WHYPER: Towards Automating Risk Assessment of Mobile Applications

Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, Tao Xie
North Carolina State University, Raleigh, NC, USA
{*rpandit, xxiao2, wei.yang*}*@ncsu.edu* {*enck, xie*}*@csc.ncsu.edu*

Abstract

Application markets such as Apple's App Store and Google's Play Store have played an important role in the popularity of smartphones and mobile devices. However, keeping malware out of application markets is an ongoing challenge. While recent work has developed various techniques to determine what applications do, no work has provided a technical approach to answer, *what do users expect?* In this paper, we present the first step in addressing this challenge. Specifically, we focus on permissions for a given application and examine whether the application description provides any indication for why the application needs a permission. We present WHYPER, a framework using Natural Language Processing (NLP) techniques to identify sentences that describe the need for a given permission in an application description. WHYPER achieves an average precision of 82.8%, and an average recall of 81.5% for three permissions (address book, calendar, and record audio) that protect frequently-used security and privacy sensitive resources. These results demonstrate great promise in using NLP techniques to bridge the semantic gap between user expectations and application functionality, further aiding the risk assessment of mobile applications.

1 Introduction

Application markets such as Apple's App Store and Google's Play Store have become the *de facto* mechanism of delivering software to consumer smartphones and mobile devices. Markets have enabled a vibrant software ecosystem that benefits both consumers and developers. Markets provide a central location for users to discover, purchase, download, and install software with only a few clicks within on-device market interfaces. Simultaneously, they also provide a mechanism for developers to advertise, sell, and distribute their applications. Unfortunately, these characteristics also provide an easy distribution mechanism for developers with malicious intent to distribute malware.

To address market-security issues, the two predominant smartphone platforms (Apple and Google) use starkly contrasting approaches. On one hand, Apple forces all applications submitted to its App Store to undergo some level of manual inspection and analysis before they are published. This manual intervention allows an Apple employee to read an application's description and determine whether the different information and resources used by the application are appropriate. On the other hand, Google performs no such checking before publishing an application. While Bouncer [1] provides static and dynamic malware analysis of published applications, Google primarily relies on permissions for security. Application developers must request permissions to access security and privacy sensitive information and resources. This permission list is presented to the user at the time of installation with the implicit assumption that the user is able to determine whether the listed permissions are appropriate.

However, it is non-trivial to classify an application as malicious, privacy infringing, or benign. Previous work has looked at permissions [2–5], code [6–10], and runtime behavior [11–13]. However, underlying all of this work is a caveat: *what does the user expect?* Clearly, an application such as a *GPS Tracker* is expected to record and send the phone's geographic location to the network; an application such as a *Phone-Call Recorder* is expected to record audio during a phone call; and an application such as *One-Click Root* is expected to exploit a privilege-escalation vulnerability. Other cases are more subtle. The Apple and Google approaches fundamentally differ in who determines whether an application's permission, code, or runtime behavior is appropriate. For Apple, it is an employee; for Google, it is the end user.

We are motivated by the vision of bridging the semantic gap between what the user expects an application to do and what it actually does. This work is a first step in

this direction. Specifically, we focus on permissions and ask the question, *does the application description provide any indication for the application's use of a permission?* Clearly, this hypothesis will work better for some permissions than others. For example, permissions that protect a user-understandable resource such as the address book, calendar, or microphone should be discussed in the application description. However, other low-level system permissions such as accessing network state and controlling vibration are not likely to be mentioned. We note that while this work primarily focuses on permissions in the Android platform and relieving the strain on end users, it is equally applicable to other platforms (e.g., Apple) by aiding the employee performing manual inspection.

With this vision, in this paper, we present WHYPER, a framework that uses Natural Language Processing (NLP) techniques to determine *why* an application uses a *per*mission. WHYPER takes as input an application's description from the market and a semantic model of a permission, and determines which sentence (if any) in the description indicates the use of the permission. Furthermore, we show that for some permissions, the permission semantic model can be automatically generated from platform API documents. We evaluate WHYPER against three popularly-used permissions (address book, calendar, and record audio) and a dataset of 581 popular applications. These three frequently-used permissions protect security and privacy sensitive resources. Our results demonstrate that WHYPER effectively identifies the sentences that describe needs of permissions with an average precision of 82.8% and an average recall of 81.5%. We further investigate the sources of inaccuracies and discuss techniques of improvement.

This paper makes the following main contributions:

- We propose the use of NLP techniques to help bridge the semantic gap between what mobile applications do and what users expect them to do. To the best of our knowledge, this work is the first attempt to automate this inference.
- We evaluate our framework on 581 popular Android application descriptions containing nearly 10,000 natural-language sentences. Our evaluation demonstrates substantial improvement over a basic keyword-based searching.
- We provide a publicly available prototype implementation of our approach on the project website [14].

WHYPER is merely the first step in bridging the semantic gap of user expectations. There are many ways in which we see this work developing. Application descriptions are only one form of input. We foresee possibility in also incorporating the application name, user reviews, and potentially even screen-shots. Furthermore, permissions could potentially be replaced with specific API calls, or even the results of dynamic analysis. We also see great potential in developing automatic or partially manual techniques of creating and fine-tuning permission semantic models.

Finally, this work dovetails nicely with recent discourse concerning the appropriateness of Android permissions to protect user security [15–18]. The overwhelming evidence indicates that most users do not understand what permissions mean, even if they are inclined to look at the permission list [18]. On the other hand, permission lists provide a necessary foundation for security. Markets cannot simultaneously cater to the security and privacy requirements of all users [19], and permission lists allow researchers and expert users to become "whistle blowers" for security and privacy concerns [11]. In fact, a recent comparison [20] of the Android and iOS versions of applications showed that iOS applications overwhelmingly more frequently use privacy-sensitive APIs. Tools such as WHYPER can help raise awareness of security and privacy problems and lower the sophistication required for concerned users to take control of their devices.

The remainder of this paper proceeds as follows. Section 2 presents the overview of the WHYPER framework along with background on NLP techniques used in this work. Section 3 presents our framework and implementation. Section 4 presents evaluation of our framework. Section 6 discusses related work. Finally, Section 7 concludes.

2 WHYPER Overview

We next present a brief overview of the WHYPER framework. The name WHYPER itself is a word-play on phrase *why per*missions. We envision WHYPER to operate between the application market and end users, either as a part of the application market or a standalone system as shown in Figure 1.

The primary goal of the WHYPER framework is to bridge the semantic gap of user expectations by determining why an application requires a permission. In particular, we use application descriptions to get this information. Thus, the WHYPER framework operates between the application market and end users. Furthermore, our framework could also serve to help developers with the feedback to improve their applications, as shown by the dotted arrows between developers and WHYPER in Figure 1.

A straightforward way of realizing the WHYPER framework is to perform a keyword-based search on application descriptions to annotate sentences describing

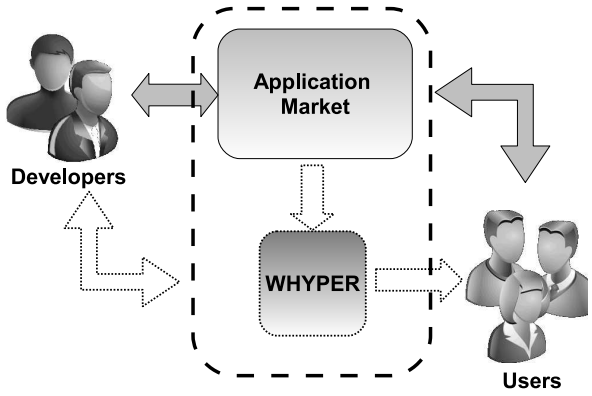

Figure 1: Overview of WHYPER

sensitive operations pertinent to a permission. However, we demonstrate in our evaluation that such an approach is limited by producing many false positives. We propose Natural Language Processing (NLP) as a means to alleviate the shortcomings of keyword-based searching. In particular, we address the following limitations of keyword-based searching:

1. **Confounding Effects**. Certain keywords such as "contact" have a confounding meaning. For instance, *'... displays user contacts, ...'* vs *'... contact me at abc@xyz.com'*. The first sentence fragment refers to a sensitive operation while the second fragment does not. However, both fragments include the keyword "contact".

   To address this limitation, we propose NLP as a means to infer semantics such as whether the word refers to a resource or a generic action.

2. **Semantic Inference**. Sentences often describe a sensitive operation such as *reading contacts* without actually referring to keyword "contact". For instance, *"share... with your friends via email, sms"*. The sentence fragment describes the need for *reading contacts*; however the "contact" keyword is not used.

   To address this limitation, we propose to use API documents as a source of semantic information for identifying actions and resources related to a sensitive operation.

To the best of our knowledge, ours is the first framework in this direction. We next present the key NLP techniques used in this work.

2.1 NLP Preliminaries

Although well suited for human communication, it is very difficult to convert natural language into unambiguous specifications that can be processed and understood by computers. With recent research advances in the area of NLP, existing NLP techniques have been shown to be fairly accurate in highlighting grammatical structure of a natural language sentence. We next briefly introduce the key NLP techniques used in this work.

Parts Of Speech (POS) Tagging [21, 22]. Also known as *'word tagging'*, *'grammatical tagging'* and *'word-sense disambiguation'*, these techniques aim to identify the part of speech (such as nouns and verbs) a particular word in a sentence belongs to. Current state-of-the-art approaches have been shown to achieve 97% [23] accuracy in classifying POS tags for well-written news articles.

Phrase and Clause Parsing. Also known as chunking, this technique further enhances the syntax of a sentence by dividing it into a constituent set of words (or phrases) that logically belong together (such as a Noun Phrase and Verb Phrase). Current state-of-the-art approaches can achieve around 90% [23] accuracy in classifying phrases and clauses over well-written news articles.

Typed Dependencies [24,25]. The Stanford-typed dependencies representation provides a hierarchical semantic structure for a sentence using dependencies with precise definitions of what each dependency means.

Named Entity Recognition [26]. Also known as *'entity identification'* and *'entity extraction'*, these techniques are a subtask of information extraction that aims to classify words in a sentence into predefined categories such as names, quantities, and expressions of time.

We next describe the threat model that we considered while designing our WHYPER framework.

2.2 Use Cases and Threat Model

WHYPER is an enabling technology for a number of use cases. In its simplest form, WHYPER could enable an enhanced user experience for installing applications. For example, the market interface could highlight the sentences that correspond to a specific permission, or raise warnings when it cannot find any sentence for a permission. WHYPER could also be used by market providers to help force developers to disclose functionality to users. In its primitive form, market providers could use WHYPER to ensure permission requests have justifications in the description. More advanced versions of WHYPER could also incorporate the results of static and dynamic application analysis to ensure more semantically appropriate justifications. Such requirements could be placed on all new applications, or iteratively applied to existing applications by automatically emailing developers of applications with unjustified permissions. Alternatively, market providers and security researchers could

use WHYPER to help triage markets [5] for dangerous and privacy infringing applications. Finally, WHYPER could be used in concert with existing crowd-sourcing techniques [27] designed to assess user expectations of application functionality. All of these use cases have unique threat models.

For the purposes of this paper, we consider the generic use scenario where a human is notified by WHYPER if specific permissions requested by an application are not justified by the application's textual description. WHYPER is primarily designed to help identify privacy infringements in relatively benign applications. However, WHYPER can also help highlight malware that attempts to sneak past consumers by adding additional permissions (e.g., to send premium-rate SMS messages). Clearly, a developer can lie when writing the application's description. WHYPER does not attempt to detect such lies. Instead, we assume that statements describing unexpected functionality will appear out-of-place for consumers reading an application description before installing it. We note that malware may still hide malicious functionality (e.g., eavesdropping) within an application designed to use the corresponding permission (e.g., an application to take voice notes). However, false claims in an application's description can provide justification for removal from the market or potentially even criminal prosecution. WHYPER does, however, provide defense against developers that simply include a list of keywords in the application description (e.g., to aid search rankings).

Fundamentally, WHYPER identifies if there a possible implementation need for a permission based on the application's description. In a platform such as Android, there are many ways to accomplish the same implementation goals. Some implementation options require permissions, while others do not. For example, an application can make a call that starts Android's address book application to allow the user to select a contact (requiring no permission), or it can access the address book directly (requiring a permission). From WHYPER's perspective, it only matters that privileged functionality may work expectedly when using the application, and that functionality is disclosed in the application's description. Other techniques (e.g., Stowaway [28]) can be used to determine if an application's code actually requires a permission.

3 WHYPER Design

We next present our framework for annotating the sentences that describe the needs for permissions in application descriptions. Figure 2 gives an overview of our framework. Our framework consists of five components: a preprocessor, an NLP Parser, an intermediate-representation generator, a semantic engine (SE), and an analyzer.

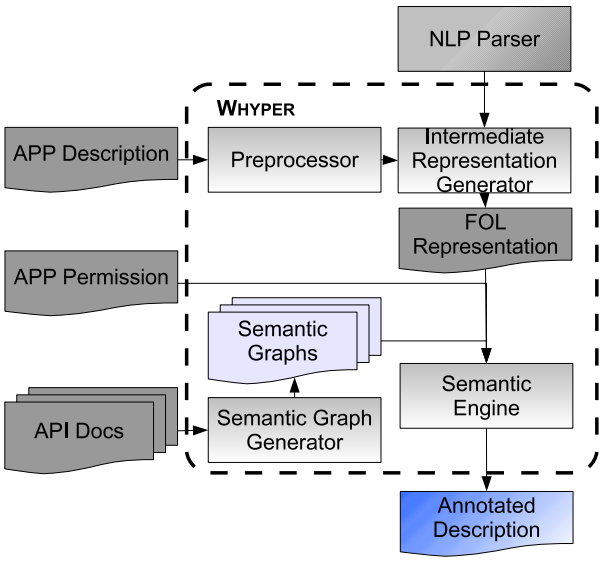

Figure 2: Overview of WHYPER framework

The pre-processor accepts application descriptions and preprocesses the sentences in the descriptions, such as annotating sentence boundaries and reducing lexical tokens. The intermediate-representation generator accepts the pre-processed sentences and parses them using an NLP parser. The parsed sentences are then transformed into the first-order-logic (FOL) representation. SE accepts the FOL representation of a sentence and annotates the sentence based on the semantic graphs of permissions. Our semantic graphs are derived by analyzing Android API documents. We next describe each component in detail.

3.1 Preprocessor

The preprocessor accepts natural-language application descriptions and preprocesses the sentences, to be further analyzed by the intermediate-representation generator. The preprocessor annotates sentence boundaries, and reduces the number of lexical tokens using semantic information. The reduction of lexical tokens greatly increases the accuracy of the analysis in the subsequent components of our framework. In particular, the preprocessor performs following preprocessing tasks:

1. Period Handling. In simplistic English, the character period ('.') marks the end of a sentence. However, there are other legal usages of the period such as: (1) decimal (periods between numbers), (2) ellipsis (three continuous periods '...'), (3) shorthand notations ("Mr.", "Dr.", "e.g."). While these are legal usages, they hinder detection of sentence boundaries, thus forcing the sub-

sequent components to return incorrect or imprecise results.

We pre-process the sentences by annotating these usages for accurate detection of sentence boundaries. We achieve so by looking up known shorthand words from WordNet [29] and detecting decimals, which are also the period character, by using regular expressions. From an implementation perspective, we have maintained a static lookup table of shorthand words observed in WordNet.

2. Sentence Boundaries. Furthermore, there are instances where an enumeration list is used to describe functionality, such as *"The app provides the following functionality: a) abc..., b) xyz... "*. While easy for a human to understand the meaning, it is difficult from a machine to find appropriate boundaries.

We leverage the structural (positional) information: (1) placements of tabs, (2) bullet points (numbers, characters, roman numerals, and symbols), and (3) delimiters such as ":" to detect appropriate boundaries. We further improve the boundary detection using the following patterns we observe in application descriptions:

- We remove the leading and trailing '*' and '-' characters in a sentence.
- We consider the following characters as sentence separators: '–', '- ', 'ø', '§', '†', '⋄', '♢', '♣', '♡', '♠' ... A comprehensive list can be found on the project website [14].
- For an enumeration sentence that contains at least one enumeration phrase (longer than 5 words), we break down the sentence to short sentences for each enumerated item.

3. Named Entity Handling. Sometimes a sequence of words correspond to the name of entities that have a specific meaning collectively. For instance, consider the phrases *"Pandora internet radio"*, *"Google maps"*, which are the names of applications. Further resolution of these phrases using grammatical syntax is unnecessary and would not bring forth any semantic value. Thus, we identify such phrases and annotate them as single lexical units. We achieve so by maintaining a static lookup table.

4. Abbreviation Handling. Natural-language sentences often consist of abbreviations mixed with text. This can result in subsequent components to incorrectly parse a sentence. We find such instances and annotate them as a single entity. For example, text followed by abbreviations such as *"Instant Message (IM)"* is treated as single lexical unit. Detecting such abbreviations is achieved by using the common structure of abbreviations and encoding such structures into regular expressions. Typically, regular expressions provide a reasonable approximation for handling abbreviations.

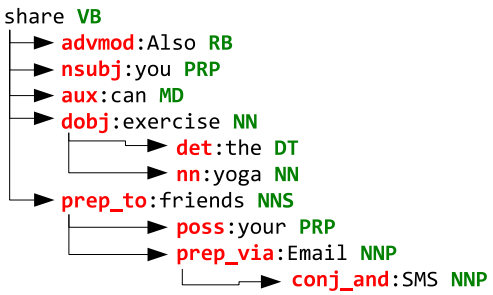

Figure 3: Sentence annotated with Stanford dependencies

3.2 NLP Parser

The NLP parser accepts the pre-processed documents and annotates every sentence within each document using standard NLP techniques. From an implementation perspective, we chose the Stanford Parser [30]. However, this component can be implemented using any other existing NLP libraries or frameworks:

1. **Named Entity Recognition:** NLP parser identifies the named entities in the document and annotates them. Additionally, these entities are further added to the lookup table, so that the preprocessor use the entities for processing subsequent sentences.
2. **Stanford-Typed Dependencies:** [24, 25] NLP parser further annotates the sentences with Stanford-typed dependencies. Stanford-typed dependencies is a simple description of the grammatical relationships in a sentence, and targeted towards extraction of textual relationships. In particular, we use standford-typed dependencies as an input to our intermediate-representation generator.

Next we use an example to illustrate the annotations added by the NLP Parser. Consider the example sentence ***"Also you can share the yoga exercise to your friends via Email and SMS."***, that indirectly refers to the READ_CONTACTS permission. Figure 3 shows the sentence annotated with Stanford-typed dependencies. The words in red are the names of dependencies connecting the actual words of the sentence (in black). Each word is followed by the Part-Of-Speech (POS) tag of the word (in green). For more details on Stanford-typed dependencies and POS tags, please refer to [24, 25].

3.3 Intermediate-Representation Generator

The intermediate-representation generator accepts the annotated documents and builds a relational represen-

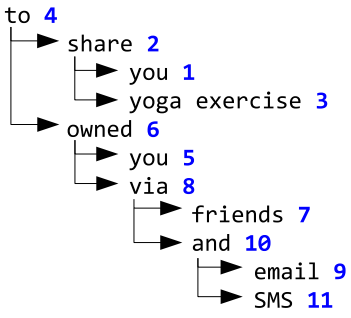

Figure 4: First-order logic representation of annotated sentence in Figure 3

tation of the document. We define our representation as a tree structure that is essentially a First-Order-Logic (FOL) expression. Recent research has shown the adequacy of using FOL for NLP related analysis tasks [31–33]. In our representation, every node in the tree except for the leaf nodes is a predicate node. The leaf nodes represent the entities. The children of the predicate nodes are the participating entities in the relationship represented by the predicate. The first or the only child of a predicate node is the governing entity and the second child is the dependent entity. Together the governing entity, predicate and the dependent entity node form a tuple.

We implemented our intermediate-representation generator based on the principle shallow parsing [34] techniques. A typical shallow parser attempts to parse a sentence based on the function of POS tags. However, we implemented our parser as a function of Stanford-typed dependencies [22, 24, 25, 30]. We chose Stanford-typed dependencies for parsing over POS tags because Stanford-typed dependencies annotate the grammatical relationships between words in a sentence, thus provide more semantic information than POS tags that merely highlight the syntax of a sentence.

In particular, our intermediate-representation generator is implemented as a series of cascading finite state machines (FSM). Earlier research [31, 33–36] has shown the effectiveness and efficiency of using FSM in linguistics analysis such as morphological lookup, POS tagging, phrase parsing, and lexical lookup. We wrote semantic templates for each of the typed dependencies provided by the Stanford Parser.

Table 1 shows a few of these semantic templates. Column "Dependency" lists the name of the Stanford-typed dependency, Column "Example" lists an example sentence containing the dependency, and Column "Description" describes the formulation of tuple from the dependency. All of these semantic templates are publicly available on our project website [14]. Figure 4 shows the FOL representation of the sentence in Figure 3. For ease of understanding, read the words in the order of the numbers following them. For instance, *"you"* ← *"share"* → *"yoga exercise"* forms one tuple. Notice the additional predicate "owned" annotated 6 in the Figure 4, does not appear in actual sentence. The additional predicate "owned" is inserted when our intermediate-representation generator encounters the possession modifier relation (annotated "poss" in Figure 3).

The FOL representation helps us effectively deal with the problem of *confounding effects* of keywords as described in Section 2. In particular, the FOL assists in distinguishing between a resource that would be a leaf node and an action that would be a predicate node in the intermediate representation of a sentence. The generated FOL representation of the sentence is then provided as an input to the semantic engine for further processing.

3.4 Semantic Engine (SE)

The Semantic Engine (SE) accepts the FOL representation of a sentence and based on the semantic graphs of Android permissions annotates a sentence if it matches the criteria. A semantic graph is basically a semantic representation of the resources which are governed by a permission. For instance, the READ_CONTACTS permission governs the resource "CONTACTS" in Android system.

Figure 5 shows the semantic graph for the permission READ_CONTACTS. A semantic graph primarily constitutes of subordinate resources of a permission (represented in rectangular boxes) and a set of available actions on the resource itself (represented in curved boxes). Section 3.5 elaborates on how we build such graphs systematically.

Our SE accepts the semantic graph pertaining to a permission and annotates a sentence based on the algorithm shown in Algorithm 1. The Algorithm accepts the FOL representation of a sentence *rep*, the semantic graph associated with the resource of a permission *g* and a boolean value *recursion* that governs the recursion. The algorithm outputs a boolean value *isPStmt*, which is `true` if the statement describes the permission associated with a semantic graph (*g*), otherwise `false`.

Our algorithm systematically explores the FOL representation of the sentence to determine if a sentence describes the need for a permission. First, our algorithm attempts to locate the occurrence of associated resource name within the leaf node of the FOL representation of the sentence (Line 3). The method `findLeafContaining(name)` explores the FOL representation to find a leaf node that contains term `name`. Furthermore, we use WordNet and Lemmatisation [37] to deal with synonyms of a word in question to find appropriate matches. Once a leaf node is found, we systematically traverse the tree from the leaf node to the root,

Table 1: Semantic Templates for Stanford Typed Dependencies

S. No.	Dependency	Example	Description
1	**conj**	*"Send via SMS and email."* **conj_and**(email, SMS)	Governor (**SMS**) and dependent (**email**) are connected by a relationship of conjunction type(**and**)
2	**iobj**	*"This App provides you with beautiful wallpapers."* **iobj**(provides, you)	Governor (**you**) is treated as dependent entity of relationship resolved by parsing the dependent's (**provides**) typed dependencies
3	**nsubj**	*"This is a scrollable widget."* **nsubj**(widget, This)	Governor (**This**) is treated as governing entity of relationship resolved by parsing the dependent's (**widget**) typed dependencies

matching all parent predicates as well as immediate child predicates [Lines 5-16].

Our algorithm matches each of the traversed predicate with the actions associated with the resource defined in semantic graph. Similar to matching entities, we also employ WordNet and Lemmatisation [37] to deal with synonyms to find appropriate matches. If a match is found, then the value `isPStmt` is set to `true`, indicating that the statement describes a permission.

In case no match is found, our algorithms recursively search all the associated subordinate resources in the semantic graph of current resource. A subordinate resource may further have its own subordinate resources. Currently, our algorithm considers only immediate subordinate resources of a resource to limit the false positives.

In the context of the FOL representation shown in Figure 4, we invoke Algorithm 1 with the semantic graph shown in Figure 5. Our algorithm attempts to find a leaf node containing term "CONTACT" or some of its synonym. Since the FOL representation does not contain such a leaf node, algorithm calls itself with semantic graphs of subordinate resources (Line 17-25), namely 'NUMBER', 'EMAIL', 'LOCATION', 'BIRTHDAY', 'ANNIVERSARY'.

The subsequent invocation will find the leaf-node "email" (annotated 9 in Figure 4). Our algorithm then explores the preceding predicates and finds predicate "share" (annotated 2 in Figure 4). The Algorithm matches the word "share" with action "send" (using Lemmatisation and WordNet similarity), one of the actions available in the semantic graph of resource 'EMAIL' and returns `true`. Thus, the sentence is appropriately identified as describing the need for permission READ_CONTACT.

Algorithm 1 Sentence_Annotator

Input: K_Graph g, FOL_rep *rep*, Boolean *recursion*
Output: Boolean *isPStmt*

```
1:  Boolean isPStmt = false
2:  String r_name = g.resource_Name
3:  FOL_rep r' = rep.findLeafContaining(r_name)
4:  List actionList = g.actionList
5:  while (r'.hasParent) do
6:      if actionList.contains(r'.parent.predicate) then
7:          isPStmt = true
8:          break
9:      else
10:         if actionList.contains(r'.leftSibling.predicate) then
11:             isPStmt = true
12:             break
13:         end if
14:     end if
15:     r' = r'.parent
16: end while
17: if ((NOT(isPStmt)) AND recursion) then
18:     List resourceList = g.resourceList
19:     for all (Resource res in resourceList) do
20:         isPStmt = Sentence_Annotator(getKGraph(res), rep, false)
21:         if isPStmt then
22:             break
23:         end if
24:     end for
25: end if
26: return isPStmt
```

3.5 Semantic-Graph Generator

A key aspect of our proposed framework is the employment of a semantic graph of a permission to perform deep semantic inference of sentences. In particular, we propose to initially infer such graphs from API documents. For third-party applications in mobile devices, the relatively limited resources (memory and computation power compared to desktops and servers) encourage development of thin clients. *The key insight to leverage API documents is that mobile applications are predominantly thin clients, and actions and resources provided by API documents can cover most of the functionality performed by these thin clients*.

Manually creating a semantic graph is prohibitively time consuming and may be error prone. We thus came up with a systematic methodology to infer such semantic graphs from API documents that can potentially be automated. First, we leverage Au et al.'s work [38] to find the API document of the class/interface pertaining to a particular permission. Second, we identify the corresponding resource associated with the permission from the API class name. For instance, we identify 'CONTACTS' and 'ADDRESS BOOK' from the

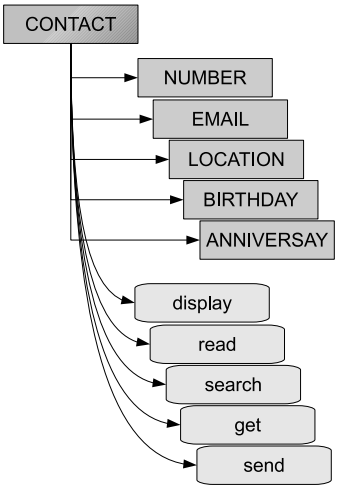

Figure 5: Semantic Graph for the READ_CONTACT permission

`ContactsContract.Contacts`[1] class that is associated with READ_CONTACT permission. Third, we systematically inspect the member variables and member methods to identify actions and subordinate resources.

From the name of member variables, we extract noun phrases and then investigate their types for deciding whether these noun phrases describe resources. For instance, one of member variables of `ContactsContract.Contracts` class leads us to its member variable "email" (whose type is `ContactsContract.CommonDataKinds.Email`). From this variable, we extract noun phrase "EMAIL" and classify the phrase as a resource.

From the name of an Android API public method (describing a possible action on a resource), we extract both noun phrases and their related verb phrases. The noun phrases are used as resources and the verb phrases are used as the associated actions. For instance, `ContactsContract.Contacts` defines operations Insert, Update, Delete, and so on. We consider those operations as individual actions associated with 'CONTACTS' resource.

The process is iteratively applied to the individual subordinate resources that are discovered for an API. For instance, "EMAIL" is identified as a subordinate resource in "CONTACT" resource. Figure 5 shows a sub-graph of graph for READ_CONTACT permission.

4 Evaluation

We now present the evaluation of WHYPER. Given an application, the WHYPER framework bridges the semantic gap between user expectations and the permissions it requests. It does this by identifying in the application description the sentences that describe the need for a given permission. We refer to these sentences as *permission sentences*. To evaluate the effectiveness of WHYPER, we compare the permission sentences identified by WHYPER to a manual annotation of all sentences in the application descriptions. This comparison provides a quantitative assessment of the effectiveness of WHYPER. Specifically, we seek to answer the following research questions:

- **RQ1**: What are the precision, recall and F-Score of WHYPER in identifying permission sentences (i.e., sentences that describe need for a permission)?
- **RQ2**: How effective WHYPER is in identifying permission sentences, compared to keyword-based searching ?

4.1 Subjects

We evaluated WHYPER using a snapshot of popular application descriptions. This snapshot was downloaded in January 2012 and contained the top 500 free applications in each category of the Google Play Store (16,001 total unique applications). We then identified the applications that contained specific permissions of interest.

For our evaluation, we consider the READ_CONTACTS, READ_CALENDAR, and RECORD_AUDIO permissions. We chose these permissions because they protect tangible resources that users understand and have significant enough security and privacy implications that developers should provide justification in the application's description. We found that 2327 applications had at least one of these three permissions. Since our evaluation requires manual effort to classify each sentence in the application description, we further reduced this set of applications by randomly choosing 200 applications for each permission. This resulted in a total of 600 unique applications for these permissions. The set was further reduced by only considering applications that had an English description). Overall, we analysed 581 application descriptions, which contained 9,953 sentences (as parsed by WHYPER).

4.2 Evaluation Setup

We first manually annotated the sentences in the application descriptions. We had three authors independently annotate sentences in our corpus, ensuring that each sentence was annotated by at least two authors. The individual annotations were then discussed by all three authors to reach to a consensus. In our evaluation, we annotated

[1] `http://developer.android.com/reference/android/provider/ContactsContract.Contacts.html`

Table 2: Statistics of Subject permissions

Permission	#N	#S	S_P
READ_CONTACTS	190	3379	235
READ_CALENDAR	191	2752	283
RECORD_AUDIO	200	3822	245
TOTAL	581	9953	763

#N: Number of applications that requests the permission; #S: Total number of sentences in the application descriptions; S_P: Number of sentences manually identified as permission sentences.

a sentence as a permission sentence if at least two authors agreed that the sentence described the need for a permission. Otherwise we annotated the sentence as a permission-irrelevant sentence.

We applied WHYPER on the application descriptions and manually measured the number of `true positives` (TP), `false positives` (FP), `true negatives` (TN) and `false negatives` (FN) produced by WHYPER as follows:

1. TP: A sentence that WHYPER correctly identifies as a permission sentence.
2. FP: A sentence that WHYPER incorrectly identifies as a permission sentence.
3. TN: A sentence that WHYPER correctly identifies as not a permission sentence.
4. FN: A sentence that WHYPER incorrectly identifies as not a permission sentence.

Table 2 shows the statistics of the subjects used in the evaluations of WHYPER. Column "Permission" lists the names of the permissions. Column "#N" lists the number of applications that requests the permissions used in our evaluations. Column "#S" lists the total number of sentences in application descriptions. Finally, Column "S_P" lists the number of sentences that are manually identified as permission sentences by authors. We used this manual identification (Column "S_P") to quantify the effectiveness of WHYPER in identifying permission sentences by answering RQ1. The results of Column "S_P" is also used to compare WHYPER with keyword-based searching to answer RQ2 described next.

For RQ2, we applied keyword-based searching on the same subjects. We consider a word as a keyword in the context of a permission if it is a synonym of the word in the permission. To minimize manual efforts, we used words present in `Manifest.permission class` from Android API. Table 4 shows the keywords used in our evaluation. We then measured the number of `true positives` (TP'), `false positives` (FP'), `true negatives` (TN'), and `false negatives` (FN') produced by the keyword-based searching as follows:

1. TP':- A sentence that is a permission sentence and contains the keywords.
2. FP':- A sentence that is not a permission sentence but contains the keywords.
3. TN':- A sentence that is not a permission sentence and does not contain the keywords.
4. FN':- A sentence that is a permission sentence but does not contain the keywords.

In statistical classification [39], *Precision* is defined as the ratio of the number of true positives to the total number of items reported to be true, and *Recall* is defined as the ratio of the number of true positives to the total number of items that are true. *F-score* is defined as the weighted harmonic mean of Precision and Recall. *Accuracy* is defined as the ratio of sum of true positives and true negatives to the total number of items. Higher values of precision, recall, F-Score, and accuracy indicate higher quality of the permission sentences inferred using WHYPER. Based on the total number of TPs, FPs, TNs, and FNs, we computed the precision, recall, F-score, and accuracy of WHYPER in identifying permission sentences as follows:

$$Precision = \frac{TP}{TP + FP}$$
$$Recall = \frac{TP}{TP + FN}$$
$$F\text{-}score = \frac{2\ X\ Precision\ X\ Recall}{Precision + Recall}$$
$$Accuracy = \frac{TP+TN}{TP + FP+TN + FN}$$

4.3 Results

We next describe our evaluation results to demonstrate the effectiveness of WHYPER in identifying contract sentences.

4.3.1 RQ1: Effectiveness in identifying permission sentences

In this section, we quantify the effectiveness of WHYPER in identifying permission sentences by answering RQ1. Table 3 shows the effectiveness of WHYPER in identifying permission sentences. Column "Permission" lists the names of the permissions. Column "S_I" lists the number of sentences identified by WHYPER as permission sentences. Columns "TP", "FP", "TN", and "FN" represent the number of `true positives`, `false positives`, `true negatives`, and `false negatives`, respectively. Columns "P(%)", "R(%)", "F_S(%)", and "Acc(%)" list percentage values of `precision`, `recall`, `F-score`, and `accurary` respectively. Our results show that, out of 9,953 sentences, WHYPER effectively identifies permission sentences with the average precision, recall, F-score, and

Table 3: Evaluation results

Permission	S_I	TP	FP	FN	TN	P (%)	R (%)	F_S (%)	*Acc* (%)
READ_CONTACTS	204	186	18	49	2930	91.2	79.1	84.7	97.9
READ_CALENDAR	288	241	47	42	2422	83.7	85.1	84.4	96.8
RECORD_AUDIO	259	195	64	50	3470	75.9	79.7	77.4	97.0
TOTAL	751	622	129	141	9061	82.8*	81.5*	82.2*	97.3*

* Column average; S_I: Number of sentences identified by WHYPER as permission sentences; TP: Total number of True Positives; FP: Total number of False Positives; FN: Total number of False Negatives; TN: Total number of True Negatives; P: Precision; R: Recall; F_S: F-Score; and *Acc*: Accuracy

accuracy of 82.8%, 81.5%, 82.2%, and 97.3%, respectively.

We also observed that out of 581 applications whose descriptions we used in our experiments, there were only 86 applications that contained at least one false negative statement that were annotated by WHYPER. Similarly, among 581 applications whose descriptions we used in our experiments, there were only 109 applications that contained at least one false positive statement that were annotated by WHYPER.

We next present an example to illustrate how WHYPER incorrectly identifies a sentence as a permission sentence (producing false positives). False positives are particularly undesirable in the context of our problem domain, because they can mislead the users of WHYPER into believing that a description actually describes the need for a permission. Furthermore, an overwhelming number of false positives may result in user fatigue, and thus devalue the usefulness of WHYPER.

Consider the sentence "*You can now turn recordings into ringtones.*". The sentence describes the application functionality that allows users to create ringtones from previously recorded sounds. However, the described functionality does not require the permission to record audio. WHYPER identifies this sentence as a permission sentence. WHYPER correctly identifies the word *recordings* as a resource associated with the record audio permission. Furthermore, our intermediate representation also correctly constructs that action *turn* is being performed on the resource *recordings*. However, our semantic engine incorrectly matches the action *turn* with the action *start*. The later being a valid semantic action that is permitted by the API on the resource *recording*. In particular, the general purpose WordNet-based Similarity Metric [37] shows 93.3% similarity. We observed that a majority of false positives resulted from incorrect matching of semantic actions against a resources. Such instances can be addressed by using domain-specific dictionaries for synonym analysis.

Another major source of FPs is the incorrect parsing of sentences by the underlying NLP infrastructure. For instance, consider the sentence "*MyLink Advanced provides full synchronization of all Microsoft Outlook emails (inbox, sent, outbox and drafts), contacts, calendar, tasks and notes with all Android phones via USB.*". The sentence describes the users calendar will be synchronized. However, the underlying Stanford parser [30] is not able to accurately annotate the dependencies. Our intermediate-representation generator uses shallow parsing that is a function of these dependencies. An inaccurate dependency annotation causes an incorrect construction of intermediate representation and eventually causes an incorrect classification. Such instances will be addressed with the advancement in underlying NLP infrastructure. Overall, a significant number of false positives will be reduced by as the current NLP research advances the underlying NLP infrastructure.

We next present an example to illustrate how WHYPER fails to identify a valid permission sentence (false negatives). Consider the sentence "*Blow into the mic to extinguish the flame like a real candle*". The sentence describes a semantic action of blowing into the microphone. The noise created by blowing will be captured by the microphone, thus implying the need for record audio permission. WHYPER correctly identifies the word *mic* as a resource associated with the record audio permission. Furthermore, our intermediate representation also correctly shows that the action *blow into* is performed on the resource *mic*. However, from API documents, there is no way for WHYPER framework to infer the knowledge that blowing into microphone semantically implies recording of the sound. Thus, WHYPER fails to identify the sentence as a permission sentence. We can reduce a significant number of false negatives by constructing better semantic graphs.

Similar to reasons for false positives, a major source of false negatives is the incorrect parsing of sentences by the underlying NLP infrastructure. For instance, consider the sentence "*Pregnancy calendar is an application that,not only allows, after entering date of last period menstrual,to calculate the presumed (or estimated) date of birth; but, offering prospects to show,week to week,all appointments which must to undergo every mother,ad a rule,for a correct and healthy pregnancy.*" [2] The sen-

[2]Note that the incorrect grammar, punctuation, and spacing are a

tence describes that the users calendar will be used to display weekly appointments. However, the length and complexity in terms of number of clauses causes the underlying Stanford parser [30] to inaccurately annotate the dependencies, which eventually results into incorrect classification.

We also observed that in a few cases, the process followed to identify sentence boundaries resulted in extremely long and possibly incorrect sentences. Consider a sentence that our preprocessor did not identify as a permission sentence for READ_CALENDER permission:

Daily Brief "How does my day look like today" "Any meetings today" "My reminders" "Add reminder" Essentials Email, Text, Voice dial, Maps, Directions, Web Search "Email John Subject Hello Message Looking forward to meeting with you tomorrow" "Text Lisa Message I will be home in an hour" "Map of Chicago downtown" "Navigate to Millenium Park" "Web Search Green Bean Casserole" "Open Calculator" "Opean Alarm Clock" "Launch Phone book" Personal Health Planner ... "How many days until Christmas" Travel Planner "Show airline directory" "Find hotels" "Rent a car" "Check flight status" "Currency converter" Cluzee Car Mode Access Daily Brief, Personal Radio, Search, Maps, Directions etc..

A few fragments (sub-sentences) of this incorrectly marked sentence describe the need for read calender permission ("*...My reminder ... Add reminder ...*"). However, inaccurate identification of sentence boundaries causes the underlying NLP infrastructure produces a incorrect annotation. Such incorrect annotation is propagated to subsequent phases of WHYPER, ultimately resulting in inaccurate identification of a permission sentence. Overall, as the current research in the filed of NLP advances the underlying NLP infrastructure, a significant number of false negatives will be reduced.

4.3.2 RQ2: Comparison to keyword-based searching

In this section, we answer RQ2 by comparing WHYPER to a keyword-based searching approach in identifying permission sentences. As described in Section 4.2, we manually measured the number of permission sentences in the application descriptions. Furthermore, we also manually computed the precision (P), recall (R), f-score (F_S), and accuracy (Acc) of WHYPER as well as precision (P'), recall (R'), f-score (F_S'), and accuracy (Acc') of keyword-based searching in identifying permission sentences. We then calculated the improvement in using WHYPER against keyword-based searching as $\Delta P = P - P'$, $\Delta R = R - R'$, $\Delta F_S = F_S - F_S'$, and $\Delta Acc = Acc - Acc'$. Higher values of ΔP, ΔR, ΔF_S, and ΔAcc are indicative of better performance of WHYPER against keyword-based search.

Table 5 shows the comparison of WHYPER in identifying permission sentences to keyword-based searching approach. Columns "ΔP", "ΔR", "ΔF_S", and "ΔAcc" list percentage values of increase in the precision, recall, f-scores, and accuracy respectively. Our results show that, in comparison to keyword-based searching, WHYPER effectively identifies permission sentences with the average increase in precision, F-score, and accuracy of 41.6%, 27.2%, and 7.7% respectively. We indeed observed a decrease in average recall by 1.2%, primarily due to poor performance of WHYPER for RECORD_AUDIO permission.

However, it is interesting to note that there is a substantial increase in precision (average 46.0%) in comparison to keyword-based searching. This increase is attributed to a large false positive rate of keyword-based searching. In particular, for descriptions related to READ_CONTACTS permission, WHYPER resulted in only 18 false positives compared to 265 false positives resulted by keyword-based search. Similarly, for descriptions related to RECORD_AUDIO, WHYPER resulted in 64 false positives while keyword-based searching produces 338 false positives.

We next present illustrative examples of how WHYPER performs better than keyword-based search in context of false positives. One major source of false positives in keyword-based search is confounding effects of certain keywords such as *contact*. Consider the sentence "*contact me if there is a bad translation or you'd like your language added!*". The sentence describes that developer is open to feedback about his application. A keyword-based searching incorrectly identifies this sentence as a permission sentence for READ_CONTACTS permission. However, the word *contact* here refers to an action rather than a resource. In contrast, WHYPER correctly identifies the word *contact* as an action applicable to pronoun *me*. Our framework thus correctly classifies the sentences as a permission-irrelevant sentence.

Table 4: Keywords for Permissions

S. No	Permission	Keywords
1	READ_CONTACTS	contact, data, number, name, email
2	READ_CALENDAR	calendar, event, date, month, day, year
3	RECORD_AUDIO	record, audio, voice, capture, microphone

Table 5: Comparison with keyword-based search

Permission	ΔP%	ΔR%	ΔF_S%	ΔAcc%
READ_CONTACTS	50.4	1.3	31.2	7.3
READ_CALENDAR	39.3	1.5	26.4	9.2
RECORD_AUDIO	36.9	-6.6	24.3	6.8
Average	41.6	-1.2	27.2	7.7

Consider another sentence "*To learn more, please visit our Checkmark Calendar web site: calendar.greenbeansoft.com*" as an instance of confounding effect of keywords. The sentence is incorrectly identified as a permission sentence for READ_CALENDAR permission because it contains keyword *calendar*. In contrast, WHYPER correctly identifies "Checkmark Calendar" as a named entity rather than resource *calendar*. Our framework thus correctly identifies the sentences as not a permission sentence.

Another common source of false positives in keyword-based searching is lack of semantic context around a keyword. For instance, consider the sentence "*That's what this app brings to you in addition to learning numbers!*". A keyword-based search classifies this sentence as an permission sentence because it contains the keyword *number*, which is one of the keywords for READ_CONTACTS permission as listed in Table 4. However, the sentence is actually describing the generic numbers rather than phone numbers. Similar to keyword-based search, our framework also identifies word *number* as a candidate match for subordinating resource *number* in READ_CONTACTS permission (Figure 5). However, the identified semantic action on candidate resource *number* for this sentence is *learning*. Since *learning* is not an applicable action to *phone number* resource in our semantic graphs, WHYPER correctly classifies the sentences as not a permission sentence.

The final category, where WHYPER performed better than keyword-based search, is due to the use of synonyms. For instance, *address book* is a synonym for *contact* resource. Similarly *mic* is synonym for *microphone* resource. Our framework, leverages this synonym information in identifying the resources in a sentence. Synonyms could potentially be used to augment the list of keywords in keyword-based search. However, given that keyword-based search already suffers from a very high false positive rate, we believe synonym augmentation to keywords would further worsen the problem.

We next present discussions on why WHYPER caused a decline in recall in comparison to keyword-based search. We do observe a small increase in recall for READ_CONTACTS (1.3%) and READ_CALENDAR (1.5%) permission related sentences, indicating that WHYPER performs slightly better than keyword-based search. However, WHYPER performs particularly worse in RECORD_AUDIO permission related descriptions, which results in overall decrease in the recall compared to keyword-based search.

One of the reasons for such decline in the recall is an outcome of the false negatives produced by our framework. As described in Section 4.3.1 incorrect identification of sentence boundaries and limitations of underlying NLP infrastructure caused a significant number of false negatives in WHYPER. Thus, improvement in these areas will significantly decrease the false negative rate of WHYPER and in turn, make the existing gap negligible.

Another cause of false negatives in our approach is inability to infer knowledge for some 'resource'-*'semantic action'* pairs, for instance, 'microphone'-*'blow into'*. We further observed, that with a small ***manual effort in augmenting semantic graphs*** for a permission, we could significantly bring down the false negatives of our approach. For instance, after a precursory observation of false negative sentences for RECORD_AUDIO permission manually, we augmented the semantic graphs with just two resource-*action* pairs (1. microphone-*blow into* and 2. call-*record*). We then applied WHYPER with the augmented semantic graph on READ_CONTACTS permission sentences.

The outcome increased ΔR value from -6.6% to 0.6% for RECORD_AUDIO permission and an average increase of 1.1% in ΔR across all three permissions, without affecting values for ΔP. We refrained from including such modifications for reporting the results in Table 5 to stay true to our proposed framework. In the future, we plan to investigate techniques to construct better semantic graphs for permissions, such as mining user comments and forums.

4.4 Summary

In summary, we demonstrate that WHYPER effectively identifies permission sentences with the average precision, recall, F-score, and accuracy of 80.1%, 78.6%, 79.3%, and 97.3% respectively. Furthermore, we also show that WHYPER performs better than keyword-based search with an average increase in precision of 40% with a relatively small decrease in average recall (1.2%). We also provide discussion that such gap in recall can be alleviated by improving the underlying NLP infrastructure and a little manual effort. We next present discussions on threats to validity.

4.5 Threats to Validity

Threats to external validity primarily include the degree to which the subject permissions used in our evaluations were representative permissions. To minimize the threat, we used permissions that guard against the resources that can be subjected to privacy and security sensitive operations. The threat can be further reduced by evaluating WHYPER on more permissions. Another threat to external validity is the representativeness of the description sentences we used in our experiments. To minimize the threat we randomly selected actual Android application descriptions from a snapshot of the meta-data of

16001 applications from Google Play store (dated January 2012).

Threats to internal validity include the correctness of our implementation in inferring mapping between natural language description sentences and application permissions. To reduce the threat, we manually inspected all the sentences annotated by our system. Furthermore, we ensured that the results were individually verified and agreed upon by at least two authors. The results of our experiments are publicly available on the project website [14].

5 Discussions and Future work

Our framework currently only takes into account application descriptions and Android API documents to highlight permission sentences. Thus, our framework can semi-formally enumerate the uses of a permission. This information can be leveraged in the future to enhance searching for desired applications.

Furthermore, the outputs from our framework could be used in conjunction with program analysis techniques to facilitate effective code reuse. For instance, our framework outputs the reasons of why a permission is needed for an application. These reasons are usually the functionalities provided by the application. In future work, we plan to locate the code fragments that implement the described functionalities. Such mapping can be used as indexes to existing code searching approaches [40,41] to facilitate effective reuse.

Modular Applications: In our evaluations, we encountered cases where a description referring to another application where the permission sentences were described. For instance, consider the following description sentence *"Navigation2GO is an application to easily launch Google Maps Navigation."*.

The description states that the current application will launch another application "Google Maps Navigation", and thus requires the permissions required by that application. Currently, our framework does not deal with such cases. We plan to implement deeper semantic analysis of description sentences to identify such permission sentences.

Generalization to Other Permissions: WHYPER is designed to identify the textual justification for permissions that protect "user understandable" information and resources. That is, the permission must protect an information source or resource that is in the domain of knowledge of general smartphone users, as opposed to a low-level API only known to developers. The permissions studied in this paper (i.e., address book, calendar, microphone) fall within this domain. Based on our studies, we expect similar permissions, such as those that protect SMS interfaces and data stores, the ability to make and receive phone calls, read call logs and browser history, operate and administer Bluetooth and NFC, and access and use phone accounts will have similar success with WHYPER.

Due to current developer trends and practices, there is class of permissions that we expect will raise alarms for many applications when evaluated with WHYPER. Recent work [7, 11] has shown that many applications leak geographic location and phone identifiers without the users knowledge. We recommend that deployments of WHYPER first focus on other permissions to better gauge and account for developer response. Once general deployment experience with WHYPER has been gained, these more contentious permissions should be tackled. We believe that adding justification for access to geographic location and phone identifiers in the application's textual description will benefit users. For example, if an application uses location for ads or analytics, the developer should state this in the application description.

Finally, there are some permissions that are implicitly used by applications and therefore will have poor results with WHYPER. In particular, we do not expect the Internet permission to work well with WHYPER. Nearly all smartphone applications access the Internet, and we expect attempts to build a semantic graph for the Internet permission will be largely ineffective.

Results Presentation: A potential after-effect of using WHYPER on existing application descriptions might be more verbose application descriptions. One can argue that it would lead to additional burden on end users to read a lengthy description. However, such additional description provides an opportunity for the users to make informed decisions instead of making assumptions. In future work, we plan to implement and evaluate interfaces to present users with information in a more efficient way, countering user fatigue in case of lengthy descriptions. For example, we may consider using icons in different colors to represent permissions with and without explanation.

6 Related Work

Our proposed framework touches quiet a few research areas such as mining Mobile Application Stores, NLP on software engineering artifacts, program comprehension and software verification. We next discuss relevant work pertinent to our proposed framework in these areas.

Harman et al. [42], first used mining approaches on the application description, pricing, and customer ratings in Blackberry App Store. In particular, they use light-weight text analytics to extract feature information, which is then combined with pricing and customer rating to analyze applications' business aspects. In contrast, WHYPER uses deep semantic analysis of sentences

in application descriptions, which can be used as a complimentary approach to their feature extraction.

The use of NLP techniques is not new in the software engineering domain. NLP has been used previously in requirements engineering [31, 32, 43]. NLP has even been used to assess the usability of API docs [44].

There are existing approaches [33, 45–47] that apply either NLP or a mix of NLP and machine learning algorithms to infer specifications from the natural-language elements such as code comments, API descriptions, and formal requirements documents. The semi-formal structure of natural-language documents used in these approaches facilitate the application of NLP techniques required for these problem areas. Furthermore, these approaches often rely on some meta-information from source code as well. In contrast, our proposed framework targets a relatively more unconstrained natural-language text and is independent of the source code of the application under analysis.

With respect to program comprehension there are existing techniques that assist in building domain-specific ontologies [48]. Furthermore, there are existing approaches [49, 50] that automatically infer natural-language documentation from source code. These approaches would immensely help in comprehension of the functionality of a mobile application. However, the inherent dependency on source code to generate such documents poses a problem in cases, where source code is not available. In contrast, WHYPER relies on application description and API documents that are readily available artifact for mobile applications.

In the realm of mobile software verification, there is existing work on permissions [2–5, 15], code [6–10], and runtime behavior [11–13] to detect malicious applications. In particular, Zhou et al. [3] propose an approach that leverages the permission information in the manifest of the applications as a criteria to filter malicious applications. They further employed static analysis to identify the malicious applications by forming behavioral patterns in terms of sequences of API calls of known malicious applications. They also propose the use of heuristics-based dynamic analysis to detect previously unknown applications. Furthermore, Enck et al. [11] also use dynamic analysis techniques (dynamic taint analysis) to detect misuse of users private information.

These previously described techniques are primarily targeted towards finding malicious applications in mobile applications. However, these approaches do little for bridging the semantic gap between what the user expects an application to do and what it actually does. In contrast, WHYPER is the first step targeted towards bridging this gap. Furthermore, WHYPER can be used in conjunction with these approaches for an improved experience while interacting with mobile ecosystem.

In addition, Felt et al. [28] apply automated testing techniques to find permission required to invoke each method in the Android 2.2 API. They use this information to detect over-privilege in Android Applications, by generating a maximum set of permissions required by an applications and comparing them to the permissions requested by the application. Although it is important from a developer perspective to minimize the set of permissions requested, the information does not empower an end user to decide what the requested permissions are being used for. In contrast, WHYPER leverages some of the results provided by Felt et al. [28] to highlight the sentences describing the need for a permission, in turn enabling end users to make informed decision while installing and application

Finally, Lin et al. [27] introduce a new model for privacy, namely *privacy as expectations* for mobile applications. In particular, they use crowd-sourcing as means to capture users expectations of sensitive resources used by a mobile applications. We believe the sentences highlighted by WHYPER, can be used as supporting evidence to formulate such user expectations at the first place.

7 Conclusion

In this paper, we have presented WHYPER, a framework that uses Natural Language Processing (NLP) techniques to determine why an application uses a permission. We evaluated our prototype implementation of WHYPER on real-world application descriptions that involve three permissions (address book, calendar, and record audio). These are frequently-used permissions that protect privacy and security sensitive resources. Our evaluation results show that WHYPER achieves an average precision of 82.8%, and an average recall of 81.5% for three permissions. In summary, our results demonstrate great promise in using NLP techniques to bridge the semantic gap of user expectations to aid the risk assessment of mobile applications.

8 Acknowledgments

This work was supported in part by an NSA Science of Security Lablet grant at North Carolina State University, NSF grants CCF-0845272, CCF-0915400, CNS-0958235, CNS-1160603, CNS-1222680, and CNS-1253346. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies. We would also like to thank the conference reviewers and shepherds for their feedback in finalizing this paper.

References

[1] H. Lockheimer, "Android and security," Google Mobile Blog, Feb. 2012, http://googlemobile.blogspot.com/2012/02/android-and-security.html.

[2] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proc. 1st ACM SPSM Workshop*, 2011, pp. 3–14.

[3] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious Apps in official and alternative Android markets," in *Proc. of 19th NDSS*, 2012.

[4] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of Android Apps," in *Proc. of 19th ACM CCS*, 2012, pp. 241–252.

[5] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck, "MAST: Triage for market-scale mobile malware analysis," in *Proc. 6th of ACM WiSec*, 2013, pp. 13–24.

[6] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting privacy leaks in iOS applications."

[7] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of Android application security," in *Proc. 20th USENIX Security Symposium*, 2011, p. 21.

[8] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. of IEEE Symposium on Security and Privacy*, 2012, pp. 95–109.

[9] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: Scalable and accurate zero-day Android malware detection," in *Proc. of 10th MobiSys*, 2012, pp. 281–294.

[10] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale," in *Proc. of 5th TRUST*, 2012, pp. 291–307.

[11] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. of 9th USENIX OSDI*, 2010, pp. 1–6.

[12] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications," in *Proc. 18th ACM CCS*, 2011, pp. 639–652.

[13] L. K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis," in *Proc. of 21st USENIX Security Symposium*, 2012, p. 29.

[14] "Whyper," https://sites.google.com/site/whypermission/.

[15] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proc. of 16th ACM CCS*, 2009, pp. 235–245.

[16] D. Barrera, H. G. Kayacik, P. C. van Oorshot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to Android," in *Proc. of 7th ACM CCD*, 2010, pp. 73–84.

[17] A. P. Felt, K. Greenwood, and D. Wagner, "The effectiveness of application permissions," in *Proc. of 2nd USENIX WebApps*, 2011, pp. 7–7.

[18] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension and behavior," in *Proc. of 8th SOUPS*, 2012, p. 3.

[19] P. McDaniel and W. Enck, "Not so great expectations: Why application markets haven't failed security," *IEEE Security & Privacy Magazine*, vol. 8, no. 5, pp. 76–78, 2010.

[20] J. Han, Q. Yan, D. Gao, J. Zhou, and R. Deng, "Comparing mobile privacy protection through cross-platform applications," in *Proc. of 20th NDSS*, 2013.

[21] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer, "Feature-rich part-of-speech tagging with a cyclic dependency network." in *Proc. HLT-NAACL*, 2003, pp. 252–259.

[22] D. Klein and D. Manning, Christopher, "Fast exact inference with a factored model for natural language parsing," in *Proc. 15th NIPS*, 2003, pp. 3 – 10.

[23] "The Stanford Natural Language Processing Group," 1999, http://nlp.stanford.edu/.

[24] M. C. de Marneffe, B. MacCartney, and C. D. Manning, "Generating typed dependency parses from phrase structure parses," in *Proc. 5th LREC*, 2006, pp. 449–454.

[25] M. C. de Marneffe and C. D. Manning, "The stanford typed dependencies representation," in *Proc. Workshop COLING*, 2008, pp. 1–8.

[26] J. R. Finkel, T. Grenager, and C. Manning., "Incorporating non-local information into information extraction systems by gibbs sampling," in *Proc. 43nd ACL*, 2005, pp. 363–370.

[27] J. Lin, N. Sadeh, S. Amini, J. Lindqvist, J. I. Hong, and J. Zhang, "Expectation and purpose: understanding users' mental models of mobile App privacy through crowdsourcing," in *Proc. 14th ACM Ubicomp*, 2012, pp. 501–510.

[28] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. of 18th ACM CCS*, 2011, pp. 627–638.

[29] Fellbaum et al., *WordNet: an electronic lexical database.* Cambridge, Mass: MIT Press, 1998.

[30] D. Klein and C. D. Manning, "Accurate unlexicalized parsing." in *Proc. 41st ACL*, 2003, pp. 423–430.

[31] A. Sinha, A. M. Paradkar, P. Kumanan, and B. Boguraev, "A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases." in *Proc. 39th DSN*, 2009, pp. 327–336.

[32] A. Sinha, S. M. SuttonJr., and A. Paradkar, "Text2Test: Automated inspection of natural language use cases," in *Proc. 3rd ICST*, 2010, pp. 155–164.

[33] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language API descriptions," in *Proc. 34th ICSE*, 2012, pp. 815–825.

[34] B. K. Boguraev, "Towards finite-state analysis of lexical cohesion." in *Proc. 3rd FSMNLP*, 2000.

[35] M. Stickel and M. Tyson, *FASTUS: A Cascaded Finite-state Transducer for Extracting Information from Natural-language Text.* MIT Press, 1997.

[36] G. Gregory, *Light Parsing as Finite State Filtering.* Cambridge University Press, 1999.

[37] Q. Do, D. Roth, M. Sammons, Y. Tu, and V. Vydiswaran, "Robust, light-weight approaches to compute lexical similarity," University of Illinois, Computer Science Research and Technical Reports, 2009.

[38] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: analyzing the Android permission specification," in *Proc. 19th CCS*, 2012, pp. 217–228.

[39] D. Olson, *Advanced Data Mining Techniques*. Springer Verlag, 2008.

[40] S. Thummalapenta and T. Xie, "PARSEWeb: A programmer assistant for reusing open source code on the web," in *Proc. 22nd ASE*, 2007, pp. 204–213.

[41] S. P. Reiss, "Semantics-based code search," in *Proc. 31st ICSE*, 2009, pp. 243–253.

[42] M. Harman, Y. Jia, and Y. Zhang, "App store mining and analysis: MSR for app stores," in *Proc. 9th IEEE MSR*, 2012, pp. 108–111.

[43] V. Gervasi and D. Zowghi, "Reasoning about inconsistencies in natural language requirements," *ACM Transactions Software Engineering Methodologies*, vol. 14, pp. 277–330, 2005.

[44] U. Dekel and J. D. Herbsleb, "Improving API documentation usability with knowledge pushing," in *Proc. 31st ICSE*, 2009, pp. 320–330.

[45] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*iComment: Bugs or bad comments?*/," in *Proc. 21st SOSP*, 2007, pp. 145–158.

[46] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language API documentation," in *Proc. 24th ASE*, November 2009, pp. 307–318.

[47] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie, "Automated extraction of security policies from natural-language software documents," in *Proc. 20th FSE*, 2012, pp. 12:1–12:11.

[48] H. Zhou, F. Chen, and H. Yang, "Developing application specific ontology for program comprehension by combining domain ontology with code ontology," in *Proc. 8th QSIC*, 2008, pp. 225 –234.

[49] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Generating parameter comments and integrating with method summaries," in *Proc. 19th ICPC*, 2011, pp. 71–80.

[50] P. Robillard, "Schematic pseudocode for program constructs and its computer automation by schemacode," *Comm. of the ACM*, vol. 29, no. 11, pp. 1072–1089, 1986.