UiRef: Analysis of Sensitive User Inputs in Android Applications

Benjamin Andow
North Carolina State University
beandow@ncsu.edu

Akhil Acharya
North Carolina State University
acachar2@ncsu.edu

Dengfeng Li
University of Illinois at Urbana-Champaign
dli46@illinois.edu

William Enck
North Carolina State University
whenck@ncsu.edu

Kapil Singh
IBM T.J. Watson Research Center
kapil@us.ibm.com

Tao Xie
University of Illinois at Urbana-Champaign
taoxie@illinois.edu

ABSTRACT

Mobile applications frequently request sensitive data. While prior work has focused on analyzing sensitive-data uses originating from well-defined API calls in the system, the security and privacy implications of inputs requested via application user interfaces have been widely unexplored. In this paper, our goal is to understand the broad implications of such requests in terms of the type of sensitive data being requested by applications.

To this end, we propose UiRef (User Input REsolution Framework), an automated approach for resolving the semantics of user inputs requested by mobile applications. UiRef's design includes a number of novel techniques for extracting and resolving user interface labels and addressing ambiguity in semantics, resulting in significant improvements over prior work. We apply UiRef to 50,162 Android applications from Google Play and use outlier analysis to triage applications with questionable input requests. We identify concerning developer practices, including insecure exposure of account passwords and non-consensual input disclosures to third parties. These findings demonstrate the importance of user-input semantics when protecting end users.

1 INTRODUCTION

Mobile applications continue to consume an increasing amount of sensitive data to perform a variety of tasks, such as personal banking, health tracking, and online shopping. Vetting these applications to identify abnormal behaviors is paramount to ensure that privacy and security requirements of users are adequately satisfied.

Prior research [6, 10, 11, 13, 14, 32] has demonstrated the utility of analyzing sensitive-data requests to identify security and privacy problems within applications, including insecure-data transmission, private-data leakage, and malware identification. Such approaches often rely on the semantics of the user data to enable automated analyses, e.g., for taint sources and runtime privacy notices. However, these approaches focus on *only* sensitive data originating from well-defined, system API calls, which explicitly represent the semantics of the data returned by the API invocation. Such focus addresses only a part of the challenge: applications also obtain a wide range of sensitive data, such as passwords and credit-card numbers, as input within their graphical user interfaces (GUIs). These data requests are largely ignored by prior analyses.

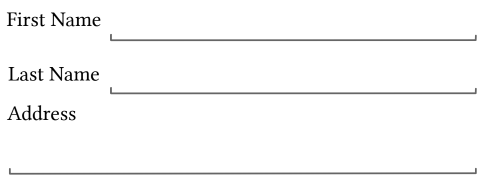
In this work, we focus on protecting sensitive data collected by third-party applications via their GUIs. Resolving the semantics of such data is challenging, as the API calls that retrieve a user input are generic and do not indicate the data's semantics. For example, a user's home address and credit-card number are retrieved by the same `getText()` API call if entered into EditText widgets. Instead, the semantics of a user input is denoted by natural-language text in the GUI, which prompt the user to enter specific types of data.

Semantics resolution of user inputs has been explored in two recent techniques: SUPOR [18] and UIPicker [22]. However, as we discuss in Section 3, both techniques have significant limitations that lead to inaccuracies. Further, both techniques fail to take into account the problem of *word ambiguity*, which is the coexistence of multiple meanings for a word or phrase. For example, the word "address" can refer to a postal address or an IP address depending on the context that the word is used, and hence can cause confusion for the existing techniques. In Section 5.3, we measure the prevalence of ambiguity during semantics resolution and show that 20.9% of input resolutions contain an ambiguous term. Although word ambiguity is a known and actively researched area in NLP, the limited amount of text within GUIs makes disambiguation especially challenging.

In this paper, we propose UiRef, a User Input REsolution Framework that automatically resolves the semantics of user-input widgets by analyzing the GUIs of Android applications. UiRef advances the state of the art using novel techniques in its three main stages: layout extraction, label resolution, and semantics resolution. First, UiRef's layout extraction provides a new hybrid analysis that accurately renders custom widgets. Prior techniques do not fully support custom widgets, which are used by 48.7% of the applications under our study. Second, UiRef's label resolution models patterns within the spatial arrangement of widgets, achieving a 20.8% improvement over prior work [18]. Finally, UiRef's semantics resolution disambiguates words within input labels. Existing word-disambiguation techniques [20] cannot be applied directly, as GUIs rarely display text in full sentences. Instead, UiRef learns different word senses based on text that appears in other widgets within layouts.

We use UiRef to perform a large-scale analysis of 50,162 applications from Google Play. UiRef resolves the user-input semantics

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WiSec '17, Boston, MA, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
978-1-4503-5084-6/17/07…$15.00
DOI: 10.1145/3098243.3098247



```
<LinearLayout width="match_parent" height="wrap_content" orientation="vertical">
    <LinearLayout width="match_parent" height="wrap_content">
        <TextView width="wrap_content" height="wrap_content" text="First Name"/>
        <EditText width="match_parent" height="wrap_content" id="@+id/first"/>
    </LinearLayout>
    <LinearLayout width="match_parent" height="wrap_content">
        <TextView width="wrap_content" height="wrap_content" text="Last Name"/>
        <EditText width="match_parent" height="wrap_content" id="@+id/last"/>
    </LinearLayout>
    <TextView width="wrap_content" height="wrap_content" text="Address"/>
    <EditText width="match_parent" height="wrap_content" id="@+id/address"/>
</LinearLayout>
```

Figure 1: Example Android layout and corresponding XML specification with stripped namespaces

for each application. We then use outlier detection to identify questionable, and potentially malicious, input requests. The goal of our analysis is to provide an understanding of what sensitive information applications are asking for and broadly identify various questionable practices in collecting user inputs that can potentially compromise users' security and privacy. Note that for this work, we do not differentiate a questionable practice from malicious intent. Such differentiation can be done by further analysis using existing techniques [6, 10, 11, 13, 14, 32]. Rather, UiRef can filter the application dataset for focused deeper inspection.

Our analysis leads to three main security and privacy findings (see Section 6). First, we find that applications request a wide range of sensitive data, including SSNs, passport numbers, and healthcare information. While legitimate requests for this data may exist, we find that many such requests do not align with the purpose of the requesting application. Second, we identify 66 applications that directly request the user's third-party account passwords (e.g., Gmail). These requests expose the user's third-party accounts to compromise, as well as violating the primary tenet behind OAuth-like solutions. Finally, we identify 6 popular applications that send sensitive-input data to advertisers without disclosing the behavior.

During our analysis, we also identify application design flaws that may result in loss of privacy. First, applications allow third-party libraries to directly request sensitive inputs from users, likely resulting in user confusion. In other words, the user cannot identify whether the library or the application is requesting data. This flaw can be exploited by malicious libraries to compromise sensitive data. Second, applications display untrusted third-party components in the same GUIs as sensitive-input requests, leaving the applications vulnerable to private-data theft due to layout-traversal attacks [29].

In summary, this paper makes the following main contributions:

- We develop novel techniques for semantics analysis of GUI inputs to considerably improve over the current state of the art (5.5%-25.6% more accurate at extracting layouts and 20.8% more accurate at resolving labels). We perform a direct comparison with prior work by implementing a representative solution in SUPOR [18].
- We propose a novel approach to address ambiguity of descriptive text in mobile applications. Ambiguity is a key challenge unaddressed by prior work, and it is particularly challenging for GUIs due to limited text. We demonstrate that word embeddings can effectively perform this task.
- We demonstrate the utility of UiRef in performing security and privacy analysis through a large-scale study on 50,162 Android applications. Our study highlights various forms of questionable, and potentially malicious, practices by developers and emphasizes the need to consider user inputs in the security and privacy analysis of mobile apps.

The remainder of this paper describes the design and implementation of UiRef, followed by its application towards security and privacy threats in user inputs.

2 BACKGROUND

When using Android applications, users interact with layouts (i.e., GUIs) to enter inputs and perform actions. Layouts consist of widgets, such as push buttons, text fields, and check boxes. These widgets are arranged and grouped within layouts through the use of *view groups*, which are containers for holding other *views* (i.e., view groups or widgets). Due to view groups, layouts are structured as hierarchical trees named as *view hierarchies*.

Layouts are typically defined by the developer at compile time using XML. Figure 1 shows a simplified version of a layout's XML file (i.e., *main_layout.xml*) and a corresponding screenshot. To render a layout, applications pass the resource identifier of the layout's XML file to the setContentView() or LayoutInflater.inflate() methods. To access a resource, developers use the Java R class generated during compilation, while the compiled application uses integer constants to internally refer to resources. For example, to render the layout in Figure 1, an app invokes setContentView(R.id.main_layout). The public.xml file in the application package (APK) provides a mapping of the integer constants to the strings used in the XML layout.

Android's SDK provides a wide range of pre-defined view groups and widgets. However, developers may also define custom views in their Java code by extending these pre-defined classes, which can then be referenced in the layout's XML files. The implementation of these custom views may customize rendering and also dynamically insert view groups and widgets within its view hierarchy.

3 PROBLEM AND CHALLENGES

Unlike information retrieved from Android platform APIs (e.g., GPS), the semantics of text inputs is often poorly defined. As such, it received limited investigation by prior research [18, 22] in contrast to the wealth of literature studying the abuse of privacy-sensitive platform APIs [11, 13, 14, 32].

Problem Statement: *This work seeks to understand what information Android apps are requesting through their GUIs by automatically resolving the semantics of the user inputs and to further analyze their potential security and privacy repercussions.*

Semantics resolution is the first step in ensuring proper handling of security and privacy sensitive input. There are many ways in which the resulting meta information can be used to enhance security analysis (e.g., semantics for taint sources). In this paper, we use the resolved semantics to achieve two main goals. First, we use it to understand the general landscape of the types of security and privacy sensitive information that applications are requesting from the user. Second, we leverage it to identify questionable, and

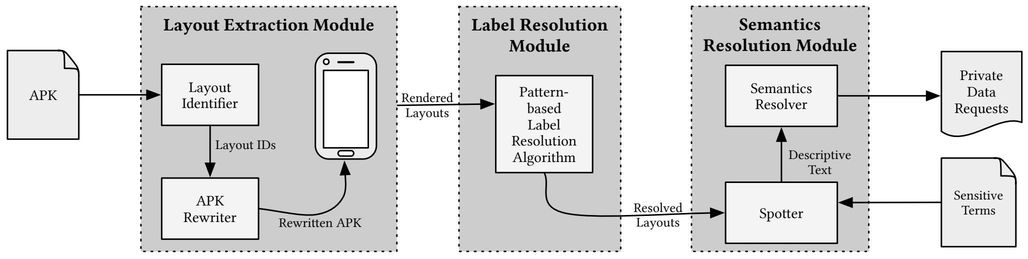


Figure 2: UiRef Architecture

potentially malicious, practices used by apps by performing outlier analysis. This analysis identifies applications that request data uncommon for the its category (e.g., a game asking for an SSN) and allows us to triage apps for manual analysis.

Our work is not the first to consider the semantics of user inputs in Android applications. Prior techniques (SUPOR [18], UIPicker [22], and AppsPlayground [28]) have significant limitations that impact the accurate and unambiguous resolution of the semantics of user inputs. We identify three areas of challenges: layout extraction, descriptive-text (label) resolution, and semantics resolution.

Layout Extraction: The spatial arrangement of widgets influences the layout's semantics. While the parent-child and sibling relationships of a layout's view hierarchy influence the spatial arrangement, it is a poor approximation of proximity due to the flexibility allowed by widgets. Further, code defining custom views may control the types of widgets displayed as well as their position in the layout. Around 48.7% of apps in our study (Section 6) use custom views.

UIPicker operates directly on the XML specification of layouts; therefore, it cannot accurately interpret the spatial arrangement of widgets and their proximities to one another. In contrast, SUPOR extracts layouts by modifying the static rendering engine of the Android Developer Tool (ADT), which provides this positioning. However, neither UIPicker or SUPOR sufficiently handles custom views. The UIPicker paper does not mention custom views. The static rendering engine of ADT used by SUPOR cannot execute the bytecode of custom views. Therefore, SUPOR renders custom views as the nearest superclass available in the Android framework, causing inaccurate and incomplete layout rendering. Finally, AppsPlayground can handle custom views, but it must dynamically navigate the GUI to reach layouts.

Descriptive-Text Resolution: To prompt the user for inputs, developers either use text widgets with a label in the nearby proximity to the input fields (Figure 1), or embed descriptive text as attributes on the input widgets (e.g., the hint and text attributes on the EditText widget). While embedded descriptive text is straightforward to resolve, identifying the correct label for an input widget is nontrivial. Even if embedded descriptive text exists for an input widget, the corresponding label widget must be resolved. Some applications use labels to denote the type of information and embedded descriptive text to provide instructions to the user (e.g., "required").

A general approach to match an input widget with a label widget is to define a distance metric. Both UIPicker and AppsPlayground use sibling relationships in layouts to resolve the associated descriptive text. However, in practice, sibling relationships do not accurately gauge proximity. In contrast, SUPOR performs label resolution by partitioning the space surrounding an input widget into nine areas, calculating a position-based weighted average for each pixel in the input widget to the nearest pixels in labels, and mapping each input widget to the label widget with the lowest weighted average. However, SUPOR suffers from multiple limitations that are sources for incorrect label resolutions as shown in Section 5. First, it always resolves labels to an input widget even if the label is *too far away* from the input widget. Second, it depends on the input widget's area size and predefined weights that bias it towards labels to the left and above, thus affecting the algorithm's generality.

Semantics Resolution: Mobile applications use short text phrases in descriptive text, making semantics resolution of user inputs challenging. Word polysemy limits the use of simple key-phrase matching techniques, and affects 20.9% of input-field resolutions (Section 5.3). Therefore, disambiguation is required.

None of the prior techniques resolve the ambiguity of words. SUPOR, UIPicker, and AppsPlayground use key-phrase matching on the input widget's associated descriptive text, but cannot differentiate the multiple semantics of that key phrase. SUPOR acknowledges this limitation, and excludes the word "address" from their key-phrase list. Additionally, UIPicker leverages developer-defined variable names and input-widget type attributes (e.g., password) in their resolution. However, such developer-defined names are not tolerant to name-based obfuscation or poor coding practices.

4 SYSTEM APPROACH AND DESIGN

The UiRef resolves the data semantics of user-input widgets by analyzing Android layouts. Figure 2 shows UiRef's architecture. The layout-extraction module instruments APKs to force the rendering of their layouts and exports the rendered layouts for further analysis. The label-resolution module identifies patterns within the placement and orientation of labels with respect to input widgets in the layout and geometrically resolves labels. The semantics-resolution module applies text-analytics techniques on the input widget's associated descriptive text to determine the expected type of information accepted by the input widgets.

4.1 Layout Extraction

The goals of the layout-extraction module are three-fold: (1) identify all layouts in an application; (2) identify the spatial relationships between UI widgets for each layout; and (3) identify the descriptive text displayed to users. The spatial relationships are needed to map label widgets to input widgets, as discussed in Section 4.2.

UiRef's layout-extraction module uses a hybrid technique to extract rendered layouts. UiRef's technique uses static analysis to identify the layouts used by the application, and dynamic on-device rendering to extract each rendered layout that users eventually interact with. Since on-device rendering allows for the execution of developer code, it allows for the correct rendering of custom views.

UiRef disassembles an APK using ApkTool [4], and collects the resource identifiers for each layout present in the APK's public.xml file. It saves the layout resource identifiers to the application's assets folder to be used later during on-device rendering. Subsequently, it injects a custom activity into the APK, and rewrites the application's manifest file to register the injected activity as an entry point. UiRef then reassembles the APK, and sideloads it onto a live device.

When the injected activity is executed, it iterates through the resource identifiers saved in the assets folder, and invokes the setContentView() method for each identifier to cause the application to render the layout. It then iterates through the rendered layout's view hierarchy to extract associated metadata, such as the coordinates of each view, visibility attributes, and text strings. This information along with the view hierarchy is exported as XML.

As discussed in Section 4.2, UiRef's label-resolution module requires a list of potential labels and a list of input widgets. To construct these lists, UiRef uses class-inheritance information to identify the types of widgets that commonly accept user inputs (i.e., EditText, AbsSpinner, CheckedTextView, RadioButton, CheckBox, ToggleButton, Switch, SwitchCompat, RatingBar), and widgets that display static text (i.e., TextView). For example, an EditText, or widgets that extend EditText, are likely to accept user inputs. Note that this technique differs from SUPOR's use of inheritance information, as SUPOR uses it during the rendering process to determine how to render custom views. In contrast, UiRef does not rely on inheritance information during rendering, as UiRef can render custom views. UiRef extracts inheritance information during layout extraction to avoid performing the Class Hierarchy Analysis offline.

To determine whether a widget accepts a user input, UiRef resolves the closest ancestor in a widget object's class hierarchy that is in Android's framework, and includes the nearest SDK class in the metadata output. For example, if a class CustomTextView1 extends android.view.Textview, UiRef would mark its ancestor as TextView. Similarly, if CustomTextView2 extends CustomTextView1, UiRef would also mark its ancestor as TextView, because CustomTextView2 extends CustomTextView1, which extends TextView. We identify 14 base widget types from Android's documentation, and use Java's instanceof operator to find the view object's nearest SDK class.

Note that UiRef executes only the code needed to render layouts, and hence does not suffer from code-coverage limitations. This technique does have two limitations. First, UiRef is limited to statically defined layouts. However, statically defining layouts is a common practice, as it typically requires less effort than dynamic generation. Second, UiRef cannot extract dynamically generated text (e.g., from network connections). However, since we focus on text labels for user inputs and not display content, we believe it is reasonable to assume that labels statically define the text.

4.2 Label Resolution

The goal of UiRef's label-resolution module is to identify the label associated with each user-input widget. It operates on the intuition that developers are consistent with the physical arrangement and orientation of labels to user-input widgets. For example, if a developer positions labels to the left of an input widget, then it is expected that other labels in the layout will also be positioned on the left. Therefore, the label-resolution module works by identifying patterns within the placement of labels relative to input widgets.

Algorithm 1 Resolution of Label to Input Widget

```
1: procedure ResolveLabels($uifs$, $labels$)
2:     $resolved \leftarrow$ init empty list
3:     do
4:         $pairs \leftarrow GenCandidateSets(uifs, labels)$
5:         $labels.remove(pairs.labels)$
6:         $uifs.remove(pairs.uifs)$
7:         $resolved.append(pairs)$
8:     while $size(pairs) > 0$
9:     return $resolved$
```

UiRef's algorithm for label resolution (Algorithm 1) is iterative, so it correctly resolves labels even if inconsistencies exist. As an example, the "Address" label in the layout of Figure 1 is inconsistent as it is placed above the input widget while other labels are placed to the left of the input widgets, and is still successfully resolved by the algorithm. During the first iteration, UiRef resolves the label with the text "First Name" to the EditText with the identifier *@+id/-first*, and "Last Name" to the EditText with the identifier *@+id/last*. During the second iteration, UiRef resolves the label with the text "Address" to the EditText with the identifier *@+id/address*.

The label-resolution module accepts the rendered layouts from the layout-extraction module as input, and resolves labels associated with user-input widgets.

Label-Resolution Algorithm: UiRef maps labels to input widgets by identifying patterns in their relative placements. Algorithm 1 describes the algorithm. The input is the sets of input widgets ($uifs$) and potential labels (*labels*) identified using class-inheritance information within the layout-extraction module (Section 4.1).

Algorithm 1 starts (Line 4) by invoking GenCandidateSets (Algorithm 2) to generate a list of candidate sets of label and input-widget pairs. For each input widget, GenCandidateSets creates a set of vectors from the input widget to all potential labels in the layout (Lines 3–8, Algorithm 2). The vectors represent the euclidean distance (i.e., magnitude) and direction (i.e., angle) between the input widget and label. calcSmallestVector (not shown) creates up to three vectors for each input-widget and label pair. Two vectors go from the two closest corners of the input widget to the corresponding corners of the label. The third vector is created if a label is directly above, below or to the sides of the input widget, being anchored at the closest point between the input widget and label.

If a vector's magnitude is greater than a predetermined threshold, it is not considered as a candidate (Lines 6–7, Algorithm 2). The algorithm then appends the input-widget and label pair to the candidate set under the entry for the corresponding vector if either of the widgets does not already exist (Line 8, Algorithm 2). During implementation, we empirically determine the threshold by taking the average distances of labels to input widgets for 100 randomly sampled applications. The distance threshold can be made independent from the device's screen size by representing the threshold as a proportion of the screen size. Note that the apps in the validation set were excluded from the datasets used in our evaluation.

After the candidate sets are constructed, GetOptimalSet (Algorithm 3) is invoked (Line 9, Algorithm 2) to extract the optimal label to input-widget mapping. The algorithm first retrieves the candidate sets with the largest set size (Line 2, Algorithm 3). If there are multiple such sets, the algorithm prioritizes sets whose labels are either above or to the left of input widgets (i.e., the vector direction is 90 degrees or 180 degrees, respectfully) (Lines 5–8, Algorithm 3).

Algorithm 2 Create Candidate Map

```
procedure GenCandidateSets($uifs$, $labels$)
    $candidates \leftarrow$ init empty candidate map object
    for each $i \in uifs$ do
        for each $l \in labels$ do
            $v \leftarrow calcSmallestVector(i, l)$
            if $v.distance > threshold$ then
                continue
            $candidates[v].appendNE(\{i, l\})$
    return $GetOptimalSet(candidates)$
```

Algorithm 3 Find Optimal Mapping

```
procedure GetOptimalSet($candidates$)
    $maxSets \leftarrow GetLargestSets(candidates)$
    $opt \leftarrow maxSets[0]$
    for each $s \in maxSets[1:]$ do
        $optLorT \leftarrow IsLeftOrTop(opt.vec)$
        $sLorT \leftarrow IsLeftOrTop(s.vec)$
        if $!optLorT$ & $sLorT$ then
            $opt \leftarrow s$
        else if $optLorT == sLorT$ AND $opt.vec.dist > s.vec.dist$ then
            $opt \leftarrow s$
    return $opt$
```

If multiple sets are of equal size and both vector directions are to the left or top of the input widgets, then GetOptimalSet selects the set with the smallest vector distance (Lines 9–10, Algorithm 3). The algorithm returns the optimal label to input-widget mapping.

Once Algorithm 1 receives the optimal mapping of label to input-widget pair, it removes the resolved pairs from the initial lists, and appends the resolved pairs to the resolved set (Lines 5–7, Algorithm 1). This process is repeated until no new labels are resolved.

4.3 Semantics Resolution of Input Widgets

The goal of semantics resolution is to resolve the types of input data prompted by the associated descriptive text. These types of data correspond to *concepts* in our terminology, which are expressed by terms (i.e., single word or phrase) in layouts. However, the mapping of terms to concepts is not necessarily disjoint. Different terms may represent the same concept (synonymy), and the same term may represent multiple concepts (polysemy). Due to polysemy, strategies of keyword-based resolution cannot be used to resolve concepts.

The semantics-resolution module uses the context surrounding a term to resolve its concept. The module has two main tasks: (1) terminology extraction, and (2) concept resolution.

4.3.1 Terminology Extraction. Terminology extraction is used to determine candidate terms that represent concepts in the domain. UiRef requires terminology of security and privacy sensitive concepts that applications use to prompt for inputs. Terms can be a single word (uniterm) or a phrase (multiterm), such as *gender* and *date of birth*. UiRef uses a data-driven technique for defining a list of security and privacy sensitive terms. The list is derived from the text displayed within layouts from the dataset in Section 6.

Our terminology-extraction process uses all text displayed in the layouts extracted by the layout-extraction module. We begin by using regular expressions to replace email addresses, URLs, and common phone-number formats (e.g., a@b.com, http://www.b.com, 123-456-7890) by their respective terms. For example, a@b.com is replaced by email_addr_example. We also substitute the # symbol with the word "number", and use regular expressions to remove prompt text, such as "enter your". We then lemmatize the text, being a common preprocessing step to normalize text. For example, lemmatizing the word "ethnicities" would change it to "ethnicity."

After preprocessing the text, we mark text that contains only a single word as a potential uniterm. To heuristically remove noise and reduce the manual post-processing efforts described below, we remove uniterms that appear only within a single layout. Note that removing uniterms may cause us to miss a few important terms; however, missed terms can be added manually.

Next, we create a candidate set of sensitive multiterms by extracting n-grams of sizes 2-4 for each piece of text. Note that this process uses a sliding window across the words in the text to find the most appropriate groupings of words.

Once the set of n-grams is created, we heuristically refine the set as follows. First, we remove n-grams that appear in less than two separate layouts. Second, we remove n-grams that are a substrings of larger n-grams and appear in only the same layouts. For example, we remove the bigram "social security" if the only time that it appears is in the longer term "social security number". Third, we remove n-grams that start or end in stopwords (e.g., "a", "the", "and"). Fourth, we remove n-grams with common verbs occurring in middle positions of n-grams (e.g., "could", "have", "would"). Note that the goal of multiterm extraction is not to extract textual prompts, but rather to extract terms that correspond to concepts. For example, we seek to extract the term first name from the following text, "What is your first name?" Finally, if a matching unigram is found when spaces are removed from n-grams, we mark the unigram as a potential synonym of the n-gram (e.g., user name and username).

Once the candidate lists are created, we perform two final manual post-processing steps. First, we read through the lists and filter out n-grams representing verb phrases not caught with our filters, and phrases that do not clearly represent concepts. Second, we scan through the lists of n-grams and uniterms, and mark down potentially sensitive terms along with their synonyms. For example, in this step, we mark "last name" as a sensitive term, and "surname" as its synonym. We also create a list of potentially ambiguous terms (e.g., name, address, number), which is used later during semantics resolution to determine which terms require disambiguation.

4.3.2 Concept Resolution. The goal of concept resolution is to determine the semantics of an input widget. For example, UiRef aims to link the concepts *first name*, *last name*, and *postal address* to the corresponding input widgets in Figure 1. Recall from Section 3 that key-phrase matching alone is not sufficient due to polysemy. Before we can disambiguate terms, we must determine the different meanings (i.e., senses) in which a term appears. Note that an automated technique is required, as all possible senses of the term must be considered when performing disambiguation.

The process of determining these different meanings is known as word-sense induction. The process of resolving the meaning of a specific instance of a term is known as word-sense disambiguation. UiRef performs these two tasks using the Adaptive SkipGram (AdaGram) model [8]. AdaGram extends Mikolov's SkipGram model [21] by using a non-parametric Bayesian approach over Dirichlet processes to learn multiple word vectors per word. For a single word, a word vector represents a *sense* in which the word appears.

Word-Sense Induction: Training an AdaGram model to perform word-sense induction requires flat text documents. To flatten text within layouts, UiRef generates a string from the text within layouts, scanning from the layout's top-left corner to the bottom-right corner. For each piece of text, UiRef preprocesses the text by performing lemmatization, stripping stopwords, and transforming the text into lowercase. For input widgets with both labels and hints, UiRef outputs the label text before the hint. For example, Figure 1 produces the following document: "first name last name address".

To allow for the disambiguation of multiterms, UiRef generates the second document for the same layout with multiterms treated as one lexical unit. UiRef collapses adjacent words that form multiterms by replacing spaces with underscores (e.g., "first name" becomes "first_name") using the list of multiterms from the previous step. For example, Figure 1 also produces the following document: "first_name last_name address".

After these two documents are created for each layout, UiRef trains an AdaGram model, which is used to perform word-sense disambiguation. Note that UiRef trains the AdaGram model using a maximum of 25 potential word senses being learned per word. This number is chosen through experimental exploration to avoid underestimating the number of word senses, as doing so would cause concepts to overlap in the same word sense.

Once the model is trained, we manually resolve the concept to which the word vector refers for each potentially ambiguous word by analyzing the terms that have the closest relation to the word vector. For example, the closest related words for one meaning discovered for "address" are *city*, *street*, *first*, *zip*, *postal*. Therefore, we mark the semantics for that meaning of "address' as a *postal address*. The closest related words for another meaning of "address" are *port*, *host username*, *ip*, and *pswd*, which we mark as *IP address*.

Semantics Resolution and Disambiguation: To resolve the semantics of each input widget, UiRef first lemmatizes the hint text, combines multiterms into one lexical unit, and removes stopwords. Next, UiRef uses the *spotter* to scan the hint's preprocessesed text to search for the sensitive terms constructed in Section 4.3.1. If the spotter locates a sensitive term that is unambiguous, UiRef marks the input widget with the term's associated concept. However, if the spotter finds a potentially ambiguous term, UiRef uses the trained AdaGram model to disambiguate the term using the surrounding context with a window size of 5 (i.e., 5 words before and 5 words after). To disambiguate a word, UiRef predicts the probability of the target term given the context, and returns the concept with the highest probability. If the hint text does not contain any sensitive terms, UiRef repeats this process with the associated label's text.

For example, UiRef resolves the input widget with the widget identifier *@+id/address* in Figure 1 as follows. Since the widget does not contain embedded text (e.g., hint or text attributes), UiRef begins by analyzing the text of the label that it resolves for this widget (i.e., "address"). UiRef's spotter finds the word "address" in the text and tags it as ambiguous. UiRef disambiguates address by extracting the surrounding context (first_name last_name), and using the model to predict the meaning of the word. UiRef predicts that "address" refers to the sense that corresponds to a postal address.

5 EVALUATION

In this section, we evaluate the effectiveness of UiRef with respect to its major modules. To evaluate UiRef's layout extraction, we use a combination of emulators and real devices. We use 12 x86 Android emulators and a single Nexus 4, both running Android 5.1.1. We run applications that contain ARM-based native libraries on the real device, as the x86 emulators do not include the translation library.

Table 1: Performance Evaluation Results

	Label Resolution		Semantics Resolution		Disambig.
	UiRef	SUPOR∗	UiRef	SUPOR∗	UiRef
Acc.	84.0%	63.2%	95.0%	90.2%	82.1%
Raw	630/750	474/750	708/745	672/745	275/335

∗ UiRef's Layout Extraction, and our reimplementation of SUPOR

Our dataset is based on the October 31, 2014 PlayDrone [33] snapshot (1.4 million applications). We perform proportionate stratified random sampling across the dataset to ensure a representative sample by using the Google Play categories and number of downloads to form our strata. For verification purposes, we use Python's *langdetect* module to ensure the dataset contains only applications with English descriptions. Our final dataset consists of 50,162 apps.

We use SUPOR [18] as a representative baseline for comparison. Since SUPOR's source or binary code was not available, we reimplement their approach. When specific implementation details are not clear or ambiguous, we contact the authors for clarifications. We do not compare our results to UIPicker due to the limitations discussed in Section 3, and the unavailability of their trained classifiers.

5.1 Layout-Extraction Performance

To evaluate the effectiveness of UiRef's layout-extraction technique, we estimate its improvement over SUPOR's technique for rendering static layouts. As discussed in Section 3, one improvement that UiRef provides over SUPOR is the rendering of custom views. To evaluate the improvement for handing custom views, we estimate a lower bound and upper bound: (1) lower bound: the number of layouts and applications that include custom views that programmatically add other views; and (2) upper bound: the number of layouts and applications that include custom views. Note that we do not evaluate other potential limitations of SUPOR's layout extraction, e.g., the accuracy of ADT's rendering.

Our dataset consists of the 50,162 applications described earlier. To identify custom views that add other views programmatically, we disassemble the APKs using ApkTool and perform class-hierarchy analysis to identify classes whose inheritance hierarchy contains the View class or any other SDK class that extends the View class (e.g., EditText, LinearLayout). Next, we perform a signature-based search on the custom view's underlying code for method invocations used to add views, such as ViewGroup→addView(View child). Since our goal is to provide a conservative estimation rather than an exhaustive analysis, we choose a signature-based technique over reachability analysis to reduce computational overhead.

Results: In total, around 25.6% of the static layouts (479,337/1,873,737) and 48.7% of the applications (24,436/50,162) contain at least one custom view. Further, 35.1% of applications in the dataset contain at least one custom view whose underlying code dynamically adds views (17,597/50,162), which impacts around 5.5% of static layouts (102,671/1,873,737). Therefore, UiRef provides between a 5.5% to 25.6% improvement over SUPOR for extracting layouts with an average extraction time of 53.7 milliseconds per layout.

5.2 Label-Resolution Performance

To evaluate UiRef's label-resolution technique, we manually annotate the label that corresponds to each input widget. We compare our manual annotations with the results from both UiRef and SUPOR's label resolutions. Note that we use UiRef's layout-extraction technique when reporting SUPOR's label-resolution results to allow for a direct comparison between the algorithms, and to reduce potential errors carried over from SUPOR's layout extraction.

We randomly select 12 applications for each of Google Play's 42 categories from the 50k dataset. We run UiRef on the 504 applications to extract layouts. Note that SUPOR handles only EditText widgets. Therefore, we remove layouts that do not contain at least one EditText widget to provide a fair comparison to SUPOR (although UiRef resolves 9 base input widget types), reducing our dataset to 280 applications. We then remove layouts whose screenshot fails or does not capture the entire layout, layouts that are duplicates of another layout, contain non-English text, are a fragment such as a search bar, or do not contain any descriptive text or icon. Our final dataset consists of 349 layouts (109 applications) with 750 input widgets where 420 widgets have associated labels, and 472 are manually tagged as containing sensitive data.

Results: Table 1 shows that UiRef provides a significant 20.8% improvement in accuracy over SUPOR when resolving labels. In total, UiRef correctly resolves the labels for 630/750 input widgets (84.0%) with an average runtime of 16 milliseconds per layout. The majority of UiRef's incorrect label resolutions are due to applications using TextViews to display non-modifiable data alongside user-input widgets, causing UiRef's pattern-based resolution algorithm to choose incorrect candidate sets. In future work, we plan to resolve this problem by applying program analysis to differentiate between labels and such TextViews. The rest of the incorrect label resolutions are due to the multiple labels per input widget (e.g., measurement units), or multiple input widgets per label (tabular arrangements). SUPOR's errors originate from the lack of maximum distance metric always resulting in a resolved label for an input widget, and from its preference for labels located to the left or above input widgets while the correct label is located below the input widget.

5.3 Semantics-Resolution Performance

We next evaluate (1) UiRef's accuracy of resolving semantics; (2) the prevalence of ambiguity during semantics resolution; and (3) the effectiveness of UiRef's disambiguation.

5.3.1 Overall Performance. To evaluate UiRef's semantics resolution, we manually annotate the 349 layouts in Section 5.2 with a semantics label for each input. We remove input widgets from our results whose ambiguity could not be resolved from viewing the screenshot alone (5 input widgets in total). We compare our manual annotations with the results produced by UiRef's semantics resolution algorithm for input widgets, and SUPOR's key-phrase matching using our term list. Note that SUPOR ignores the resolution of ambiguous terms, such as "name", so our re-implementation of SUPOR also ignores ambiguous terms.

Results: Table 1 shows that UiRef achieves 95.0% accuracy when resolving the semantics of input widgets (4.8% increase in accuracy over SUPOR). UiRef's incorrect resolutions are due to UiRef not having enough context to disambiguate the term (14/37), the spotter missing sensitive keywords (13/37), insufficient parsing of the text (5/37), incorrect label resolutions (4/37), and incorrectly marking a non-sensitive input request as sensitive (1/37). Although SUPOR has a 90.2% accuracy with this dataset, such result is an overapproximation due to dataset selection, as the dataset has a low number of input widgets with ambiguous terms, and the majority of input widgets with SUPOR's incorrect label resolutions are resolved using the embedded text attributes (171/192). In Section 5.3.3, we measure the prevalence of ambiguity during semantics resolution.

5.3.2 Prevalence of Ambiguity. To demonstrate the importance of disambiguation when resolving semantics of input widgets, we measure the impact of ambiguity on semantics resolution. We run UiRef's semantics resolution on the 50,162 applications described in Section 5 and output input requests where the descriptive text (i.e., hint, label, or text) used for semantics resolution contains one of the 19 ambiguous terms shown in Table 2. From the 50,162 applications, there are 175,101 input requests requiring semantics resolution across 71,291 layouts and 15,642 applications.

Results: Table 2 shows that the resolution of 20.9% (36,600/175,101) of input requests contain an ambiguous term within the descriptive text used when resolving semantics. Further, ambiguity affects the semantics resolution of 36.1% (25,720/71,291) of the layouts and 63.9% (10,003/15,642) of applications. The prevalence of ambiguity clearly demonstrates limitations of key-phrase-based techniques for resolving semantics. For example, SUPOR ignores the resolution of the term "name", which impacts the resolution of 5.6% (9,753/175,101) of input fields being semantically resolved. Similarly, the term "address" affects 1.8% (3,228/175,101) of semantics resolutions. The pervasiveness of ambiguity when resolving semantics demonstrates the necessity of a disambiguation technique.

5.3.3 Disambiguation Performance. To evaluate UiRef's disambiguation technique, we evaluate UiRef's accuracy on 19 ambiguous terms shown in Table 2. For each ambiguous term, we randomly select 25 input requests that contain the ambiguous term as a hint, label, or text attribute of an input widget from the dataset in Section 5. Note that for terms that do not appear in at least 25 input requests (e.g., "cc"), we select the maximum number of samples available. We substitute layouts with another randomly selected layout if we cannot manually resolve ambiguity when viewing the screenshot. Further, since certain layouts are duplicated across applications and can skew the evaluation (e.g., over-approximating accuracy), we substitute layouts with another randomly selected layout if it is a duplicate of a previously annotated layout for that term. Our dataset consists of 362 input requests from 296 layouts (250 applications). For each layout, we manually resolve ambiguity by viewing the screenshot and XML file and then compare our manual annotation to UiRef's prediction.

Note that UiRef does not resolve the semantics of an input widget requesting non-sensitive data (e.g., cement age for the term "age"), occurring with 26 widgets in our dataset. Such result is due to word-sense induction not learning a *sense* for the concept due to under-representation in the dataset. However, unresolved semantics for non-sensitive requests should not affect the results, as UiRef's goal is to identify sensitive requests. Thus, we remove these 26 widgets from our dataset, resulting in 335 input requests.

Results: Table 2 shows that UiRef achieves an 82.1% accuracy for resolving the ambiguity of sensitive terms (275/335). The results demonstrate that UiRef's disambiguation provides a significant advantage over key-phrase-based techniques, such as SUPOR. In

Table 2: Disambiguation Evaluation Results

Ambiguous Term (<=25 widgets)	**Example Concepts**	**Prevalence of Ambiguity During Semantics Resolution** (175,101 widgets)			**Disambiguation Performance** (335 widgets)	
		Requests	**Layouts**	**Apps**	**Correct**	**Incorrect**
account_number	bank account, utility account	292 (0.2%)	274 (3.8%)	181 (1.2%)	20	3
address	IP address, postal address	3,228 (1.8%)	2,905 (4.1%)	2,310 (14.8%)	22	2
age	person age, product age	334 (0.2%)	297 (0.4%)	198 (1.3%)	21	3
cc	carbon copy, credit card	61 (0.03%)	55 (0.1%)	48 (0.3%)	3	0
cm	person height, product height	1,599 (0.9%)	1,569 2.1%)	1,553 (9.9%)	4	2
day	birth day, current day	7,297 (4.2%)	5,502 (7.7%)	2,095 (13.4%)	1	5
destination	physical location, file location	165 (0.1%)	158 (0.2%)	141 (0.9%)	16	7
first	first name, first place	325 (0.1%)	307 (0.4%)	254 (1.6%)	22	3
ft	person height, product height	1,537 (0.9%)	1,530 (2.1%)	1,521 (9.7%)	1	2
height	person height, product height	246 (0.1%)	233 (0.3%)	148 (0.9%)	19	2
last	last name, last place	201 (0.1%)	176 (0.2%)	144 (0.9%)	20	2
lb	person weight, product weight	3,056 (1.7%)	1,548 (2.2%)	1,533 (9.8%)	2	4
location	physical location, file location	4,577 (2.6%)	4,545 (6.4%)	2,653 (17.0%)	23	2
month	birth month, expiration month	386 (0.2%)	313 (0.4%)	235 (1.5%)	3	6
name	person name, file name	9,753 (5.6%)	9,497 (13.3%)	7,403 (47.3%)	19	5
number	phone number, card number	1,641 (0.9%)	1,330 (1.9%)	995 (6.4%)	20	3
security_code	CCV code, verification code	145 (0.1%)	136 (0.2%)	109 (0.7%)	22	2
weight	person weight, product weight	285 (0.2%)	273 (0.4%)	181 (1.2%)	19	3
year	birth year, expiration year	1,472 (0.8%)	1,394 (1.9%)	1,174 (7.5%)	18	4
Total		36,600 / 175,101 (20.9%)	25,720 / 71,291 (36.1%)	10,003 / 15,642 (63.9%)	275/335 (82.1%)	60 / 335 (17.9%)

particular, UiRef resolves the semantics of the term "address" with 88.0% accuracy (22/25) where SUPOR ignores the term due to ambiguity. UiRef's disambiguation also shows a 91.6% accuracy (22/24) when distinguishing between credit-card verification (CCV) codes and passphrases denoted by the term "security code." In this case, UiRef correctly disambiguates "security code" as CCV code (15 widgets), and as a passphrase (7 widgets).

6 SECURITY AND PRIVACY ANALYSIS

Our primary motivation for creating UiRef was to classify the security and privacy sensitive inputs provided by users. In this section, we use UiRef to perform a large-scale study of 50,162 Google Play applications from Section 5 to understand the scope of questionable security and privacy practices We ran UiRef on the dataset and triaged applications using outlier detection for manual inspection. We aim to explore two main research questions:

RQ1: *What types of security and privacy sensitive information are mobile applications asking for?*

RQ2: *What are the security and privacy implications of sensitive input requests?*

Note that our analysis is not intended to be comprehensive. Rather, we intend to broadly highlight various forms of questionable practices by application developers. Our hope is that our initial findings will drive further research in understanding the security and privacy challenges associated with user input requests. Extended results are available on the project website [5].

6.1 Sensitive Information Requests

To answer RQ1, we consolidated the sensitive input requests identified by UiRef for every application. We grouped this information into 9 categories based on high-level semantics (Table 3). To further understand the sensitive input requests made by applications, we use the following methodology: (1) We manually identify relevant sensitive concepts extracted by UiRef, (2) for interesting input requests, we view the layout's screenshot, and (3) if further clarification is required, we analyze the application's disassembled code. Our analysis results in several interesting findings as follows.

Finding 1: *Applications request a wide-range of security and privacy sensitive information.* Table 3 shows the types of sensitive information requested by applications. The most frequent information requests are related to account or contact information (e.g., usernames/email addresses, passwords), which is due to applications requiring account login or registration. Applications also request a substantial amount of personal information (e.g., the person's name, date of birth, gender, age, race, marital status, religion, and political affiliation). Other requests include sensitive personal identifiers (e.g., SSN - 52 applications, driver's license number - 51 applications). Finally, applications request a range of financial data (e.g., credit card numbers), vehicular data (e.g., vehicle identification numbers—VIN), location data (e.g., postal addresses), and device data (e.g., IMEI). IMEI represents an interesting use case as the applications are asking the user to input the IMEI rather than retrieving the IMEI in the background. Note that for this study, we determine the sensitivity of the terms based on own experiences, however, UiRef's approach is generic and can be easily adapted for alternative terms with different sensitivities (see Section 7).

Finding 2: *Applications directly request third-party passwords defeating the purpose of OAuth-like solutions.* Knowledge of a password leads to full control of users' accounts. A primary goal of OAuth is to allow third-parties to access a user's resources without having direct access to their password. Further, the user can revoke access without changing the password. If the user enters the password directly into the application, the application can still request an OAuth token; however, it eliminates much of OAuth's benefits.

We found 68 applications directly requesting the user's Twitter (55 apps), Gmail (10 apps), Facebook (2 apps), and Adobe (1 app) credentials within internally-defined layouts. Most Twitter password requests (51/55) come from the WinterWell JTwitter library, which requests the user's password within a static Android layout. The remaining 4 applications requesting the user's Twitter password use other third-party libraries to request an OAuth token (e.g., Twitter4J). During our analysis, we found that the JTwitter library also has an option to request OAuth tokens within an embedded WebView. This is still a bad practice, as the application that embeds the WebView can access all of the WebView's data. In fact, Google deprecated support for OAuth requests to Google in embedded WebViews for security concerns [1]. The applications requesting Gmail (10), Facebook (2), and Adobe (1) passwords directly request the data for login purposes and are subject to the same problem.

Finding 3: *Applications frequently request sensitive financial information.* We found that around 8% of applications (4,433/50,162) are

Table 3: Consolidated Sensitive Information Requests

Category	Sensitive Information Requests (# Applications)
Account/Contact	username_or_email (11047), passwd (10251), phone_num (6307), twitter_passwd (55), wifi_passwd (17), gmail_passwd (10), social_media_url (8), wifi_ssid (7), ftp_passwd (6), smtp_passwd (5), facebook_passwd (2), nq_account_passwd (1), mint_passwd (1), gritsafe_passwd (1), exchange_passwd (1), adobe_passwd (1)
Personal	person_name (8057), date_of_birth (2285), gender (1237), company_name (499), person_age (178), person_weight (149), job_title (121), person_height (109), school_name (75), education_info (38), marital_status (37), demographic_info (27), native_language (8), citizenship (7), birth_place (7), marriage_date (5), religion (4), political_affiliation (3),
Financial	credit_card_info (4433), loan_info (1974), bank_account_info (156), salary (116), bank_info (78), house_financial_info (57)
Vehicle/Driver	vehicle_info (173), license_plate (66), vehicle_vin (61), insurance_policy_num (52), vehicle_registration (11), license_expiry_date (3)
Device	device_id (106), mac_address (27), serial_num (26), service_provider (19), device_manufac_info (6), sim_card (6),
Health	medication_name (70), prescription_num (27), drug_dosage (23), blood_pressure (22), blood_type (16), heart_rate (14), body_mass_index (11), blood_glucose (6), doctor_email_id (2)
Personal Id #	SSN (52), driver_license_num (51), id_num (20), tax_id (5), passport_num (3), student_id (1), medicaid_num (1)
Family Member	family_member_name (30), family_member_phone (9), guardian_email (3), mother_birth_place (2)
Location/Travel	location_info (6761), flight_num (28)

requesting the user's credit card information (e.g., credit card number, security code, and expiration date). Exposing credit card details and other financial information to untrusted third-party applications is high risk, as demonstrated by the numerous data breaches in recent years [3]. Further, applications that are not fully compliant to the Payment Card Industry's Data Security Standard [2] (PCI DSS) place the user at potential risk for financial loss and fraud. PCI DSS outlines standards for merchants that handle credit card information, such as encrypted storage and transmission. Based on our findings, we believe that there is a need for deeper analysis to identify non-compliance, and explore alternate payment solutions, such as opting for centralized and well-tested billing processors (e.g., Google's In-app Billing service), to reduce associated risks.

6.2 Identifying Anomalous Input Requests

To explore RQ2, we used outlier analysis to triage applications for manual analysis. We apply an unsupervised technique, called the Ranking-based Outlier Analysis and Detection (ROAD) algorithm [31] at the granularity of Google Play's 25 application categories (games subcategories combined into one category), to generate a ranked list based on the likelihood that a record is an outlier. Note that we modify the distance function of the ROAD algorithm by using a probability-based value weighting function, so that rare attributes have a larger influence on the distance function. An application is considered to be an outlier based on its distance from the largest cluster (or second largest cluster if the largest cluster requests no sensitive input), where clusters are formed based on the types of sensitive information they request. Interestingly, only the Weather category had sensitive input requests associated with the largest cluster. Section A (Appendix) provides a detailed explanation on ROAD and how we use it to detect outliers in our dataset. Table 4 (Appendix) shows an except of the results of our outlier analysis, where the counts represent the number of applications.

Our approach is to use outlier analysis as a filtering mechanism to identify applications that merit further deeper analysis. We focus our manual analysis on the top-10 outlier applications (based on the distance from the largest cluster) for each of the 25 Google Play categories (250 apps in total). Our deeper manual analysis involves reviewing the screenshots of the application's layouts and its disassembled code to explore how the application is using the data. The following discussion reports our high-level findings.

Finding 4: *Applications are making questionable requests asking for too much sensitive data.* Table 4 (Appendix) shows the contrast between expected and unexpected input requests by outlier apps. When the largest cluster requests a concept (e.g., username) it is not unexpected for the outlier to also request it. However, outlier analysis identifies a number of interesting cases where the requested data does not align with the app's category. For example, in the Communication category, outliers request the user's religion, marital status, and demographics. We inspected these apps and found that some request the data to send to advertisers and disclose this purpose to users, while others request it to search for friends.

In the Personalization category, outlier applications ask for data such as the user's birthday, birth place, and bank account information. Through manual inspection, we found the app (com.Chinese_I_Ching_Horoscope_20706) requests the user's birthday and birthplace to provide horoscope readings, but send the data to a third-party website to provide the service without notifying the user. The app (psmainapp-ui) requesting bank account information (bank name and credentials) provides a vault for encrypted data storage. Upon manual analysis, we found that the app uses a constant seed for key generation ('sknpyvvn'), which puts the users' data at risk.

Similarly, in the Game category, outliers ask for a wide-range of personal information, such as the user's name, salary, age, marital status, gender, and SSN, which we discuss further in Findings 5-7.

Finding 5: *Applications disclose sensitive input requests to advertisers.* We found 6 game applications requesting the user's zip code, age, salary, gender, marital status, education information, ethnicity, and political affiliation. On further analysis, we found that they were disclosing this information to the Millennial Media advertising network for targeted advertisements. The apps were all developed by the same developer (Brett Plummer), and 3 of them are relatively popular[1], as they have 100-500k downloads. The apps collect the sensitive data requests in layouts claiming to be used for profile information, which is misleading as it is only used to disclose to advertisers. Further, the privacy policies and layouts do not specify that the sensitive data provided is being disclosed to advertisers.

Finding 6: *Applications include third-party libraries that directly request sensitive information.* We found 2 game apps[2] that include a third-party library (i.e., Skillz eSports) that displays layouts to request a wide-range of sensitive information, such as SSNs, passport numbers, names, addresses, credit card data, and phone numbers. Although we could not successfully run the apps to verify the requests due to crashing, we viewed the Skillz eSports's website and found that it requests this data to allow users to withdraw their winnings. The main concern in such cases is that the user may not understand to which entity they are disclosing information. For example, consider the case where the user trusts the main application and is willing to disclose their SSN to that application, but does not trust Skillz eSports with their information. Since the layouts do not denote who (library or application) is collecting the information, users may potentially expose data to third-parties they do not trust.

[1]com.alaskajim.{football, bible2, rockmusic}
[2]com. binarypumpkin.bingo.{easter,usa}

Finding 7: *Applications are displaying untrusted ads in the same windows as sensitive data requests.* We found the same 6 applications discussed in Finding 5 displaying ads in layouts that requests sensitive input. Prior work [29] showed that untrusted code can traverse UIs to steal private data in layouts. For example, an untrusted widget loaded within the same view hierarchy can obtain the root view, and then traverse back down the hierarchy to search for input widgets with sensitive data. Our finding demonstrates the need for mechanisms such as LayerCake [29].

Takeaway: The findings from our study motivate the need to analyze how apps use sensitive data requested through input widgets. These findings were possible due to the availability of reliable user input semantics from UiRef. Although our analysis is not comprehensive, we found several concerning practices and sensitive input requests with limited manual analysis, further motivating the need for automated analysis to focus on the apps triaged by UiRef.

7 DISCUSSION

Threats to External Validity: The dataset used in the study may not be representative of the entire market. To mitigate this threat, we use proportionate stratified sampling to select our dataset, using the application categories and popularity to form the strata. Further, UiRef's outlier detection assumes that the majority of applications are well behaved and not requesting an extraneous amount of data. Although this assumption is acceptable for Google Play applications, this assumption may not be transferable to other markets.

Threats to Internal Validity: UiRef extracts the statically defined layouts from applications. Applications may also dynamically generate and modify layouts at runtime, and display content in WebViews, which may result in false negatives. Reconstructing dynamic layouts using static analysis is a challenging problem, left as future work. Applications may also include unreachable or unused layouts in the APKs, such as layouts bundled with libraries or as artifacts from testing. As this characteristic can lead to false positives, identifying reachable layouts from the applications' entry points may mitigate this problem. Although textual labels are commonly used for internationalization, icons and images may also denote semantics, leading to false negatives. Integrating optical character and image recognition can be conducted in future work.

Automatically extracting a comprehensive lexicon of security and privacy related terms is an open problem and warrants future investigation. Although an incomplete lexicon may result in false negatives, as UiRef uses it to determine types of information considered private, UiRef's techniques are general and the list can be substituted once a more comprehensive lexicon becomes available. Further, users commonly progress through sequences of layouts to accomplish tasks within applications, such that prior layouts may provide context into the semantics of the current layout. The lack of prior context may lead to imprecision when resolving certain layouts. In future work, we plan to explore leveraging context from prior layouts in workflows to assist in resolving semantics.

8 RELATED WORK

Although there has been a considerable amount of work that focuses on mobile malware analysis, we limit our discussion to privacy analysis of mobile applications. TaintDroid [11] pioneered the area by using dynamic taint analysis to identify Android applications that leak private information, such as GPS location and device identifiers. PiOS [10] and AndroidLeaks [14] are the initial static analysis duals of TaintDroid for iOS and Android, respectively. Due to the various challenges associated with statically analyzing Android applications, a number of additional static-analysis frameworks [12, 13, 15, 24, 25] have been proposed. Other program information than information flows is also used to study privacy in mobile applications. Han et al. [17] compare API calls within iOS and Android applications, and show iOS applications call significantly more privacy-sensitive APIs than their Android counterparts. In prior work, the privacy semantics of information is unambiguous, as it comes from a well-defined APIs. In contrast, UiRef resolves privacy semantics from ambiguous user-input APIs.

The act of sending privacy-sensitive information to the network does not constitute a violation. AppIntent [34] pairs screenshots with potential privacy leaks for human review. Recent research has sought to lessen the need for human review by using natural language processing. WHYPER [26], AutoCog [27], and CHABADA [16] consider the application's text description to help infer the user's expectation of security and privacy relevant actions. AsDroid [19] checks the coherence between UI text and event callbacks triggering sensitive behavior. UiRef complements these approaches by providing comprehensive contextual semantics of the UI. Pluto [9] assesses user data exposure by resolving the semantics of data originating from well-structured files (e.g., SQL, XML, JSON). Closest to our work are SUPOR [18] and UIPicker [22] for which we provide a detailed comparison in Section 3.

Other work has focused on using program analysis to extract the structure and sequence of GUIs as intermediate representations. GATOR [30] extracts GUI-layout hierarchies and transition graphs for test-input generation by performing static reference analysis and control-flow analysis. A^3E [7] constructs activity-transition graphs to drive automated application exploration by using static taint tracking to resolve intents that flow to method invocations that launch activities. Test-input generation and automated exploration tools would benefit from UiRef by using it to resolve the semantics of input widgets, and use such semantics to generate input data.

9 CONCLUSION

While prior studies of Android security and privacy have focused on information from well-defined APIs, they have largely ignored user inputs as a source of sensitive information. In this paper, we have presented UiRef for resolving the security and privacy semantics of data entered into input widgets. We have introduced novel techniques that achieve an overall accuracies of 95.0% at semantics resolution and 82.1% at disambiguation. We have used UiRef to perform a large-scale study of 50,162 apps, and identified concerning practices, including insecure exposure of account passwords and non-consensual input disclosures to third parties. Our findings demonstrate that user-input semantics could provide a unique perspective into improving mobile-app security and privacy.

10 ACKNOWLEDGMENTS

This material is based upon work supported by the Maryland Procurement Office under Contract No. H98230-14-C-0141. This work is also supported in part by NSF grants CNS-1513690, CNS-1513939, CNS-1434582, and a Google Faculty Research Award.

REFERENCES

[1] 2016. Modernizing OAuth interactions in Native Apps for Better Usability and Security. https://developers.googleblog.com/2016/08/modernizing-oauth-interactions-in-native-apps.html. (2016).

[2] 2016. PCI Security Standards Council. https://www.pcisecuritystandards.org/. (2016).

[3] 2016. Privacy Rights Clearinghouse Data Breaches. https://www.privacyrights.org/data-breaches?title=. (2016).

[4] 2017. ApkTool. http://ibotpeaches.github.io/Apktool. (2017).

[5] 2017. UiRef Project Website. https://wspr.csc.ncsu.edu/uiref/. (2017).

[6] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining Apps for Abnormal Usage of Sensitive Data. In *Proceedings of the International Conference on Software Engineering (ICSE).*

[7] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA).*

[8] Sergey Bartunov, Dmitry Kondrashkin, Anton Osokin, and Dmitry Vetrov. 2015. Breaking Sticks and Ambiguities with Adaptive Skip-gram. *arXiv preprint arXiv:1502.07257* (2015).

[9] Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang, and Carl A. Gunter. 2016. Free for All! Assessing User Data Exposure to Advertising Libraries on Android. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS).*

[10] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS).*

[11] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[12] Xinming Ou Fengguo Wei, Sankardas Roy and Robby. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*

[13] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[14] Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. 2012. AndroidLeaks: Automatically Detecting Potential Privacy Leaks In Android Applications on a Large Scale. In *Proceedings of the International Conference on Trust and Trustworthy Computing (TRUST).*

[15] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS).*

[16] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking App Behavior Against App Descriptions. In *Proceedings of the International Conference on Software Engineering (ICSE).*

[17] Jin Han, Qiang Yan, Debin Gao, Jianying Zhou, and Robert Deng. 2013. Comparing Mobile Privacy Protection through Cross-Platform Applications. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS).*

[18] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. 2015. SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps. In *Proceedings of the USENIX Security Symposium.*

[19] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. In *Proceedings of the International Conference on Software Engineering (ICSE).*

[20] Michael Lesk. 1986. Automatic Sense Disambiguation using Machine Readable Dictionaries: How to Tell a Pine Cone from an Ice Cream Cone. In *Proceedings of the International Conference on Systems Documentation.*

[21] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in neural information processing systems.*

[22] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, , and XiaoFeng Wang. 2015. UIPicker: User-Input Privacy Identification in Mobile Applications. In *Proceedings of the USENIX Security Symposium.*

[23] Michael K. Ng, Mark Junjie Li, Joshua Zhexue Huang, and Zengyou He. 2007. On the Impact of Dissimilarity Measure in k-Modes Clustering Algorithm. In *IEEE Transactions on Pattern Analysis and Machine Intelligence.*

[24] Damien Octeau, Somesh Jha, and Patrick McDaniel. 2012. Retargeting Android Applications to Java Bytecode. In *Procedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE).*

[25] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective Inter-component Communication Mapping in Android with EPPIC: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the USENIX Security Symposium.*

[26] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *Proceedings of the USENIX Security Symposium.*

[27] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. AutoCog: Measuring the Description-to-permission Fidelity in Android Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*

[28] Vaibhav Rastogi, Yan Chen, and William Enck. 2013. AppsPlayground: Automatic Large-scale Dynamic Analysis of Android Applications. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY).*

[29] Franziska Roesner and Tadayoshi Kohno. 2013. Securing Embedded User Interfaces: Android and Beyond. In *Proceedings of the USENIX Security Symposium.*

[30] Shengqian Yang and Hailong Zhang and Haowei Wu and Yan Wang and Dacong Yan and Atanas Rountev. 2015. Static Window Transition Graphs for Android. In *IEEE/ACM International Conference on Automated Software Engineering.*

[31] N.N.R. Ranga Suri, Musti Narasimha Murty, and Gopalasamy Athithan. 2012. An Algorithm for Mining Outliers in Categorical Data through Ranking. In *Proceedings of the IEEE International Conference on Hybrid Intelligent Systems (HIS).*

[32] Omer Tripp and Julia Rubin. 2014. A Bayesian Approach to Privacy Enforcement in Smartphones. In *Proceedings of the USENIX Security Symposium.*

[33] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A Measurement Study of Google Play. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS).*

[34] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. 2013. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*

A APPENDIX: OUTLIER DETECTION

Due to space constraints, full details on outlier analysis are available on the project website [5]. The goal of outlier analysis is to identify abnormal input requests that do not align with the app's purpose and functionality. Since different types of apps vary in types of the data that they request, comparing sensitive input requests across a broad range of apps is not sufficient to identify outliers. For example, a financial app may be expected to request the user's income while a game would not be expected to do so. Therefore, outlier analysis is performed at the granularity of Google Play's 25 app categories (18 game subcategories collapsed into one game category).

To detect outliers for each Google Play category, we first manually collapse sensitive concepts into 75 semantic buckets (e.g., first name and last name are grouped in a *person_name* bucket). Second, we apply an unsupervised technique, called the Ranking-based Outlier Analysis and Detection (ROAD) algorithm [31], to generate a ranked list based on the likelihood that a record is an outlier.

We modify the distance function proposed by Ng et al. [23] by multiplying the dissimilarity score by a probability-based value weighting function so that rare attribute values have a larger influence on the score. For example, if most apps do not request the user's SSN, it is more significant that an app requests the user's SSN than not. Therefore, the attribute value of requesting the user's SSN (i.e., rare values) holds more weight in the distance function than the attribute value of not requesting the user's SSN (i.e., frequent values). The weighting function, is the reciprocal of the square root of the probability that the value occurs in the categorical dataset. The probability is the frequency of the value divided by the total number of categorical data records. The weighting function returns a higher weight for rare occurrences of attribute values. Full details on the modified distance function and the ROAD setup are available on the project website [5].

Table 4: Input Request Outliers Excerpt (Full results on project website [5])

Category	k	Cluster	Concepts
Communication	92	Largest (83)*	username_or_email (83), passwd (83)
		Outliers (438)	phone_num (289), username_or_email (275), passwd (252), person_name (195), location (142), credit_card (59), DOB (43), gender (16), loan_info (12), job_title (9), serial_num (6), company_name (5), school_name (4), wifi_passwd (2), twitter_passwd (2), service_provider (2), person_age (2), marital_status (2), mac_addr (2), lic_plate (2), id_num (2), gmail_passwd (2), facebook_passwd (2), education_info (2), device_id (2), bank_acct (2), vehicle_vin (1), vehicle_info (1), smtp_passwd (1), sim_card (1), religion (1), person_wt (1), person_ht (1), family_contact_phone (1), device_manufac_info (1), demographic_info (1)
Finance	125	Largest (60)*	username_or_email (60), passwd (60)
		Outliers (389)	username_or_email (235), passwd (234), person_name (222), location (215), phone_num (164), credit_card (86), DOB (66), bank_acct (54), salary (46), loan_info (43), bank_info (40), SSN (25), gender (25), company_name (19), vehicle_info (10), house_fin_info (5), marital_status (4), job_title (4), driver_lic_num (4), vehicle_vin (3), person_age (3), lic_plate (3), family_member_name (3), school_name (2), insurance_policy_num (2), birthplace (2), vehicle_reg (1), tax_id (1), person_wt (1), mint_passwd (1), id_num (1)
Game	56	Largest (425)*	username_or_email (425), passwd (425)
		Outliers (604)	username_or_email (348), person_name (313), passwd (200), location (142), DOB (114), gender (87), credit_card (86), phone_num (83), person_age (52), demographic_info (10), salary (9), marital_status (9), education_info (9), SSN (3), school_name (3), passport_num (3), twitter_passwd (2), person_wt (2), device_id (2), person_ht (1), loan_info (1), job_title (1), gmail_passwd (1), bank_info (1), bank_acct (1)
Health & Fitness	142	Largest (105)*	username_or_email (105), phone_num (105), passwd (105), location (105), loan_info (105), person_name (105), credit_card (105)
		Outliers (554)	username_or_email (367), passwd (342), person_name (298), location (244), phone_num (194), DOB (193), credit_card (176), gender (105), person_wt (75), person_ht (60), medication_name (18), person_age (17), blood_pressure (10), drug_dosage (9), heart_rate (8), blood_type (8), company_name (7), body_mass_index (7), loan_info (6), family_member_name (5), job_title (4), blood_glucose (4), prescription_num (3), education_info (3), demographic_info (3), birthplace (2), bank_acct (2), SSN (1), service_provider (1), serial_num (1), school_name (1), marital_status (1), insurance_policy_num (1), gmail_passwd (1), doctor_email_id (1)
Libraries & Demo	13	Largest(5)*	passwd (5), username_or_email (4), person_name (3)
		Outliers(18)	person_name (12), username_or_email (10), passwd (10), location (10), phone_num (8), credit_card (6), device_id (3), DOB (3), person_age (2), SSN (1), loan_info (1), gender (1), education_info (1)
Media & Video	48	Largest (49)*	username_or_email (49), passwd (49)
		Outliers (179)	passwd (146), person_name (82), username_or_email (80), location (48), phone_num (45), credit_card (35), DOB (21), gender (9), loan_info (8), wifi_passwd (6), mac_addr (4), ftp_passwd (3), wifi_ssid (2), twitter_passwd (2), smtp_passwd (2), device_id (2), company_name (2), social_media_url (1), medication_name (1)
Medical	88	Largest (87)*	username_or_email (87), phone_num (87), passwd (87), location (87), person_name (87), DOB (87), credit_card (87)
		Outliers (258)	username_or_email (196), person_name (160), passwd (155), phone_num (133), location (109), credit_card (68), loan_info (49), gender (32), medication_name (27), DOB (22), person_age (21), person_wt (15), person_ht (12), blood_pressure (10), drug_dosage (9), prescription_num (6), heart_rate (6), family_member_name (5), family_contact_phone (4), body_mass_index (3), blood_type (3), blood_glucose (2), twitter_passwd (1), SSN (1), serial_num (1), marital_status (1), insurance_policy_num (1), doctor_email_id (1), device_id (1), company_name (1)
Music & Audio	45	Largest (138)*	username_or_email (138), passwd (138)
		Outliers (246)	person_name (156), username_or_email (152), passwd (134), location (97), phone_num (84), credit_card (59), DOB (43), loan_info (24), gender (21), company_name (12), vehicle_info (1), twitter_passwd (1), school_name (1), person_age (1), mac_addr (1), id_num (1), device_manufac_info (1)
News & Magazines	60	Largest (163)*	username_or_email (163), passwd (163)
		Outliers (429)	username_or_email (341), passwd (309), person_name (281), location (219), phone_num (161), credit_card (145), company_name (113), DOB (38), gender (30), loan_info (14), bank_acct (5), service_provider (4), person_age (4), lic_plate (4), twitter_passwd (2), school_name (2), political_affil (2), job_title (2), id_num (2), demographic_info (2)
Personalization	27	Largest (134)*	username_or_email (134), phone_num (134)
		Outliers (93)	username_or_email (59), passwd (58), person_name (52), phone_num (41), location (39), DOB (27), credit_card (27), gender (7), loan_info (2), vehicle_vin (1), vehicle_info (1), serial_num (1), salary (1), lic_plate (1), device_id (1), birthplace (1), bank_info (1)
Productivity	108	Largest (95)*	username_or_email (95), passwd (95)
		Outliers (417)	username_or_email (276), passwd (262), person_name (254), location (199), phone_num (178), credit_card (74), DOB (48), loan_info (46), vehicle_info (43), company_name (43), gender (19), job_title (12), salary (11), insurance_policy_num (7), driver_lic_num (7), lic_plate (6), vehicle_vin (5), SSN (4), person_age (4), marital_status (4), bank_info (4), bank_acct (4), serial_num (3), person_ht (3), device_id (3), twitter_passwd (2), sim_card (2), school_name (2), person_wt (2), gmail_passwd (2), wifi_passwd (1), vehicle_reg (1), student_id (1), religion (1), nq_account_passwd (1), mac_addr (1), gritsafe_passwd (1), family_contact_phone (1), blood_type (1)
Shopping	81	Largest (108)*	password (108), username_or_email_address (72), full_name (72), phone_number (36), location_info (36), loan_info (36), credit_card_info (36)
		Outliers (214)	username_or_email_address (148), location_info (132), password (115), full_name (110), phone_number (98), credit_card_info (73), date_of_birth (26), gender (21), bank_account_info (20), bank_info (17), company_name (15), person_age (5), loan_info (3), vehicle_info (2), twitter_password (2), person_weight (2), person_height (2), vehicle_vin (1), tax_id (1), service_provider (1), school_name (1), native_language (1), marital_status (1), license_plate (1), issue_date (1), flight_number (1), family_member_name (1)
Social	108	Largest (84)*	username_or_email (84), passwd (84)
		Outliers (420)	username_or_email (312), passwd (290), person_name (260), location (208), phone_num (188), DOB (110), credit_card (85), gender (68), person_age (23), loan_info (22), job_title (12), school_name (7), company_name (6), marital_status (4), education_info (4), twitter_passwd (3), blood_type (2), vehicle_info (1), serial_num (1), person_wt (1), flight_num (1), family_member_name (1), family_contact_phone (1), citizenship (1)
Tools	109	Largest (157)*	username_or_email_address (157), password (157), location_info (157), full_name (157), credit_card_info (157), phone_number (156), date_of_birth (155)
		Outliers (634)	password (317), username_or_email_address (301), location_info (219), phone_number (189), full_name (168), credit_card_info (51), date_of_birth (35), gender (28), loan_info (13), company_name (13), person_weight (11), mac_address (10), bank_account_info (10), person_height (9), person_age (6), wifi_password (5), serial_number (5), school_name (5), device_id (5), wifi_ssid (4), salary (4), vehicle_info (3), job_title (3), id_number (3), gmail_password (3), driver_license_number (3), service_provider (2), mother_birth_place (2), license_plate (2), twitter_password (1), social_security_number (1), smtp_password (1), medication_name (1), marital_status (1), insurance_policy_number (1), house_financial_info (1), ftp_password (1), education_info (1), drug_dosage (1), driver_id (1), class_name (1), citizenship (1)
Transportation	89	Largest (35)*	location (35)
		Outliers (244)	username_or_email (185), person_name (152), passwd (147), location (141), phone_num (140), credit_card (71), bank_acct (23), vehicle_info (20), loan_info (13), flight_num (13), DOB (13), company_name (11), gender (7), vehicle_vin (6), insurance_policy_num (5), driver_lic_num (5), device_id (5), lic_plate (3), job_title (3), school_name (2), person_wt (2), person_ht (2), bank_info (2), vehicle_reg (1), twitter_passwd (1), service_provider (1), guardian_email (1), family_member_name (1), family_contact_phone (1), driver_id (1)
Travel & Local	108	Largest (111)*	location (111)
		Outliers (806)	username_or_email (653), passwd (598), person_name (515), location (495), phone_num (371), credit_card (299), gender (129), DOB (115), loan_info (92), company_name (20), flight_num (13), SSN (9), vehicle_info (7), person_age (5), bank_acct (4), service_provider (3), twitter_passwd (2), id_num (2), driver_lic_num (2), driver_id (2), citizenship (2), school_name (1), person_wt (1), person_ht (1), native_language (1), lic_plate (1), job_title (1), demographic_info (1), bank_info (1)
Weather	20	Largest (40)	location (40)
		Outliers (51)	username_or_email (38), location (34), passwd (30), person_name (17), phone_num (9), mac_addr (2), DOB (2), credit_card (2), company_name (1)