Code between the Lines: Semantic Analysis of Android Applications

Johannes Feichtner[1,2] and Stefan Gruber[1]

[1] Institute of Applied Information Processing and Communications (IAIK)
Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria
[2] Secure Information Technology Center – Austria (A-SIT)
Seidlgasse 22, 1030 Vienna, Austria

Abstract. Static and dynamic program analysis are the key concepts researchers apply to uncover security-critical implementation weaknesses in Android applications. As it is often not obvious in which context problematic statements occur, it is challenging to assess their practical impact. While some flaws may turn out to be bad practice but not undermine the overall security level, others could have a serious impact. Distinguishing them requires knowledge of the designated app purpose. In this paper, we introduce a machine learning-based system that is capable of generating natural language text describing the purpose and core functionality of Android apps based on their actual code. We design a dense neural network that captures the semantic relationships of resource identifiers, string constants, and API calls contained in apps to derive a high-level picture of implemented program behavior. For arbitrary applications, our system can predict precise, human-readable keywords and short phrases that indicate the main use-cases apps are designed for. We evaluate our solution on 67,040 real-world apps and find that with a precision between 69% and 84% we can identify keywords that also occur in the developer-provided description in Google Play. To avoid incomprehensible black box predictions, we apply a model explaining algorithm and demonstrate that our technique can substantially augment inspections of Android apps by contributing contextual information.

1 Introduction

As many Android applications perform security-critical tasks, it is crucial to validate their implementation security using static and dynamic program analysis. In recent years, researchers have elaborated various approaches to disclose possible leaks of private data, identify malware, or to uncover security deficiencies in Android apps. Typically, the results of these analyses fall into two categories: firstly, a classification into malevolent or harmless or, secondly, concrete results of specific aspects the inspection has been aiming for. While both types may be adequate with regards to the particular objectives, they barely evolve to a superior level where the implemented behavior and context of program statements is also taken into account.

In practice, missing context awareness leads to situations where researchers disclose security flaws in execution traces but are unable to comprehend the impact or relevance of the finding in terms of the actual purpose of an application. E.g., basically, it is problematic if a constant, hard-coded key is used for encryption. However, if this happens within an advertisement library where encryption is only used for obfuscation, the impact of the finding needs to be assessed differently. Similarly, it depends on the use-case of an app whether supplying GPS information via HTTPS to an external entity, such as for assistance in traffic navigation, is a legitimate action or the undesirable leakage of sensitive data.

In a broader sense, these examples highlight what analyses are currently unable to cover: the *semantic understanding* of applications. Rather than gaining a high-level picture of the functionality and security of a program, common approaches for inspection focus on single instructions at the lowest possible level. While this is undoubtedly a legitimate level to determine the immediate effects on memory calls and registers, we are still missing a platform that enables us to reason about the effects of coherent code parts on the overall program state.

Augmenting app analysis by contextual information, such as the intended purpose and designated functionality, is of utmost importance to obtain a holistic picture of app behavior. However, currently no solutions exist that could relate the metadata of an app with their actual implementation. This situation is aggravated by the fact that developer-provided descriptions are often minimal, inaccurate, and miss key information. Within this context, we formulate the following problems: *(1) Which attributes of an application describe its behavior? (2) How to identify the main purpose of an app? (3) What keywords and phrases should be included in a description text to represent an app's functionality?*

In this paper, we introduce a solution that infers the main purpose of Android apps based on their implementation. Leveraging the recent advances in neural networks, our work attempts to capture and classify semantic relationships between apps. Our system works unsupervised, involves no labeling of data sets, and is trained with real-world app samples that are only coarsely pre-filtered, e.g., regarding the language of descriptions. The output is not only a prediction of what functionality our systems believes to be realized within an application. Using a model explanation algorithm, we also obtain an insight into what is relevant in apps, can explain the reasoning of predictions, and based on this knowledge, derive meaningful keywords and short phrases in natural language.

In summary, we make the following key contributions:

- To infer the functionality from Android app implementations, we propose a combination of three dense neural networks that combine knowledge extracted from resource identifiers, string constants, and API calls. Our system delivers concise keywords and short phrases that describe the main purpose of apps[3].
- We train, validate, and test our models with 67,040 apps from Google Play. In a case study, we demonstrate the practical relevance and plausibility of predictions by contrasting them with the developer-provided app description.

[3] Our implementation is available at: https://github.com/sg10/apk-verbalizer

– To assess the quality of our system and to avoid incomprehensible black box predictions, we apply the model explaining algorithm SHAP [8]. It enables us to understand the influence of network input features on the derived output.

The outcome of this work represents a notable contribution towards a holistic analysis of Android applications. It helps researchers and users to foster an understanding of what functionality is actually implemented in Android apps.

2 Related Work

Aligning the description of Android apps with the alleged functionality and permission usage has become a growing field of research. In the following, we present related work on this topic and point out differences to our solution.

Behavior Modeling. Hamedani et al. [4] strive to find the most appropriate category for an app based on 14 implementation-related features that are processed using different classification algorithms. As shown in a case study by Kowalczyk et al. [6], using as many app attributes as possible for classification, tends to depict apps more accurately and improves the performance of all kinds of analysis tasks. Takahashi et al. [11] follow this principle and consider several thousand features. They combine permissions, API methods, categories, and presumed cluster assignments to identify malware based on Support Vector Machines. CLANdroid [12], in contrast, aims to identify similar apps by defining semantic anchors that refer to sensor information, permissions, intents, and identifiers. Based thereon, they use Latent Semantic Index to derive a matrix representation for every app. MalDozer [5] leverages a convolutional neural network to find harmful behavior by discretizing sequences of invoked API methods. FlowCog [9] adopts natural language processing to infer whether apps provide sufficient semantics for users to understand privacy risks emerging from the information flow.

App Descriptions. Among all related research, the work of Zhang et al. [15] comes closest to this paper. By extracting keywords contained in the call graph of permission-related API calls, the authors intend to derive a description of security-related app behavior. For instance, if the call graph contains the method `KeyguardManager.isKeyguardLocked()`, it is modeled as the words "the phone", "be", "locked". To assess the quality of app descriptions, Kuznetsov et al. [7] extract identifiers and strings from an app's XML definitions and semantically compare them with the developer-provided description text.

Sensitive APIs. In a combination of static code inspection and text analysis, Watanabe et al. [14] present a keyword-based technique to correlate access to privacy-relevant resources with app descriptions. AutoCog [10] correlates permission-related API calls with frequently occurring text fragments. As a result, semantic patterns are derived that can provide an insight into why Android apps request certain permissions. With a focus on potential abuse of sensitive APIs, Gorla et al. [3] derive app clusters based on pre-labeled description topics. Related to that, the approach of Gao et al. [2] infers expectable permissions by applying statistical correlation coefficients after mining topics from descriptions using NLP techniques and Latent Dirichlet Allocation (LDA).

3 Behavior Modeling of Android Apps

A naïve approach to identify functionality implemented in Android applications would be to statically define rules for classifying source code. However, the evolving nature of smartphone apps with constantly changing APIs and the usage of third-party libraries would make it cumbersome to spot and label specific behavior. As a remedy, our approach leverages modern methods of machine learning that work unsupervised and involve no prior labeling of data sets.

Before designing a neural network that predicts the main purpose of apps, we need to tackle a basic question: *Which attributes of an app describe its behavior?* Users can answer this question intuitively by installing and testing an application. Vendors would refer to the source code to derive similar conclusions. The approach presented in this paper is inspired by both perspectives and focuses on information sources that are included within the code and resources of Android app archives.

We attempt to model Android app behavior from two different angles. On the one hand, we consider static string resources that indicate what an app does from a user's and developer's perspective. On the other hand, we describe a program by the Android API calls it includes, e.g., to access sensitive information, draw UI effects, or implement event listeners. Based on the presence and co-occurrence of calls, we expect to see individual patterns that characterize different functionality.

In the following, we outline the features our neural network will use as an input to infer a semantic understanding of the purpose of apps:

- **App Resource Identifiers**: *Semantic information provided by developers.* In order to access resources, such as UI elements, graphics, or multilingual definitions from program code, Android uses IDs that unambiguously identify individual elements. Although these values can be chosen arbitrarily during development, they usually correspond semantically to the resource content.

- **String Constants**: *UI text and functional descriptions, shown to the user.* Static UI elements, language variables, and URLs are typically stored within app resources. When shown to the user, these constants provide valuable semantic information regarding the purpose of an app and actions users can perform. E.g., if an app includes UI elements containing the string values "new transaction", "account balance", and "money transfer", its implemented functionality most likely targets financial transactions.

- **API Calls**: *Define how an app interacts with the Android OS environment.* The widespread use of third-party libraries, code obfuscation techniques, and the multitude of possible usage scenarios make it challenging to identify the individual semantics for every code block. We, thus, postulate that the behavior of apps is not (only) determined by the interaction of individual code fragments, but especially by their interaction with the operating system and users. Consequently, to infer implementation behavior, we focus on calls to APIs of the Android framework. By their modus operandi, they, e.g., control access to sensitive user data, device sensors, visual effects, media processing, and networks and, thus, clearly define the functionality of apps.

As each of these three feature types is embedded within a different semantic context, it is not viable to simply collect all occurrences and use them as a combined input for a neural network. For a more accurate representation, we propose to train three separate neural networks that take different input features but share the same underlying architecture and produce the same type of output.

4 Semantic App Analysis

We design a dense neural architecture to infer the implemented functionality from real-world Android applications. Our goal is to develop a system that can process an unknown app archive and delivers keywords and short phrases that describe the main purpose. To remediate the "black box" usually associated with neural networks, we require that our solution provides an insight into which input features are decisive for predictions.

In the **training** phase, we train three separate neural networks with Android app archives and their developer-provided descriptions from Google Play. For each app, we first extract all relevant semantic features, weigh their importance using TF-IDF and use the resulting vector as input for the corresponding neural network. In parallel, we build a TF-IDF model with app description texts that will be used to derive a neural network output in natural language.

In the **prediction** phase, our system receives an app not seen during training. After deriving and processing a TF-IDF vector representation of all features included in a given archive, each neural network will return a list of key words and short phrases that commonly occur in app descriptions when certain input features are used. As shown in Figure 1, the output of the network for predicting description words based on given string constants may, e.g., consist of the words *sms*, *messenger*, and *friends*, with the adjacent decimal value expressing the relevance of the predictions. An algorithm for explaining neural networks called SHAP (see Section 4.4) is then applied to find out which input features contribute most to the prediction of these output tokens. Summarizing the outputs of all three models and sorting them regarding the shown relevance provides us with a ranked list of description fragments that describe the main purpose of an app.

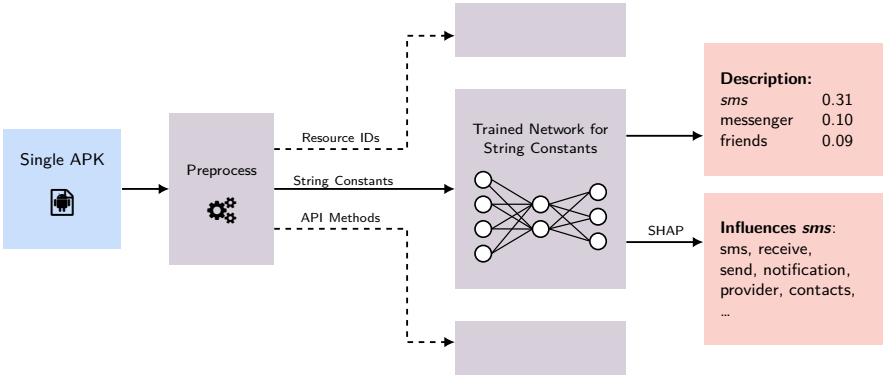


Fig. 1. Prediction of implemented app functionality using three dense neural networks.

4.1 Feature Preprocessing

Before training a neural network, it is essential to prepare the data for efficient learning. In the following, we cover the preprocessing steps that are applied to all developer-provided descriptions used in the training phase and the semantic input features processed by our networks after extracting them from app archives.

App Resource Identifiers. By parsing the XML files provided as resources in Android app archives, we obtain a list of identifiers consisting of alphanumeric characters and underscores. Unlike variable or function names in source code, identifiers are typically not obfuscated but stored in the way app vendors define them during development. The name, or identifier, usually reflects its purpose to some extent and can also give hints about the overall app. In practice, values are mostly made up of words or word combinations that are linked either by underscores or formatted via camel-case, e.g., *select_image_dialog*, *confirmRemove*, *pay_btn*, or *start_quiz_headline*. The challenge is therefore to decompose these values meaningfully in order to capture semantic relations. Without tokenization, e.g., it would not be possible to determine that the identifiers *select_image_dialog* and *select_video_dialog* imply similar actions that differ only in *image* and *video*. For a semantically more accurate representation, we split the words into smaller alphanumeric entities, i.e., *select*, *video*, *image*, *dialog* and link them as n-grams.

String Constants. Android, by design, allows apps to display UI elements in different languages. Therefore, vendors have to provide translations for all UI-related string values that are referenced by language-agnostic resource identifiers. In this work, we aim to infer keywords and short phrases in English only. To achieve this, we mimic the behavior of the Android operating system and try to match identifiers with constants by primarily searching them in language files that are supposed to include values in English, i.e., `values-en.xml` or `values-en-us.xml`. Only in the case of mismatch, we fallback to default definitions in `values.xml`. This simple resolution strategy ensures that the corpora of values subsequently trained in TF-IDF models consist mainly of English words.

After extracting all relevant string constants from an app, we iteratively decompose each value into substrings by splitting at non-alphanumeric characters, e.g., whitespaces, HTML tag brackets, dots, etc. While most resulting tokens are likely app-specific, others supposedly occur frequently across multiple apps. To estimate the relevance of individual tokens in relation to all apps, we use the tokens and their occurrence count to build a TF-IDF model. Thereby, we leverage the property of TF-IDF that rarely occurring and very frequent tokens are ignored to maintain a reasonable dictionary size. As a result, for each app, we obtain a TF-IDF vector that can be used as input for a neural network.

API Calls. Inspecting the call graph of Android apps enables us to identify and count invocations of Android APIs. We process the reverse-engineered source code of the app archive and build a call graph based on static, explicit code statements. We enrich the graph with additional edges by resolving inheritance relations and implicit data flows using EdgeMiner [1] by Cao et al. As Android

apps have no predefined entry points, *Activities*, *Services*, and *Providers* defined in the `AndroidManifest.xml` of each app are used as the starting point for modeling the call graph. This approach ensures that we capture only calls of API methods that implement an app's main functionality.

Our goal is to count execution paths, i.e., connections, between app entry methods $E_{in,j}$ and API call methods $E_{out,k}$. Therefore, we use the Dijkstra algorithm to check for each node $E_{in,j}$ whether there is a connection to $E_{out,k}$ in the graph. If so, we increment a counter for API call k. We count all methods as a combination of their (fully qualified) class and their method name.

As with resource identifiers and string constants, we create a TF-IDF model for API methods. For each app, we now have a list of pairs that consist of method calls and how often it was found in the app's source code. By using the TF-IDF algorithm, we decide which method names end up in the dictionary, based on their frequency. TF-IDF then transforms this information and returns a 1-dimensional decimal vector for each app sample that we can use as neural network input.

App Descriptions. As the output of our machine learning model, we want to infer feature-related parts of the app description. Intuitively, we use $n = (1, 2, 3)$ to cover phrases that include one, two, and three words. With stopword removal and Porter stemming, we reduce the number of frequent word combinations that are of comparably minor importance beforehand, e.g., *take some photos* and *take a photo* both become *take photo*. By stemming tokens, removing stopwords, and windowing with three different window sizes, we aim to capture more meaning.

The tokens, regardless of whether the model finds frequent single occurrences or combinations, are stored in their stemmed form. Stemming removes parts of the word to subsume multiple word variations and facilitate computation. It often does not, however, reduce the words to a stem that can be easily read by humans. Since the stemming transformation is not a bidirectional transformation due to the loss of information, an accurate *un-stemming* method cannot exist. Stemmed tokens contrast our goal to provide human-readable description fragments.

As a solution, we use a greedy algorithm to recover original words from their stem. Therefore, we keep track of all the stemming transformations, i.e., whenever a token T is altered and results in its stemmed version $\widehat{T} = f_{\text{stem}}(T)$. The suffix removal of stemming leads to $\widehat{T}$ having multiple corresponding original tokens T, so we collect the number of times of: $T \to \widehat{T}$. After processing all descriptions, we obtain an association count table that lists how often $\widehat{T}$ was caused by each original T. E.g., if the stemming result of a token is *locat*, it will be replaced with *location*, regardless of whether the original token was *location*, *located*, *locating*, or *locate*. By counting how often an original token results in a particular stemmed token, we can replace the stemmed token by its most common origin.

Ultimately, each description is represented by a list of tokens that we want to transform via a TF-IDF model. The model has features that consist of single tokens, 2-grams, and 3-grams. Stemmed tokens are re-transformed to their most likely original, non-stemmed word to be more easily readable afterwards. Hence, for each app description, we obtain a 1-d vector with normalized decimal numbers between 0 and 1, which we can subsequently use as machine learning targets.

4.2 Model Architecture

We propose a combination of three models of dense neural networks to predict keywords and short phrases that characterize a given Android application. Each model produces n-grams as output and receives TF-IDF vectors with either resource identifiers, string constants, or method names as input. In this section, we highlight the advantages of dense neural networks for our problem and present our network architecture regarding the set of chosen layers and hyperparameters.

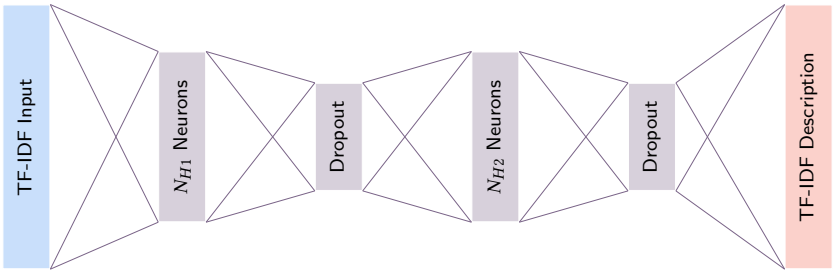


Fig. 2. Dense neural architecture to infer TF-IDF vectors with descriptive keywords from TF-IDF vectors of resource identifiers, string constants, and API calls.

Figure 2 illustrates our network architecture. The bag-of-words representation via TF-IDF vectors enforces positional constraints, i.e., the value for a particular word is always put into the same vector cell. This input property allows us to use a standard dense network structure in contrast to other convolutional or recurrent architectures that factor in positional and sequential information. Between dense layers, we apply dropout for regularization. As the output of each neuron consists of floating-point numbers, we have a regression task and use a linear activation function for the output layer and mean-squared error as a loss function.

The size of the input and output depend on the dictionary size of the TF-IDF models. Precisely, the dictionary sizes are limited by the minimum and the maximum document frequency, i.e., in how many apps a certain method call or resource identifier token occurs at all. Here, especially the lower boundary is crucial. If the minimum document frequency is too high, we miss information the neural network could use to infer the output more precisely. In case the minimum is too low, the model remembers too many tokens that rarely occur, and the dictionary becomes very large. The larger the input space, the more inputs and weight parameters are stored in memory, and the longer the training process takes. The selection of TF-IDF model parameters, thus, binds the training process. To find suitable network architectures, we used random search. For this non-exhaustive search, we trained networks with one to three hidden layers and 1,000 to 15,000 hidden neurons. Dropout was randomly set between 0% and 40%.

We choose the parameters empirically by trying different setups and observing the resulting dictionary sizes and model performances. Therefore, we set the minimum document frequency for each of the three input types to 2% of the total number of documents (apps), and the corresponding maximum frequency to 20%. This range means, e.g., if we have an app dataset of size N and a token occurs n

Table 1. Neural network configurations of the three models.

	Input TF-IDF # Features	Network Hidden Layers	Description TF-IDF # Features
Resource Identifiers	3315	2968, 3265, 1393 (3 Layers)	6140
String Constants	6391	2898, 3105 (2 Layers)	6140
API Methods	11735	5891 (1 Layer)	6140

times, it only ends up in the dictionary if it occurs in $0.02N \leq n \leq 0.2N$ apps. Table 1 lists our final network configurations and the TF-IDF dictionary sizes.

4.3 Model Training

The three models are trained using the mean-squared error measure as a loss function. As a performance metric, however, it is not fit for the purpose. Unfortunately, it does not give any intuitive expression of how well the model performs. Thus, we discretize the description TF-IDF vectors by choosing a threshold θ, above which we set the vector element to 1, or 0 otherwise. We can then use standard performance metrics like F-score, precision, and recall on these binary vectors to compare an actual description's vector with a description prediction.

Our correlation-based learning approach tries to find similarities between apps and thus neglects app-specific terms in the description. From a performance point of view, this means that we can expect a lower recall than precision. For early stopping, we are required to choose a pivotal performance metric that measures whether training should stop or continue. We, thus, use a weighted F-score ($\beta = 0.5$) that rates precision higher than recall. Consequently, we apply the $F_{0.5}$ performance for early stopping to find a good final training state.

4.4 Explaining Predictions

The essence of machine learning is finding patterns via function approximations in a given set of data. Due to the complex inner working of networks, it is not always obvious how predictions are derived. To find out which input items contribute to the prediction of keywords, we apply the model explainer SHAP [8].

SHAP is a method proposed to estimate the importance of sample features. The algorithm behind it is based on Shapley values, a concept in cooperative game theory: of n potential players, several combinations of $k \leq n$ players are possible, i.e., can play together against the bank. Each combination of players achieves a different (monetary) result. The Shapley value shows the contribution of each player by incorporating different combinations. Lundberg et al. used this concept in combination with additive feature attribution. By masking out several parts of the input features, different model results per sample are obtained. The results can be united according to Shapley, but this is computationally expensive. SHAP provides several approximations, e.g., one for neural networks called Deep SHAP. By leveraging knowledge about the network's parameters and structure, and not treating it as a black box, Deep SHAP creates a simpler, approximated model. In this work, we apply Deep SHAP on our neural network models.

5 Evaluation

The goal of this evaluation is twofold. First, we investigate the performance of our neural network with real-world Android apps. Second, applying our solution on a hand-picked set of applications, we compare predictions about the presumed functionality of apps with the actual description text from Google Play.

5.1 Dataset

We evaluate our approach using real-world applications from the PlayDrone dataset [13]. We opted for this repository of apps as it does not only feature raw app archives but also makes the vendor-provided app description available.

After downloading 115,294 Android apps and their corresponding metadata, we removed cross-platform apps as they implement their core functionality with web technologies and lack the corresponding resource identifiers, string constants, and API calls. From the remaining set of 85,915 apps, we filtered apps that had no descriptions in English language and ensured that preprocessing each description text resulted in at least 20 TF-IDF vectors. This boundary was set to reduce the potential impact of insignificant samples on the training process.

Table 2 highlights the final set of apps we used to train, validate, and test each network input feature. 20% of apps used for training are randomly picked to be also part of the validation set. This partitioning scheme is required to prevent overfitting of our machine learning model and to ensure meaningful predictions. The test set includes 1,000 randomly chosen apps that are not used during training. We build the set such that it only includes apps with a reasonably good description. Therefore, we make the simplifying assumption that apps with a higher download count tend to have higher description quality and, thus, prefer samples from comparably popular apps. We sort all apps in the dataset by download count and take every third app until we obtain 1,000 test samples.

5.2 Results

We trained neural networks for resource identifiers, string constants, and API calls, each with a set of 66,040 apps. To ensure an unbiased evaluation, the three models were validated using 20% of training data and tested individually with 1.5% of previously unseen data to confirm their final performance.

Table 2. Subsets of Android apps used as neural network input.

	# Apps
Android apps crawled	115,294
Cross-platform apps	29,379
English descriptions and $\geq$ 20 TF-IDF tokens	67,040
Training set	66,040
Validation set (20%)	13,208
Test set	1,000

Table 3. Performance on the test set of the three neural network input types via discretized TF-IDF vectors. Discretization threshold: $\theta = 0.05$.

	Resource Identifiers	**String Constants**	**API Calls**
Precision	79%	84%	69%
Recall	27%	19%	18%
$F_{0.5}$-Score	57%	50%	44%

The evaluation results on the test set are summarized in Table 3. The direct comparison of F-scores shows that resource identifiers yield the best results, while API calls perform significantly worse, with string constants in between. While precision values range between 69% and 84%, the recall column presents low values for all models. We attribute this mainly to two reasons. First, descriptions contain lots of words specific to the app that are hard to generalize. This makes reconstructing many of these rarely occurring words difficult. Second, the TF-IDF model for the description output does not take synonyms into account. E.g., if the description contains the word *image*, but the word *photo* is predicted, it counts as a mismatch and lowers the recall despite the semantic correctness.

In practice, these results mean that our trained neural networks can well predict keywords and short phrases that also occur in the developer-provided description. High precision and low recall imply that the rate of false negatives is higher than the rate of false positives. This is desirable in our setting because a lower false positive rate also produces fewer false attributions of app functionality.

5.3 Case Study

For a better understanding on the practical relevance of functionality predictions, in the following, we take a closer look at each model's output regarding two music-related apps that were not used during training. We visualize the top 8 predictions and relevance values via word clouds. The font size of each token is set with respect to the weight (relevance) the models assign to all outputs.

Figure 3 illustrates the top-ranked predictions of the three models for the music video streaming app *Vevo*. Apart from *video* being top-ranked, the nature of a video streaming and sharing platform is expressed by the phrases *tv show / tv channels*, *movies*, *subscription*, *music*, *live* and *content*. The tv-related phrases show that the models cannot distinguish between traditional television and online video streaming. As the predictions stem from many other apps, we reason that

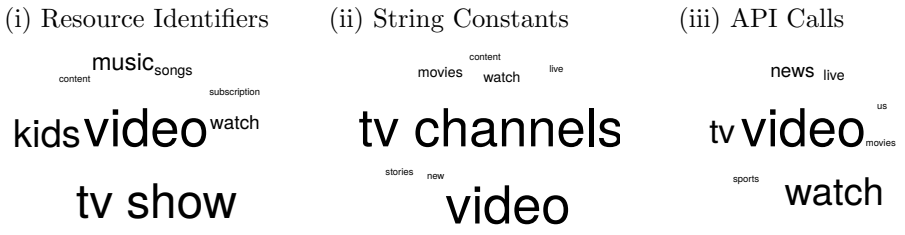
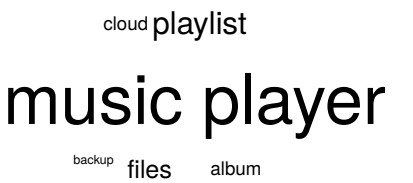


Fig. 3. Word clouds with each model's predictions for the video streaming app *Vevo*.

(i) Predicted Functionality



(ii) Actual Description

4shared Music was created for those, who can't live without music and don't want their attention to be attracted with anything else, but music while listening to it. Using your 'Search' menu item you can look for music files you like and add them to your playlist at 4shared Music. Moreover, you can upload tracks from your Android device to your 4shared Music. With 4shared Music you can enjoy 15GB of space for your music and nothing out of place. Upload and add all music files you like and make your life even more enjoyable with 4shared!

Fig. 4. Comparison of the real description of *4shared Music* and our models' predictions.

the neural networks understand the domain of the input and learn to cluster video-related applications internally. We also see that the inferred tokens based on API calls are much more general. The overall domain of the app becomes clear but, e.g., no n-grams, such as *tv channels* or *tv show* were learned. Overall, despite their independent reasoning, the three models each yield descriptive information and can correctly identify the app's main purpose.

The app *4shared Music* is a music player that accesses audio files stored on the cloud storage provider 4shared. In Figure 4, we contrast the developer-provided description with the summarized predictions of our three models. Our neural networks correctly found that the app is a *music player*, dealing with *playlists* and *albums*. They also identified the second domain of the app, the online storage platform, in terms of *cloud*, *backup*, and *files*. While all these tokens make sense, the actual app description text does not mention all of them, e.g., *player*, *cloud*, and *backup* are absent. In other words, since the description text does not cover these tokens literally, the measurable performance (see Section 5.2) decreases despite the good generalization. An accurate but abstracted word cloud that is intelligible to humans is, thus, difficult to measure.

5.4 Prediction Explanation

Each of our three machine learning models predicts a list of keywords and short phrases based on a given Android app archive. Apart from seeing this result, we also want to know which word predictions are caused by which input items. Therefore, we apply Deep SHAP (see Section 4.4) to all model predictions.

If, e.g., our resource identifier model outputs the word *dictionary*, we want to find the influences of these predictions. A reasonable, for humans understandable relation would be input tokens, such as *search*, *word*, or *translate*. Instead, in case meaningless tokens are predicted, the model would have learned this correlation as "noise" from similar apps but not from a particular app feature.

Table 4. SHAP algorithm applied on two predictions for the app *Slacker Radio*.

(i) Resource Identifiers

Description Token(s)	Input Tokens	SHAP
music player	artist	0.0122
	album	0.0107
	playlist	0.0071
	art	0.0034
	lyrics	0.0032

(ii) String Constants

Description Token(s)	Input Tokens	SHAP
music	playlist	0.0321
	song	0.0230
	stations	0.0125
	songs	0.0096
	tracks	0.0039

To assess network input-output relations, we take one sample and get the top prediction for it, i.e., we focus on the network's output with the highest numeric value. Then, we calculate the SHAP values for all inputs and list the corresponding input features and their SHAP values. Table 4 shows this result for the app *Slacker Radio*. The predicted keywords with the highest values were *music player* for resource identifiers and *music* for string constants. By looking at the top input influences, we can see that the two different network models make their decision based on reasonable inputs. These input tokens affect the output in a way that is easily comprehensible and verifiable by humans.

From applying SHAP to many samples, we noticed that for resource identifiers and string constants, found correlations are mostly self-evident. Although we also found many Android apps where the model based on API calls returned very accurate keywords, the associated SHAP values were not intuitively traceable. For instance, the *Vevo* app (see Figure 3) has *video* as its top predicted term. The associated SHAP values refer to generic methods belonging to the `Activity` class from the Android API that, by their design, are unspecific to multimedia apps. We assume that in such cases, implementations make use of a specific set of methods that are then considered as a sort of fingerprint to identify video-related app purposes. In other cases, SHAP explanations for API calls show very obvious correlations. E.g., for the keyword *shake*, we found the `SensorManager` class of the Android API among the closest-related input features. Overall, our qualitative analysis using the SHAP model explanation algorithm confirmed that all our models could very well outline the main purpose of most real-world applications.

6 Conclusion

In this work, we presented a solution to describe the main purpose of Android apps in natural language by analyzing resource identifiers, string constants, and API calls contained in app archives. Based on a combination of three dense neural networks, our approach accurately captures semantic relationships among apps. We carefully evaluated our approach on 67,040 real-world Android apps and showed that with a precision between 69% and 84% our neural networks could predict keywords and short phrases that also occur in the developer-provided description in Google Play. Our solution provides an effective method to describe the behavior of unknown app implementations.

References

1. Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., Chen, Y.: EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In: Network and Distributed System Security Symposium – NDSS'15. The Internet Society (2015)
2. Gao, H., Guo, C., Wu, Y., Dong, N., Hou, X., Xu, S., Xu, J.: AutoPer: Automatic Recommender for Runtime-Permission in Android Applications. In: 43rd IEEE Annual Computer Software and Applications Conference, COMPSAC 2019, Milwaukee, WI, USA, July 15-19, 2019, Volume 1. pp. 107–116. IEEE (2019)
3. Gorla, A., Tavecchia, I., Gross, F., Zeller, A.: Checking app behavior against app descriptions. In: International Conference on Software Engineering – ICSE'14. pp. 1025–1035. ACM (2014)
4. Hamedani, M.R., Shin, D., Lee, M., Cho, S., Hwang, C.: AndroClass: An Effective Method to Classify Android Applications by Applying Deep Neural Networks to Comprehensive Features. Wireless Communications and Mobile Computing **2018**, 1250359:1–1250359:21 (2018)
5. Karbab, E.B., Debbabi, M., Derhab, A., Mouheb, D.: MalDozer: Automatic framework for android malware detection using deep learning. Digital Investigation **24**, S48–S59 (2018)
6. Kowalczyk, E., Memon, A.M., Cohen, M.B.: Piecing together app behavior from multiple artifacts: A case study. In: Symposium on Software Reliability Engineering – ISSRE'15. pp. 438–449. IEEE Computer Society (2015)
7. Kuznetsov, K., Avdiienko, V., Gorla, A., Zeller, A.: Checking app user interfaces against app descriptions. In: Workshop on App Market Analytics – WAMA. pp. 1–7. ACM (2016)
8. Lundberg, S.M., Lee, S.: A Unified Approach to Interpreting Model Predictions. In: Neural Information Processing Systems – NIPS'17. pp. 4765–4774 (2017)
9. Pan, X., Cao, Y., Du, X., He, B., Fang, G., Shao, R., Chen, Y.: FlowCog: Context-aware Semantics Extraction and Analysis of Information Flow Leaks in Android Apps. In: USENIX Security'18. pp. 1669–1685. USENIX Association (2018)
10. Qu, Z., Rastogi, V., Zhang, X., Chen, Y., Zhu, T., Chen, Z.: AutoCog: Measuring the Description-to-permission Fidelity in Android Applications. In: Conference on Computer and Communications Security – CCS'14. pp. 1354–1365. ACM (2014)
11. Takahashi, T., Ban, T.: Android application analysis using machine learning techniques. In: AI in Cybersecurity, pp. 181–205. Springer (2019)
12. Vásquez, M.L., Holtzhauer, A., Poshyvanyk, D.: On automatically detecting similar Android apps. In: International Conference on Program Comprehension – ICPC'16. pp. 1–10. IEEE Computer Society (2016)
13. Viennot, N., Garcia, E., Nieh, J.: A measurement study of google play. In: Measurement and Modeling of Computer Systems – SIGMETRICS'14. pp. 221–233. ACM (2014)
14. Watanabe, T., Akiyama, M., Sakai, T., Mori, T.: Understanding the Inconsistencies between Text Descriptions and the Use of Privacy-sensitive Resources of Mobile Apps. In: Symposium On Usable Privacy and Security – SOUPS'15. pp. 241–255. USENIX Association (2015)
15. Zhang, M., Duan, Y., Feng, Q., Yin, H.: Towards Automatic Generation of Security-Centric Descriptions for Android Apps. In: Conference on Computer and Communications Security – CCS'15. pp. 518–529. ACM (2015)