the layout XML file; (2) The sequence-to-sequence learning component takes a sequence that represents an input field context and map it to a target sequence of words that represents an information type phrase. The results from these two steps are shown as artifacts in figure 2. We next present the details for each step in the following two sub-sections.

Input Context Analysis

In the GUI context analysis phase, we first decompile the app's APK files and extract all XML layout files[1] associated with pre-drawn GUIs in the app. A layout XML file declares the ViewGroups in the GUI. A View may [2] have multiple attributes such as id and text. In our study, we only focus on id, text, and hint attributes, since they are typically related to the semantics of a view. Android provides seven types of input controls [3] to help interact with app GUIs, including button, checkbox, text fields, etc. In this research, we only focus on *EditText* for user input analysis to identify the related information types. To construct the context sequence for a target EditText View, we analyze the XML file and gradually add IDs, text, and hints related to all the Views preceding the target EditText View resulting in context sequence. Moreover, strings will be transformed to their corresponding English phrases in textttstring.xml when adding to context sequence. The following example shows the context sequence extracted from listing 1 the EditText View with ID `due_date` which should be mapped to "envelope due date" through the learning process:

Context Sequence: {trans message header, trans message text, name label, Envelope Name, amount label, Budget Amount, name, amount, 0.00, period label, Budget Period, helper text amount, period, Select a Budget Period, helper text period, Monthly, extra fields, due date label, Due Date (optional), due date}

Next we will introduce how to infer information types using the extracted sequences using sequence to sequence modeling.

Sequence-to-Sequence Modeling

Recurrent Neural Networks (RNNs) (Rumelhart et al. 1988) and specifically Long Short-Term Memory (LSTM) models (Hochreiter and Schmidhuber 1997) are natural generalization of feedforward neural networks used for processing long sequential data such as sentences (Rumelhart et al. 1988; Werbos 1990). RNNs connect computational units of the network in a directed cycle such that at each time step *i*, a unit in the RNN takes both the input of the current step (i.e., the $word_i$ in the sequence), and the hidden state of the same unit from the previous time step *i-1* (Guo, Cheng, and Cleland-Huang 2017). However, a standard RNN model can map a source sequence to a target sequence whenever the dimensionality of the source and target is known ahead of time (Sutskever, Vinyals, and Le 2014). To solve this problem, Cho et al. proposed a model that uses two RNNs as encoder and decoder which maps the source sequence to a fixed size vector which is then mapped to a target sequence (Cho et al. 2014). However, RNNs are known for losing long term dependencies between words in the source sequence (Bengio, Simard, and Frasconi 1994) and therefore, LSTM networks were introduced to preserve long term dependencies through a memory cell vector in the recurrent unit (Hochreiter and Schmidhuber 1997).

Information type phrases are comprised of sequence of words with various lengths which are not known at the time. To identify the information types from the GUI context sequences, we plan to use two LSTMs (Sutskever, Vinyals, and Le 2014) which maps a source to a target sequence. First, we encode the input sequence to a vector of fixed dimension that includes the semantics of the input sequence using a multi-layered LSTM. Next, we feed the input vector to another LSTM which decodes the target sequence from the vector.

Figure 3 depicts all the elements of the sequence to sequence learning model using two LSTMs. We now describe each part of the model in details. In the first step, we present the source sequence as a vector of word tokens $(x_1, x_2, ..., x_s)$, where x_i corresponds to the *ith* word in the source sequence. Next, each word is mapped to its vector representation through *Word Embedding* layer (Mikolov et al. 2013). We plan to learn the embedding vectors and build the vocabulary from Wikipedia text as general corpus for this work. The goal of our model is to estimate the conditional probability $p(y_1, ..., y_t|x_1, ..., x_s)$, where $(y_1, ..., y_t)$ is the target sequence with length t which differs from the source sequence length s. For this reason, the embedded source vectors are sequentially fed into the LSTM units which result in a single vector X representing the semantics of the source sequence. Next, the model computes the probability of $(y_1, ..., y_t)$ with another LSTM network whose initial hidden state is set to X which represents the source sequence semantics:

$$p(y_1, ..., y_t|x_1, ..., x_s) = \prod_{i=1}^{t} p(y_i|X, y_1, ..., y_{i-1})$$

In this equation, $p(y_i|X, y_1, ..., y_{i-1})$ predicts each word in the target sequence using the previous predicted words and the source sequence semantics. This prediction is modeled using a softmax classifier that assigns a probability to all the words in the vocabulary and selects the word with the highest probability as y_i. It is also necessary that both source and target sequences end with a special vector representation <EOS>, which enables the model to define a distribution over sequences of various length (Sutskever, Vinyals, and Le 2014). In the final stage, the predicted $(y_1, ..., y_t)$ is transformed to related word tokens $(w_1, ..., w_t)$ using the *Word Embedding* layer.

[1] https://developer.android.com/guide/topics/ui/declaring-layout.html

[2] https://developer.android.com/guide/topics/ui/overview.html#Layout

[3] https://developer.android.com/guide/topics/ui/controls.html

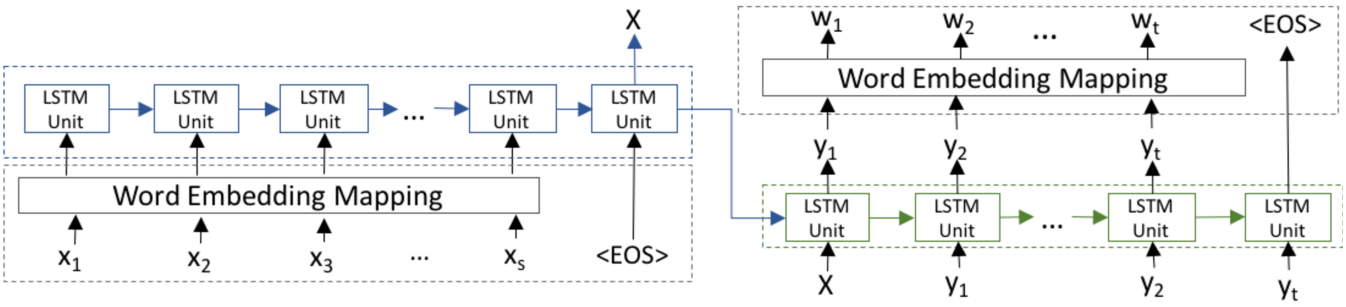

Figure 3: Mapping source sequence to target sequence using two LSTM networks

Proposed Experiment Setup

We have constructed an initial dataset and we plan to expand it and base our experiment on the expanded set. To elicit input field types from crowd workers, we designed a free listing survey (Bernard 2011), in which workers were asked to identify the information type that describes the information entered into the app through a specific UI input field, shown in a red circle in the screenshot (see Figure 1). Each survey consists of 3-5 screenshots, and we surveyed 53 input fields from 19 apps. We recruited 30 participants per survey using Amazon Mechanical Turk to yield 393 HITs. Participants of the surveys were located in the United States with an overall HIT approval rating greater than 95%.

We obtained 30 information types per input field. Because there are multiple ways to describe the same concept, we pre-processed the results to more easily compare elicited types as follows: The pre-processing steps are similar to porter stemming in natural language processing, were verb conjugation is removed to make verb comparisons easier (Porter 1980; 2001). After pre-processing, we combine similar type names for each field and calculate the type name frequency, which is the number of workers who provided each syntactically unique type name per field. Finally, for each field, we select the most frequent type name, which remains linked to a set containing the less frequent type names for that field.

We understand that 53 input fields is not a sufficient number for training the sequence to sequence model. However, we believe we can extend this study to acquire a sufficient number of training samples.

We also analyzed the 53 input fields and infer input types by concatenating the file name and input field labels. The results was compared with the most frequent input types provided by crowd workers showing 33.9% match. This suggest that a naive approach with local context is not effective.

Acknowledgment

The authors are supported in part by NSF Awards CNS-1330596, CCF-1464425, CNS-1748109, NSA Grant on Science of Security, and DHS grant DHS-14-ST-062-001.

References

Bengio, Y.; Simard, P.; and Frasconi, P. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks* 5(2):157–166.

Bernard, H. R. 2011. *Research methods in anthropology: Qualitative and quantitative approaches*. Rowman Altamira.

Cho, K.; Van Merriënboer, B.; Gulcehre, C.; Bahdanau, D.; Bougares, F.; Schwenk, H.; and Bengio, Y. 2014. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Guo, J.; Cheng, J.; and Cleland-Huang, J. 2017. Semantically enhanced software traceability using deep learning techniques. In *Proceedings of the 39th International Conference on Software Engineering*, 3–14. IEEE Press.

Hochreiter, S., and Schmidhuber, J. 1997. Long short-term memory. *Neural computation* 9(8):1735–1780.

Krebs, P., and Duncan, T. D. 2015. Health app use among us mobile phone owners: A national survey. *JMIR mHealth uHealth* 3(4):e101.

Mikolov, T.; Chen, K.; Corrado, G.; and Dean, J. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mint by the numbers: Which user are you?, https://blog.mint.com/credit/mint-by-the-numbers-which-user-are-you-040616/. Accessed: 2017-08-23.

Porter, M. F. 1980. An algorithm for suffix stripping. *Program* 14(3):130–137.

Porter, M. F. 2001. Snowball: A language for stemming algorithms.

Rumelhart, D. E.; Hinton, G. E.; Williams, R. J.; et al. 1988. Learning representations by back-propagating errors. *Cognitive modeling* 5(3):1.

Slavin, R.; Wang, X.; Hosseini, M. B.; Hester, J.; Krishnan, R.; Bhatia, J.; Breaux, T. D.; and Niu, J. 2016. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, 25–36. New York, NY, USA: ACM.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, 3104–3112.

Werbos, P. J. 1990. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE* 78(10):1550–1560.

Zimmeck, S.; Wang, Z.; Zou, L.; Iyengar, R.; Liu, B.; Schaub, F.; Wilson, S.; Sadeh, N.; Bellovin, S. M.; and Reidenberg, J. 2017. Automated analysis of privacy requirements for mobile apps. In *Network and Distributed System Security Symposium NDSS*.