International Journal of Information Security
https://doi.org/10.1007/s10207-022-00601-x

REGULAR CONTRIBUTION

Lib2Desc: automatic generation of security-centric Android app descriptions using third-party libraries

Beyza Cevik[1] · **Nur Altiparmak**[1] · **Murat Aksu**[1,2] · **Sevil Sen**[1]

Abstract
Android app developers are expected to specify the use of dangerous permissions in their app descriptions. The absence of such data indicates suspicious behavior. However, this is not always caused by the malicious intent of developers; it may be due to the lack of documentation of the third-party libraries they use. To fill this gap in the literature, this study aims to enrich application descriptions with security-centric information of third-party libraries. To automatically generate application definitions, the study explores classifying libraries and extracting code summaries of library methods that use dangerous permissions and/or leak data. Both the textual information of third-party libraries and their source code are used to create these definitions. To the best of our knowledge, this is the first approach in the literature that creates app descriptions based on third-party libraries.

Keywords Android security · Description-to-permission fidelity · Third-party libraries · NLP · NLG

1 Introduction

Researchers have been exploring the use of natural language processing (NLP) techniques in Android security. The central distribution of Android applications provides a platform for developers and users to share textual data about applications such as descriptions, user reviews, and ratings. In the last decade, this "metadata" has been used in Android security under four categories [1]: description-to-behavior fidelity, description generation, privacy, and malware detection.

Early studies aim to discover inconsistencies between dangerous permissions requested by an app and its metadata, particularly app descriptions. This problem, known as description-to-permission fidelity [2], is based on the assumption that the usage of dangerous permissions should be explained within the app description. In the literature, there are recent promising studies based on recurrent neural networks [3,4] and attention mechanism [5] that could automatically reveal such inconsistencies automatically. However, these studies have revealed low levels of performance on apps with short descriptions. These apps also decrease the accuracy of systems that categorize applications according to their metadata [6].

Another gap between permissions and descriptions results from the usage of third-party libraries (TPL) [7]. The literature has shown that some applications can use more than 20 TPLs [8], and on average more than 60% of application code consists of libraries [9]. In Android, since permissions are requested at the app level, TPLs may directly use these permission requests, although this may result in certain security concerns. It has been shown that the number of dangerous permissions in ad libraries has increased [10]. Moreover, some libraries fail to indicate their usage of such dangerous permissions in their documentation [11].

The current study proposes an approach called "Lib2Desc"in order to automatically generate app descriptions. Different from other studies in the literature [12,13], these approaches focus on generating descriptions related

✉ Beyza Cevik
beyzaccevik@gmail.com

Nur Altiparmak
nur.altiparmak@hacettepe.edu.tr

Murat Aksu
murat.aksu@bakircay.edu.tr

Sevil Sen
ssen@cs.hacettepe.edu.tr

[1] WISE Lab., Department of Computer Engineering, Hacettepe University, Ankara, Turkey

[2] Department of Computer Engineering, Izmir Bakircay University, Izmir, Turkey

Published online: 04 August 2022

to the usage of TPLs in the app. Here, we use both the library code and textual documentation in order to explain the security-sensitive usage of such libraries. First, we consider what the libraries do and to which categories they belong, and extract this information from their documentation. Then, we aim to explain the usage and leakage of sensitive data from libraries through the use of source code summarization techniques.

The proposed approach is aimed at assisting developers, users, and also market stores. First, it will help app/library developers to automatically generate app descriptions. Developers are often unaware of the permissions and data used by TPLs due to their focus on writing customized code as well as a lack of library documentation. Therefore, automating text generation based on the use of TPLs will be of help to developers. Depending on the libraries used, they may even add descriptions of the libraries used directly to their app descriptions and policies. Second, it will provide users with detailed app descriptions that can be scrutinized prior to installation and usage of an app. Lastly, although market stores do not create app descriptions, the proposed approach may be utilized by market stores to enhance developer-written descriptions.

This current study aims to deliver the following contributions:

- To the best of the authors' knowledge, this represents the first study that aims to comprehensively explain the use of sensitive data through TPLs, through the use of both the textual data and code of libraries.
- The study constructs a dataset[1] of TPLs, and then analyzes the usage of TPLs from a security perspective. As far as we know, the current literature offers no such dataset of TPLs. Currently, there are 2247 libraries, with 3% of them making use of dangerous permissions.
- It represents the first study to group TPLs into their categories in order to explain their usage in application descriptions. In this study, the effects of both the semantic information obtained from the libraries' documentation and the API usage in their code on the library categorization problem are analyzed.
- Source code summarization techniques based on neural models are explored to explain the usage reasons for dangerous permissions, and for data leakages associated with TPLs. This represents the first usage of such a technique to tackle the description generation problem in Android systems. The study proposes a new model based on transformers called "Code2Desc."Besides the proposal of a new model to address the problem, a successful model is proposed for the summarizing of Java source code fragments called "Neural Code Sum (NCS)"[14], which is adapted to the Android environment according to two different settings: fine-tuning the published NCS model with the introduced Android API dataset[1] that contains Android API code-explanation pairs, and training the model from scratch using only the Android API dataset. Each of these models are compared according to the BLEU score.
- The proposed models are also evaluated on a small dataset consisting of TPL methods that use dangerous permissions and leak data manually according to the following metrics: readability, correctness, expressiveness vulnerability, and the audience targeted (i.e., user or developer). Finally, examples of the generated descriptions are presented.

The remainder of the paper is organized as follows: Sect. 2 summarizes the related approaches in the literature under the following four subsections: description generation, analysis of TPLs, and source code summarization.Section 3 describes the steps to collect TPLs and Android apps to be used in further experiments. It also summarizes the analysis of collected TPLs from the security perspective. The proposed approach for the automatic generation of descriptions based on TPLs is introduced in detail in Sect. 4, presenting a detailed evaluation of the proposed approach and exemplar generated texts. The limitations of the approach are also discussed. Finally, Sect. 5 is devoted to presenting the concluding remarks of the study.

2 Related work

This section is divided into three subsections. In the first subsection, the existing studies on Android description generation are presented. Next, a summarization is provided of the studies published on the privacy analysis of TPLs, which mainly focuses on the detection of ad libraries since the classification of TPLs is one of the primary aims of the current study. The final subsection addresses studies regarding source code summarization techniques, since they are applied in the current study to explain methods that use dangerous permission and/or leak data.

2.1 Description generation

One of the first attempts to generate Android application descriptions was called "DescribeMe."[12]. Its aim being to generate security-centric Android App descriptions by combining static code analysis, subgraph mining, and NLG techniques. Static code analysis was used to extract behav-

[1] The datasets generated during and/or analyzed during the current study are available in the Lib2Desc repository, [https://wise.cs.hacettepe.edu.tr/projects/desre/Lib2Desc/].

ior graphs from Android source code, with subgraph mining techniques are applied to reduce complex graph sizes and to discover particular security behavior patterns. Then, each vertex of the constructed graphs was traversed so as to generate a natural language sentence corresponding to these paths. DescribeMe [12] can effectively generate informative descriptions for some but not all permissions; this is because some permissions cannot be associated with API calls and their parameters. With a different perspective, CLAP [15] concentrates on recommending a list of potential permission requirements for the permission explanation. It is based on finding similar apps according to information such as title, description, category, and permissions and then extracts sentences from the descriptions of the similar apps identified.

PERSCRIPTION [16] generates personalized security-centric descriptions by using statistical natural language generation techniques. PERSCRIPTION [16] learns users' linguistic preferences and concerns through their behavior, such as denial of specific permission, and then identifies that permission usage as a concern for the user. Users are categorized according to their behavior based on the Big Five Personality model [17]. Users' linguistic preferences such as frequently used word types are then analyzed in order to generate user-specific descriptions. Then, different syntactic templates for different personality traits according to the findings are constructed based on statistical natural language generation.

AutoPPG [18] focuses on creating readable and correct privacy policy descriptions regarding the personal privacy information-related behavior of Android applications. As far as we know, this is the only study that has used information collected about TPLs; however, only data leaked by TPLs were actually presented. Another work [19] examined human-authored reports written by malware analysts in which text mining and machine learning techniques were applied to the reports in order to explain the malicious behavior in natural language First, a feature set was constructed from negative (unexpected behavior) and positive (normal behavior) weights extracted from the L-1 Regularized Logistic Regression model [20] that is developed for malware detection. Each application is represented by API calls, actions, and permissions, and then passed to the model. An algorithm based on the beam search technique was designed to extract a list of keywords indicative of malware to locate descriptions including keywords from the list, and then to rank them according to the cosine similarity between the TF-IDF vectors of sentences. Consequently, it automatically extracts promising description sentences.

2.2 Analysis of third-party libraries

In the literature, some studies analyze TPLs from the security perspective. One of the earliest studies in the literature called "AdRisk"[21] analyzed a total of 100 ad libraries and showed that they could perform risky behaviors from sending sensitive information to allowing non-trusted dynamically loaded code to be executed. Some ad libraries use dangerous permissions provided by applications, but fail to document them [11]. In that study, 13 libraries were analyzed in detail and it was shown that some libraries could access and send private information. Moreover, these data are sent in clear text in each of these libraries under analysis other than Youmi. Lastly, it is shown that four of them could run external code, hence becoming susceptible to attack. Permissions requested by ad libraries over time were investigated in [10], and it was shown that an increasing number of both permissions and dangerous permissions were requested by the ad libraries. Moreover, it was pointed out that the usage of dangerous permissions was less common among the more popular libraries.

While the analysis studies on libraries have mainly been based on static analysis, a recent study based on dynamic analysis showed that many TPLs access private information, and even leak such confidential data which therefore presents a potential security risk [22]. In their study, they classified data leakage into four cases, with particular focus on cases where the data leakage was directly caused by the TPLs.

In CHABADA [23], applications were clustered based on app descriptions. Then, sensitive APIs used by each cluster were determined, and deviations from the behavior of clusters then identified using OC-SVM. In [24], a similar outlier detection approach was applied, but for the formation of each cluster's behavior, only sensitive APIs used by custom code were used. They underlined that apps do not provide information regarding TPL functionality in their description; hence, the use of TPLs could affect outlier detection. The results showed that some deviations found by CHABADA no longer existed when TPLs were removed from the custom code. This result underlines the necessity for information regarding TPLs to be present in app descriptions.

Since the category of a library could provide information about its functionality and the permissions it may require, such category information could be added to the description of apps that use that library. However, there are only a few studies [25,26] that have classified TPLs, and differently from the current study too, and have also only classified ad libraries. AdDetect [25] first extracts the modules of an app according to the decoupling technique, which is based on hierarchical clustering. This process consists of three steps; extracting the package hierarchy of the app and identifying the representative packages, estimating the dependencies among these representative packages and the construction of a Package Dependency Graph (PDG), and then applying hierarchical clustering to recover the individual modules. In total, 576 semantic features were collected from each module and given as inputs to the SVM classifier. These semantic features were then divided into three groups. The first group extracts

information related to the usage of Android components, device-specific and user profile information such as the usage of IMEI, whereas the second group obtains the permissions' usages, and the third group covers the features corresponding to the usage of selected API calls that provide access to critical system resources. The experimental results showed that AdDetect was able to detect ad libraries with 95.34% accuracy and was proven to be robust to obfuscation. PEDAL [26] also detects ad libraries very effectively by extracting features from applications bytecode. A total of 128 features were extracted related to the usage of Android components, the usage of selected Android permissions, the usage of visual elements since they are frequently used to display ads in ad libraries, the usage of information sources and sink, and the usage of APIs for runtime permission check. Whereas each of these studies only detect ad libraries, the current study aims to classify libraries by using features extracted both from the libraries' metadata and code. In other words, we generate a model for multiclass library classification. Another important difference is that the features are extracted not only from the code, but also from the metadata, and also that they are text-based.

2.3 Source code summarization

As in many fields, artificial intelligence-based solutions have been proposed in recent years to address numerous software engineering problems [27]. One such area is the creation of natural language descriptions for a given piece of code [28], which is defined as source code summarization, which focuses on briefly and clearly describing the behavior of software code fragments. Automatic source code summarization has therefore been investigated by researchers as it helps to comprehend source code, and also has extensive application domains such as automatic documentation generation, source code captioning, and code commenting.

In recent years, neural models have been widely used to address this problem as they can develop solutions that require less human effort than template-based methods. This sequence-to-sequence (Seq2Seq) learning problem performs translations from one language to another, essentially performing a programming language to natural language conversion. However, using only the words sourced from software fragments is insufficient to adequately describe what a piece of actually does. The structure of the language must therefore also be considered in order to understanding a program's behavior. As such, words and symbols that describe the behavior of a program, for example its control and data flow, are also utilized in such models [29].

One of the first essential studies in the literature [30] extracted high-level events related to a method by using the control flowchart and abstract syntax tree (AST) of that code. For this purpose, they utilized the SWUM (Software Word Usage) model. In a later study [31], PageRank and SWUM methods were used together in order to extract phrases from Java source code and to generate natural language expressions of methods. The most important method calls were extracted using PageRank, and then the keywords to be used in the natural language representation of the method were extracted from these calls using SWUM.

Recently, neural network approaches for automatic source code summarization have become very popular, due in part to the advances made in deep learning and the availability of high-volume data. Most of the recent approaches [32,33] adopt the sequence-to-sequence (Seq2Seq) neural machine translation architecture proposed to date in numerous natural language processing applications [34]. The Seq2Seq network architecture uses an encoder-decoder structure composed of RNNs, LSTM, or GRUs. While RNN and LSTM-based architectures are much preferred in these models, transformer-based architectures [14,35] have been proposed only very recently.

Neural Code Sum (NCS) [14] is a state-of-the-art transformer-based approach that has been very recently proposed to address the problem. As in NCS [14], most other studies [29,32,35] use source code written in programming languages such as Java and Python that support multiple programming paradigms (e.g. object-oriented, scripting). The comparisons carried out in [14] show that the full model version of NCS, which uses relative position representations and a copy-attention mechanism that allows the model to copy from input source code, outperforms the other models by a significant margin (a BLEU score of 44.58). However, a recent study [36] that deeply analyzed neural code summarization models in the literature showed that various factors could affect the performance of models such as data preprocessing steps, selection of BLEU metrics (sentence level, corpus level, smoothing method, etc.), dataset splitting approach, and dataset characteristics.

Since API calls are one of the most important features in a piece of code, the knowledge of a model generated for matching API calls with their call definitions was transferred so as to solve code summarization in [37]. It was shown that the performance of code summarization increased with the transfer of API call knowledge. A study examining human-generated code summaries [38] showed that as the detail in code summaries increases, so does the use of keywords related to API calls. According to the study, while high-level code summaries use keywords related to APIs at a rate of 76.78%, this increases to 93.75% in low-level (more detailed) code summaries [38]. Hence, some neural models use a method's API calls to extract a summary [38]. Instead of using the names of API calls, some models use the documentation (description) of those calls together with the code and AST tree. [39]. However, it was shown that the models' performance decreased when the number of API calls used

increased due to the long definitions that originated from API calls.

In the literature, there have been limited approaches to summarizing the source code of programs developed utilizing the event-driven programming paradigm, in which the program flow is controlled by user actions named events (e.g., keypress, mouse clicks). In other words, while previous studies have worked on large datasets generated from Java methods and their definitions, the current study aims to expand upon these approaches in order to explain Android application methods. For this purpose, Android API calls and their documentations are utilized. This study is different from other studies in the literature in that it aims to generate code summaries of Android source code developed in Java language that support the event-driven architecture of Android OS. In addition, the study aims to provide a simple explanation for suspicious behavior in code fragments from the security perspective. In the future, this approach could be further improved with an extended dataset that includes explanations, particularly for suspicious and malicious behaviors.

3 Data collection

3.1 Collecting third-party libraries

A total of 2247 TPLs were collected from Android Arsenal [40], Maven Central [41], Spring.io [42], JCenter [43] and Jitpack [44], which are all repositories commonly used by the Android developer community. These repositories each contain various information about TPLs such as their source code, API documentation, general descriptions, and categories. However, only Android Arsenal contains information about the categories of TPLs that are shown as tags selected by developers; therefore, the categorization of TPLs was conducted semi-automatically in the study. First, the categories listed in Table 1 were determined based on the functionalities of TPLs presented in their descriptions and tags. In this step, the most discriminate words of each category were also extracted. Then, the keyword-based approach was applied to group the TPLs into their categories. Although this is a somewhat simplified approach, it is applicable to most TPLs in the dataset of the current study. For example, all TPLs in the PERMISSION category include the keyword "permission"in their description. Similarly, all TPLs in the DATE & TIME PICKERS category include at least one of the following keywords: "date,""time," "calendar,""range,"or "picker."Following this initial filtering, three authors of the current study manually verified the TPLs' categories. In some cases, where the descriptions were not found to be that informative, the code and the TPL explanations shared on Github were analyzed. The most significant issue was that the categories were not homogeneously distributed, so, whilst there were a large number of libraries in some categories, there were insufficient numbers of libraries to work on in other categories. The homogeneous distribution of data is important, especially for the multiclass classification of libraries. Therefore, the sample for categories with inadequate volume were increased by applying the same automatic keyword-based filtering and manual verification steps.

The dataset used also contains the source code of the libraries; however, the source code of some libraries downloaded from the repositories may be empty. Moreover, in some cases, the decompilation of code in the libraries might fail. Therefore, static analysis was applied to 87% of all the libraries in the dataset. The source code of the libraries downloaded from these repositories was generally not obfuscated contrary to the code obtained from the decompilation of applications that use the same libraries. This allowed us to analyze the libraries' code statically, and only a small number of the libraries in the dataset were found to be partly obfuscated. While only 88% of libraries include a general explanation, 85% included category information. The categories of the libraries in the dataset are listed in Table 1.

Overall, 16% of the libraries in the dataset use permissions, and 3% use dangerous permissions. While the most frequent permission found in the libraries was ACCESS_NETWORK_STATE, the most used dangerous permission was ACCESS_COARSE_LOCATION. The permissions used by TPLs were extracted using Axplorer [45]. Although it may be said that Axplorer has certain limitations such as supporting up to Android API Level 25, it has the most up-to-date Android API permissions mappings in the current literature. More information about the dangerous permission usage of libraries is provided in Table 2. Most of the libraries (approx. 63%) with LOCATION permission were found to be in the LOCATION category. Another category that uses this permission, although in only about 14% of cases, was the UTILS category. Libraries in other categories were also found to use this permission, albeit rarely. The READ_PHONE_STATE permission was found to be used by libraries belonging to various different categories in our dataset.

3.2 Collecting Android applications

TPLs form an extensive part of an APK (Android Package Kit). In the literature, it is shown that, on average, more than 60% of application code belongs to TPLs [9]. However, the usage of TPLs yields security risks due to the potential malicious use of dangerous permissions requested by an application. In addition, it has been shown that they could leak sensitive data [11,21]. Therefore, the usage of dangerous permissions and data leakage by TPLs were analyzed as part of the current study. In addition to the library dataset,

Table 1 Categories of TPLs

Categories	# of samples
UI	524
UTILS	206
CAMERA	133
LOCATION	107
AUDIO	87
MESSAGE	82
DEBUG	74
DATE & TIME PICKERS	69
BLUETOOTH	68
ANALYTICS	65
ADVERTISEMENTS	53
NETWORK	51
DATABASE	48
TOOLS	43
PERMISSIONS	41
PURCHASE	40
SECURITY	36
SENSOR	33
FILE_SYSTEM	30
STORAGE	26
MEDIA	23
TEXTTOSPEECH	19
TESTING	12
LOCALIZATION	8
MAIL	7
CLOUD STORAGES	6
USB	5
FACE RECOGNITION	4

a total of 7152 applications were also collected from the official Android market in order to analyze how they use TPLs from a security perspective. Since the most frequently encountered dangerous permission used by applications and libraries is LOCATION, applications that use these permissions are given priority to download.

In the literature, several approaches have been used for TPL detection. The white list approach, in which a list of libraries is constructed and the package names in applications are searched against this list, is not considered resilient to package name obfuscation. Hence, new approaches for the automatic detection of libraries have been proposed in recent years. The most popular and recent TPL detection tools are LibRadar [46], LibD [47], LibPecker [48], LibScout [49], LibID [50]. In the current study, LibRadar was selected for TPL detection due to its considerably low detection time when compared to other tools. LibRadar employs static analysis and uses library-specific features such as the number of API calls in its TPL detection.

First, LibRadar was executed on the application dataset. According to the output, each application was shown to use seven TPLs on average. Second, applications were then decompiled. Libraries using dangerous permissions in the dataset were extracted using Axplorer [51], which matches API calls with the permissions used. The results showed that 77% ($n = 5.562$) of applications used dangerous permissions through TPLs, and that 83% of these applications declared that dangerous permissions were used by libraries in their manifest files. The most used permission in our dataset was INTERNET, and the most used dangerous permission was established to be ACCESS_FINE_LOCATION as expected. This permission is mostly used by the "android.location. LocationManager. getLastKnownLocation"API call, which returns the last known location of a device. More information on the uses of dangerous permissions is provided in Table 3.

Lastly, the libraries in the dataset that leak data were extracted using FlowDroid [52], which is a static analysis tool that analyzes data flow in Android and Java programs. In total, 61 unique libraries were found to leak data. The majority of the leaked data came from network usage, with network information found to be the data most leaked. Table 4 and Table 5 present more information on the methods used for leaking and data leaked by TPLs, respectively.

4 Lib2Desc

The proposed approach, Lib2Desc, consists of three components as shown in Fig. 1. First, where available, the general information about a given library (e.g., its purpose)

Table 2 Dangerous permissions used by TPLs

Permission	# of libraries	Most Used API Call
ACCESS_COARSE_LOCATION	47	android.location.LocationManager.getLastKnownLocation
ACCESS_FINE_LOCATION	46	android.location.LocationManager.getLastKnownLocation
READ_PHONE_STATE	5	android.telephony.TelephonyManager.listen
SEND_SMS	1	android.telephony.SmsManager.sendTextMessage
READ_EXTERNAL_STORAGE	1	android.telephony.SmsManager.sendTextMessage

Table 3 Dangerous permissions used through TPLs in apps

Permission	# of applications	Most used API Call
ACCESS_COARSE_LOCATION	5529	android.location.LocationManager.getLastKnownLocation
ACCESS_FINE_LOCATION	5401	android.location.LocationManager.getLastKnownLocation
READ_PHONE_STATE	1636	android.telephony.TelephonyManager.listen
SEND_SMS	211	android.telephony.SmsManager.sendTextMessage
READ_EXTERNAL_STORAGE.	211	android.telephony.SmsManager.sendTextMessage
ACCESS_LOCATION_EXTRA_COMMANDS	57	android.location.LocationManager.sendExtraCommand
USE_SIP	24	android.net.sip.SipManager.open
RECEIVE_MMS	7	android.telephony.SmsManager.downloadMultimediaMessage

Table 4 Leaking methods used by TPLs

Leakage type	# of libraries
NETWORK	47
FILE	2
NO_CATEGORY	5
UNKNOWN	7

Table 5 Data leaked by TPLs

Leaked data	# of libraries
NETWORK_INFORMATION	54
LOCATION_INFORMATION	2
CALENDAR_INFORMATION	4
ACCOUNT_INFORMATION	1

is extracted from its documentation. Then, the second component aims to automatically identify the category of the library, whilst the final component creates a summary of library methods using dangerous permissions and/or that leak data. The primary combined purpose of these components is to provide the user with an explanation as to why TPLs use dangerous permissions or leak data. While the most fine-grained information is obtainable from the third component, it may not perform that well if the method is exposed to significant obfuscation. Even in such cases, the aim is to elicit some basic information about the library and its category through the first two components in order that the users can, at the very least, evaluate whether or not the requested permission or data leakage is considered compatible with the general purpose and category of the library in question.

4.1 Extraction of general information

This component aims to assess the quality of the collected dataset, and then to extract the general information of the libraries. The BERT (Bidirectional Encoder Representations from Transformers) framework is a commonly used solution applied for a variety of tasks in NLP, such as question answering, sentiment analysis, and language inference. Here, BERT [53] models were employed for a question-answer task, with the focus being on answering questions asked within a certain context using natural language.

First, three predefined questions were determined with the aim to extract information from the textual description of TPLs (Q1: "What is this?", Q2: "What is the purpose?", Q3: "What does it do?"). Then, these three questions, together with the preprocessed descriptions of 450 TPLs, which form the context for the questions, were fed into the model to extract information from the textual description as output. In the preprocessing of the TPL descriptions, fundamental operations such as removing code pieces, eliminating non-Unicode characters were applied. For the current study, the BertForQuestionAnswering extractive question-answer model from the Transformers package, as developed by Hugging Face [54], was employed as it is known to provide state-of-the-art NLP models. Finally, the model's output, as in answers to the aforementioned questions, were analyzed in detail.

Each library's textual description was then automatically extracted from their websites, and then analyzed in terms of content before being classified as insufficient/noisy or sufficient data. Overall, 82% of the descriptions were found to be sufficient. Next, the validity of the answers produced by the model to the three questions were checked and separately tagged as valid or invalid for both all descriptions and for those with only sufficient data (see Table 7). The overall content sufficiency and textual quality of library data was then tagged as valid where at least one of the questions was answered correctly. In the context of description generation or library classification, even answering one of these questions may constitute important or descriptive information. Table 7 presents the number of valid tags for each question, as well as the total number of valid answers, in which the TPL descriptions have at least one valid answer. The results reveal that 92% of libraries with sufficient textual description contained at least one valuable item of information. These

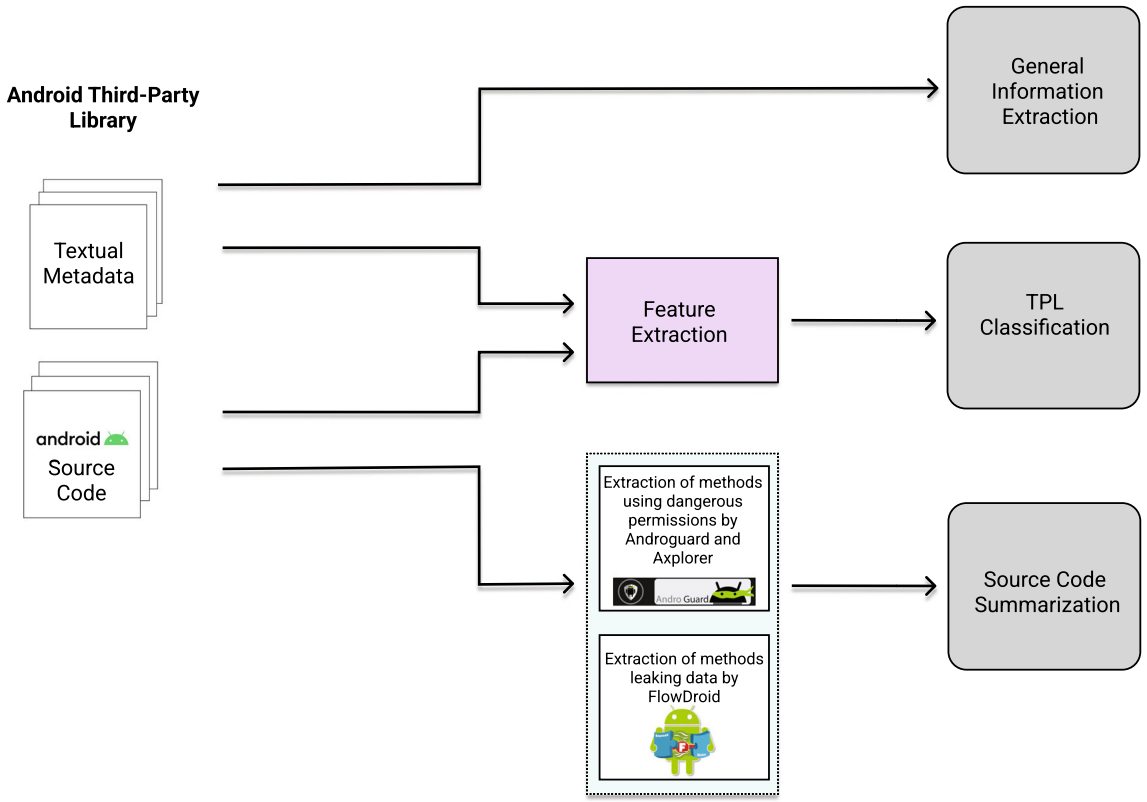


Fig. 1 Overview of the proposed approach

results and their statistics provide insight into the quality of the dataset, and thereby its potential application through revealing whether or not the collected data is deemed adequate to resolve tasks such as library classification. A few example answers produced by the model for each question are presented in Table 6.

Table 6 Exemplar answers generated by BERT

Noise FFT	
Q1:What is this?	A FFT computation library for Android
Q2:What is the purpose?	to be easy to use
Q3:What does it do?	compute FFT
Soundify	
Q1:What is this?	Soundify Library
Q2:What is the purpose?	Offer to you the data transmission via sound waves
Q3:What does it do?	Makes the transformation and transmission
	Of certain information into sound waves
Android playlist file parser	
Q1:What is this?	Android Playlist File Parser
Q2:What is the purpose?	To parse a playlist file provided as an InputStream
Q3:What does it do?	Please check the format's

4.2 Classification of third-party libraries

The radical increase in mobile applications in recent years has led to significant interest in Android application development. Developers and companies have therefore released numerous TPLs for application developers to use. These libraries are widely used as they make the application development process both easy and quick to use. As with apps, libraries are categorized into different groups based on their functionality, such as advertising or media. Automatically classifying these categories will help to promote better understanding as to why some libraries use dangerous permissions or leak data, and to help identify suspicious behavior for a given category. This constitutes the primary aim of this component.

Developers provide textual descriptions to explain the functionality, purpose, and usage capability of libraries, whilst the textual metadata reveals similarities between libraries with similar functionality. Therefore, the current study aimed to automatically investigate the categorization of libraries by utilizing the textual metadata provided using machine learning techniques. The effect of different feature

Table 7 The accuracy of answers produced by BERT

Accuracy	All data	Sufficient data
Q1	0.60	0.65
Q2	0.62	0.71
Q3	0.64	0.75
Overall accuracy	0.86	0.92

groups on the classification of libraries was then analyzed. Few studies in the literature [25,26] have applied machine learning techniques for the classification of Android libraries, and primarily these studies have targeted the identification of ad libraries that are used for displaying advertisements within Android applications, for monetizing applications, and for the provision of user-specific advertisements. The literature has reported that these types of libraries access sensitive information such as users' current location and e-mail addresses, therefore, it may be said that such studies show a level of parallelism with the current study in terms of the classification of libraries for Android security. However, the current study differs in that it approaches the task as a multiclass classification problem. Another difference is that our approach utilizes not only features extracted from the code, but also textual features obtained from the metadata. In the current study, several feature groups were extracted, and their effect on the performance of the proposed solution then evaluated. Further explanations regarding these features are provided in the subsequent section.

4.2.1 Data preparation

Data were collected from the libraries listed in Sect. 3.1, from which a total of 29 different categories are represented. However, not all of the categories were able to provide a sufficient volume sample for the purposes of classification due to the semi-automated labeling of TPL categories. Having a balanced and significantly large representative dataset is considered important for any machine learning method. In order to achieve a balanced dataset in which each category offers a sufficient sample size, some of the lesser used categories were excluded. The intended objective here was both to try and achieve a sufficient sample size for each category, and also to have enough categories overall to successfully perform multiclass classification as the primary purpose. Based on this tradeoff, a total of 10 categories were used, with each consisting of at least 60 samples, as can be seen in Table 8. The illustration in Fig. 3 further supports this approach by showing how the sample size plays a key role in classification performance. Whilst 80% of the selected samples were used for the purpose of training, the remaining 20% were used for testing.

Table 8 Categories used in TPL classification

Category	Sample size
UI	524
UTILS	206
CAMERA	133
LOCATION	107
AUDIO	87
MESSAGE	82
DATE & TIME PICKERS	69
BLUETOOTH	68
ANALYTICS	65
DEBUG	74

The current study used both TPL descriptions and code in order to classify and categorize the TPLs. As such, certain preprocessing steps were applied to both the TPL descriptions and code. First, the library explanations were separated into distinct words, and then grouped and evaluated according to different inflectional word forms using the classical NLP method known as lemmatization. During this process of simplification, the NLTK [55] library was employed, which is known to be widely used in NLP tasks. Then, stopwords were eliminated by using NLTK. To clarify, both lemmatization and stopword elimination are frequently applied tasks in the field of NLP as they simplify the textual data entering the classifier and thereby help to eliminate noisy data that negatively affects machine learning models. API calls were also extracted from the TPL code. These API calls were represented in two different ways: textually and encoded as 0 or 1, in order to indicate the existence of a particular API call in the code. API calls in the form of textual data were divided into keywords according to pascal-case spelling style rules, which is a typing style paradigm frequently employed by developers. An exemplar keyword extraction is provided in Table 10. The API call vector was then encoded as 0 or 1, according to whether or not the library makes the call. The feature groups used in the current study are listed as shown in Table 9.

4.2.2 Experiments and results

In the study's experiments, the LinearSVC classification algorithm from the Scikit-learn library [56] was employed as the classifier. As previously stated, a total of 10 categories with more than 60 samples were used to build the classifier. The experimental results were evaluated according to accuracy, precision, sensitivity, and f-score values. The results of the experiments performed with each different feature group are presented in Table 11.

Table 9 Feature groups used in the classification of TPLs

Feature group	Components	Vector
F1	Library descriptions and extracted keywords from API Calls	TF-IDF vector of concatenated components
F2	Library descriptions	TF-IDF vector
F3	Extracted keywords from API calls	TF-IDF vector
F4	API calls	One-hot encoded vector

Table 10 Keywords extraction from API calls

Original API call text	Extracted features
GetAction ()Ljava/lang/String;	Get, Action
GetColor (I)I	Get, Color

Table 11 TPL classification results

Feature group (%)	Accuracy (%)	Precision (%)	Recall (%)	F-Score (%)
F1	74	75	74	73
F2	75	76	74	75
F3	55	54	55	53
F4	56	57	56	56
F1 and F4	60	60	60	59
F2 and F4	60	60	60	59
F3 and F4	57	57	57	56

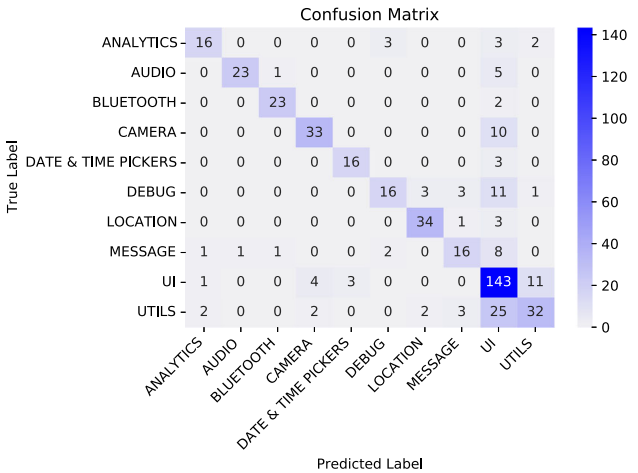


Fig. 2 Confusion matrix of TPL classification

The experimental results show that the most successful model was obtained using only the features extracted from the library descriptions, which have a more complex structure than API call vectors, and are represented as one-hot encoded or textual form. Moreover, they perform better at describing a library's functionality. Although feature vectors consisting of textual or one-hot encoded API calls provide information about library categories, combining the library descriptions with API calls adds noise to the input vector. In other words, as these features are in different languages, with library descriptions considered natural language, whilst API calls are formal language (i.e., code), this can negatively affect the understanding of the model. As a result, TPL descriptions produce a better result when used singularly in order to resolve the library classification problem.

The results showed that the model using encoded API call vectors was proven as superior to the model using the text form API call vectors. This result is due to the encoded vector containing a greater level of information since it has more dimensions than the textual vector. Moreover, the model using a concatenation of these two vectors as input was shown to outperform models using only API call representations. In other words, although the representation of API calls is different, they are unaffected by the aforementioned noise problem since both vectors are sourced from the same formal language space and have more dimensions than vectors that use only a single API call representation.

In order to see the relationships between libraries, the confusion matrix of the best model is presented as Fig. 2. The following categories were found to be successfully predicted by the model: BLUETOOTH, LOCATION, AUDIO, and DATE & TIME PICKERS. However, the model confused the categories of ANALYTICS, DEBUG, CAMERA, MESSAGE, UTILS, and UI with each other. As such, the model mislabeled the ANALYTICS category as DEBUG, and the CAMERA, MESSAGE, and UTILS categories as UI. The ANALYTICS and DEBUG categories, by their very nature, contain common functionality such as analysis and monitoring and also some that may be linked to the UI category. For instance, both camera and SMS functions are accessible via the user interface (UI), so it is therefore possible for these categories to have some of the same functionality as the UI category. The UTILS category is very general and may therefore also contain functionality related to the UI.

Finally, the effects of the sample size on the experimental results were analyzed using 20, 30, and 40 samples selected from each category, respectively, and the models were each trained with these samples and evaluated using the same test set. The performance results of these tests are provided in Fig. 3. As can be seen from the results, the training size significantly affected the performance of the model, which was an expected result. Therefore, categories with samples exceeding 60 were used in the experiments. It is also believed that the results would improve significantly by increasing the training set size further still.

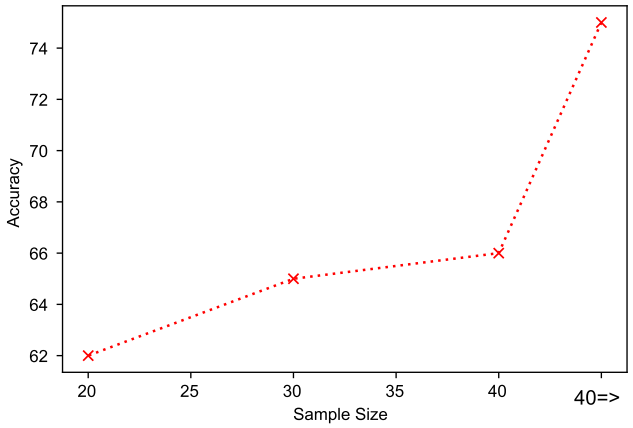


Fig. 3 Experimentation conducted on a training set of 20, 30, and 40 samples for 10 categories. Experiments use F2 as their input vector

4.3 Source code summarization

In the literature, TPLs are shown to access and collect private information by using static [57] and dynamic [22] analysis techniques. It has also been shown recently that many TPLs have access to private information, and therefore could even pose a security risk if they were to leak confidential data [22]. Since such suspicious behaviors are not declared in TPL documentation [57], it is important to extract this type of security-critical information and to include within the apps' descriptions that use such libraries. The main aim of this component is to resolve or adapt this problem, which is described as source code summarization in the literature, to the Android environment.

As in many other fields, machine learning approaches have gained significant importance with the increases seen in big data and the advancements in both neural models and computational resources. Recent approaches [32,33] to source code summarization have utilized sequence-to-sequence (Seq2Seq) neural networks [34], which take an input sequence (words, letters, time series, etc.) and then output another sequence. The encoder captures the input sequence and encapsulates the information into a context vector. Then, the decoder generates the output sequence based on the received context vector. Models based on the Seq2Seq architecture have been shown to achieve a high rate of success in various problems such as text summarization, machine translation, and grammatical error correction.

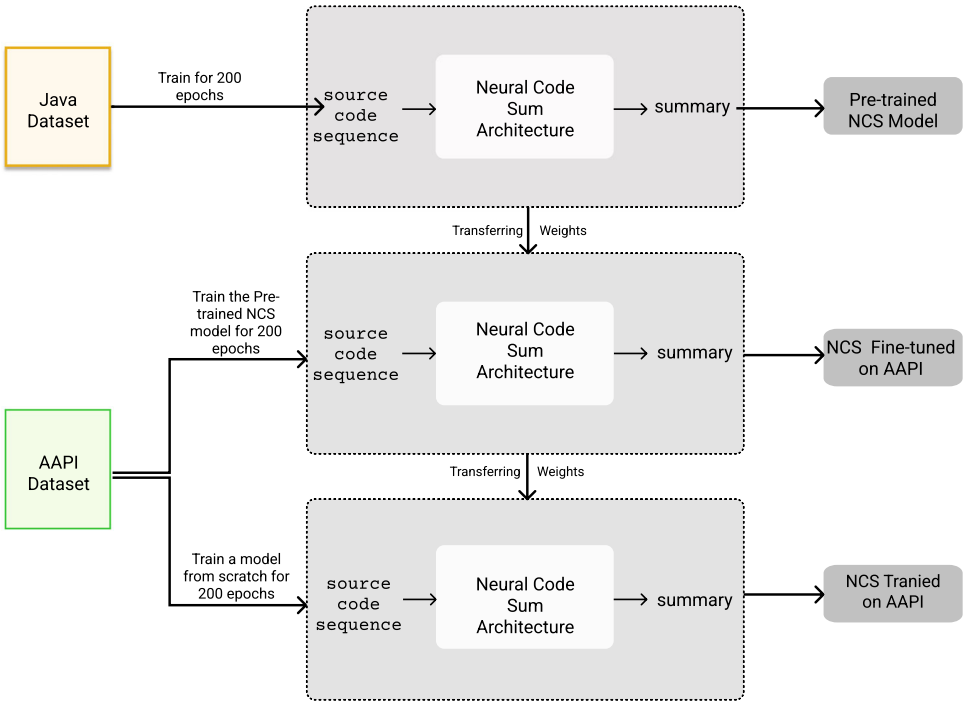
In the RNN-based Seq2Seq models, the encoder requires two forms of input: the current input sequence and the representation of the previous sequence. Thus, the output depends on both the current input and also that of the previous input. The model helps to preserve the sequential information in the hidden state and to make use of it in the subsequent prediction. Although Seq2Seq models significantly improve machine translation tasks compared to traditional machine translation approaches, this underlying mechanism of RNN-based Seq2Seq models does present certain limitations: (1) neural networks become deeper as the sequence gets longer, which can cause a vanishing gradient problem, and (2) RNNs cannot preserve long-range dependencies in long sequences as they can only recall the memory from one previous step. Transformers have become very popular since they can overcome these limitations associated with RNNs. Therefore, two transformer-based Seq2Seq approaches were employed in the current study's model to tackle the code summarization problem. In the first approach, the state-of-the-art transformer-based approach called neural code sum (NCS) [14] was adopted, and second, a new approach called Code2Desc is introduced.

4.3.1 NCS

NCS [14] is a transformer-based model with an attention mechanism, which has been shown to improve source code summarization performance. Attention is a mechanism used to strengthen certain features and weaken others and is applied to the feature vector during the parallelization of vector operations. In addition to the self-attention mechanism, NCS [14] uses a copy-attention mechanism in order to enable Transformer [58] to use a copy of an input vocabulary in the output sequence.

As in NCS [14], most of the studies in the literature use source code written in programming languages such as Java and Python, and therefore support multiple programming paradigms such as object-oriented and scripting programming language. However, few approaches actually summarize the source code of programs developed utilizing the event-driven programming paradigm in which the program flow is controlled by user actions (e.g., keypress, mouse clicks) named events. Since Android uses an event-based programming language, the NCS model is used in two different ways as shown in Fig. 4. The Android API (AAPI) dataset[1], which includes both Android API calls and their corresponding documentation, is used in both approaches: (1) In the fine-tuned model, the trained and published model by the authors of NCS is used to initialize the weights of the model, and then the model is fine-tuned using the AAPI dataset; (2) In the AAPI model, the NCS model is trained from scratch using only the AAPI dataset. Both models use the same hyperparameter settings, and are both trained with the Adam optimizer for 50 epochs. However, the early termination strategy is applied unless there is no improvement seen from the model for 20 iterations. The other hyperparameters are as follows. The learning rate is 0.001, the mini-batch size is 64, and the dropout rate is 0.2. In addition, parameters related to the vocabulary for source code and code summaries are used as in NCS [14]. During the inference, the beam size, which helps to return high probability outputs, is set to 4.

Fig. 4 NSC approach overview



4.3.2 Code2Desc

The current study introduces a new transformer-based machine translation model called "Code2Desc."The transformer, which forms the basis of the model, includes an encoder and a decoder with an attention mechanism and a positional encoder, which maintains the parallel structure of the converter and the order of the words. The model is trained for 48 epochs and uses a 64 batch size. The model has a dictionary size of 50,000 entries and produces 256-dimensional embeddings.

The model performs the following steps. The AAPI training set is vectorized separately for both code fragments and descriptions. At each training step, the vectorized inputs are sent to the encoder in order to generate a new representation, which is then transmitted to the decoder and the target words obtained (target words from 0 to N). The decoder then attempts to predict the subsequent words in the target word string (N+1 and beyond). Finally, the weights of the model are updated according to the predictions made.

The architecture of the proposed model is illustrated as shown in Fig. 5. Regular expressions (regex) are applied to the model's output so as to increase both the readability and accuracy of the predictions produced in the inference dataset, which the model has not previously seen before. These regular expressions delete sequentially repeated words and stopwords occurring at the end of sentences. An example of this last step is presented in Fig. 6. The NLTK and regex libraries are used for the writing of regular expressions; however, it should be noted that the same regular expressions are applied to the NCS-based models' outputs for the purposes of fair comparison.

4.3.3 Datasets

Three datasets were used in the current study to train and evaluate the previously described models. The first is a publicly available Java dataset [59], which was originally used to develop the NCS [14] model. Even though a few datasets can be found in the literature for Java code summarization (e.g., Funcom [60], NCS [14]), to the best of our knowledge, there are no publicly available Android source code datasets. Therefore, in order to achieve the aim of generating descriptions for Android source code, the second and third datasets were created by the authors of the current study.

The second dataset, called AAPI (Android API), includes both the Android API call source codes together with their natural language descriptions taken from the Android API References. The dataset includes a total of 95,380 source code-summary pairs. Since API calls could help to analyze the behavior of Android applications and sensitive API calls are extensively used to detect malicious Android applications [61], they were also employed in the study for the purpose of source code summarization. Moreover, it has been shown that as the detail in human-generated code summaries increases, so does the use of keywords related to API calls [38]. The original NCS model was trained with a Java source code dataset. Android Java is relatively different from the core Java language that is limited to the object-oriented programming paradigm. The AAPI dataset, which is a platform-targeted

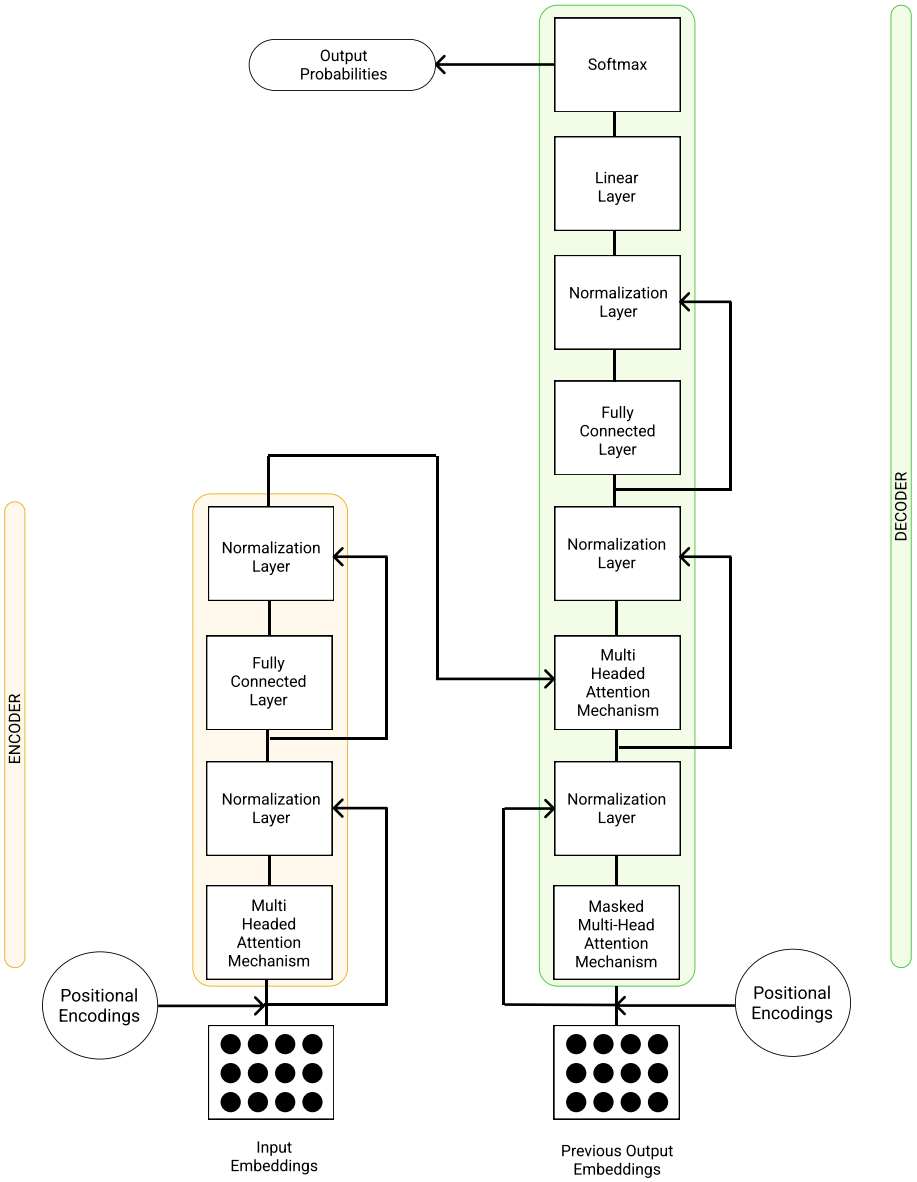


Fig. 5 Code2Desc's architecture

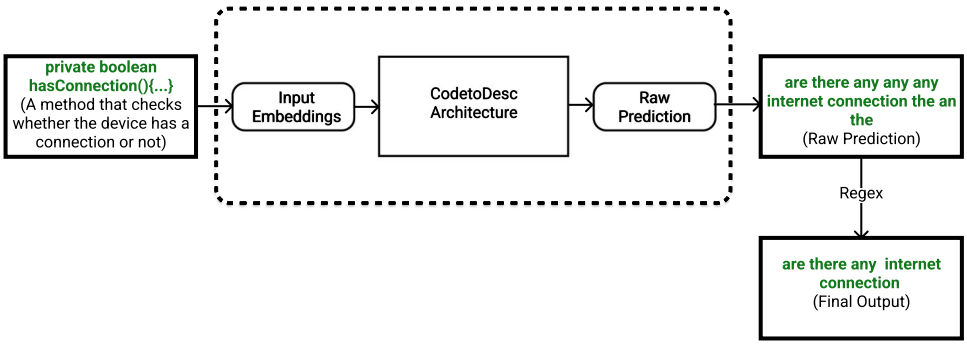


Fig. 6 An example of the post-process step on Code2Desc's predictions

Table 12 Datasets used for source code summarization

Dataset	Train	Validation	Test
Java	69,708	8714	8714
Android API (AAPI)	69,565	9831	9953
Android source code	0	0	400

Table 13 Bleu scores of the models based on AAPI dataset

Model	Bleu
Fine-tuned NCS model	43.56
AAPI Model	45.56
Code2Desc	27.93

dataset, is then used to increase the NCS model's capability to summarize source code written in the event-based Java programming language for developing Android apps. More specifically, this dataset is used for updating the fine-tuned NCS [14] model weights and learning the AAPI and Code2Desc models' weights.

The Android source code dataset is the third dataset used in the study. It contains Android methods that may include security-sensitive behavior such as the use of dangerous permissions or leaking data. Unlike the first two datasets, the Android source code dataset contains only the source code of the library methods, and not their textual descriptions. Therefore, it is only used for the testing of the models, with the output of the models evaluated manually. The sizes of each dataset are presented in Table 12.

The newly introduced Android API and Android source code datasets were preprocessed [14] in the same way as the Java dataset, by splitting the camel case notation into subtokens. Furthermore, natural language summaries shorter than two words were excluded from the datasets, and Android API summaries indicating that they were inherited from other documentation were also discarded. The remaining API source code and summary pairs were shuffled and randomly split into training, validation, and testing datasets based on a proportion of 8:1:1, which was shown to be effective in [36].

4.3.4 Experiments

Three models were evaluated using the AAPI testing dataset. Their resulting BLEU scores (see Table 13) showed that they were comparable with the state-of-the-art approaches [36] in terms of source code summarization. The results shows that training the NCS model from scratch using the platform-targeted AAPI dataset performed better than the fine-tuned NCS model. This gap results from the training data that each model used. As the fine-tuned model is based on the original NCS model, and trained using source code written in the core Java programming language, these results also prove the differences between Android and Java core programming, and the positive effect of API calls on source code summarization. The original NCS models delivered promising results for Android source code summarization, which has not previously been addressed. The Code2Desc model revealed a lower performance than the NCS-based models, although differences between the architectures may go toward explaining this outcome. The deeper and more complex structure of NCSs may help it to learn the problem more easily than seen with the Code2Desc model.

The BLEU metric, which compares candidate textual translation to one or more reference translations, is a standard metric applied in source code summarization. However, since the Android source code dataset only contains the source code of library methods that use dangerous permissions or leak data, the quality of the generated descriptions on this dataset were evaluated manually. In total, four evaluation metrics were employed:

Readability valuates sentences that do not contain syntactic errors and are deemed easy to read. It should be noted here that a generated sentence could be easily converted to a readable sentence by applying post-processing, such as eliminating repetitive sentences and sentences without a predicate, or removing non-alpha characters.

Correctness evaluates how well the generated sentences describe what the given method does. Approaches in the literature generally evaluate models based on open source code; however, the dataset used in the current study included obfuscated methods. As manual summarization of these methods can prove tricky, these cases were also evaluated and shown in the results.

Expressiveness of vulnerability evaluates if the summarization has information related to the usage of dangerous permissions or data leakage. Here, the expectation was that sentences would not explicitly refer to the usage of permissions, such as "This method uses the ACCESS_COARSE_LOCATION permission."For dangerous permissions, sentences which implicitly refer to their usage such as "get the location of restaurants nearby"are considered valid. The description-to-permission fidelity problem [62] aims to find such permission sentences within the app descriptions, else the problem would be very straightforward and readily solvable using a keyword-based approach. For data leakage, sentences directly referring to the accessing of sensitive data are tagged as valid sentences, as well as those indicating a sink point that may cause data leakage.

Audience divides the target recipients for the generated summaries into two groups, as developers and users. Although source code summarization techniques are mainly targeted at developers, the current study also added generated sentences to app descriptions, which are mainly read by users. It should be noted, however, that if the resulting

sentences are not readable, their audience is tagged as not applicable.

In order to evaluate models using the aforementioned metrics, two subset methods that leak data and use dangerous permissions were extracted from the dataset. Both of the security behaviors, namely the usage of dangerous permission and the existence of data leakage, were evaluated using datasets consisting of 100 methods. The NCS-based models were compared using the same datasets; however, Code2Desc uses different subsets due to the elimination of methods in the final step. In that step, sentences shorter than two words, having stopwords occurring at the end of a sentence, having consecutive repeating words, and found to contain "nan"as a word, were all eliminated. This process of elimination affects different methods for each model. Moreover, it may also present an unfair advantage over certain architectures.

The analysis results of the sentences generated from source code that uses dangerous permissions or has data leakage are presented in Tables 14 and 15, respectively.

Although the models were trained with the AAPI dataset, a different dataset was used to conclude the developed models' capability to generate security-centric annotations for pure Android source code fragments. Moreover, the average length of methods/API calls (378 chars) in the AAPI dataset was much shorter than those in the Android source code dataset (659 chars for dangerous permissions, 910 chars for leaking data). The results showed that the AAPI model, where the proposed NCS method was trained using only the AAPI dataset, outperformed the other models in each of the applied metrics.

The performance gap between the NCS-based models and Code2Desc could be explained due to NCS's deeper and more complex architecture, and that some NCS mechanisms such as copy-attention do not exist in Code2Desc. In most cases, the dangerous permission usage was able to be easily and directly extracted from calls such as *SetGPSListeners(): set GPS listeners, GetLastKnownLocation(): get last known location* Having these types of API calls in the methods had a positive effect on the results due to the copy-attention mechanism used in NCS-based models. This mechanism copies words, which are source code fragments in terms of the current study, to the target sequence with the help of attention. When the copied words are directly related to the method's functionality and expressiveness of dangerous permission, the model's performance is increased. Moreover, since many permissions directly relate to obfuscation-resilient API calls, these models perform reasonably well even on obfuscated methods.

It should be noted, however, that the readability score covers sentences that become readable only after post-processing. While the NCS-based models produce more sentences that require post-processing (23–24% of all sentences), only 12% of the sentences generated by Code2Desc required post-processing, such as for the elimination of repetitive phrases.

The performance of the proposed models on the methods that leaked data was found to be much lower than for those using dangerous permissions, especially in terms of the *the expressiveness of vulnerability* metric. In the manual analysis, it was observed that the methods using dangerous permissions were more straightforward, shorter, self-explanatory, and less obfuscated. Moreover, the usage of permission might be explained with the existence of one API call. However, generally speaking, it was not found to be the case for those methods in the dataset that leaked data.

Methods that are identified as potentially leaking data may also contain event-based methods such as *onClick()*, *onContextItemSelected()* as they may listen to and capture user events such as button clicks, which could arguably trigger data leakage. These methods are generally considered to be longer than those that utilize dangerous permissions. Moreover, they were observed to have more than one functionality, which could positively affect the performance of the model (in terms of output correctness), since if one of them is explained correctly in the generated output, the output is tagged as being correct.

Obfuscation negatively influences generated sentences as obfuscation replaces meaningful keywords that are possibly descriptive of the functionalities held by the methods. Besides, it hinders the copy-attention mechanism's positive contributions and even causes a negative influence in all of the metrics. As shown in Table 15, when the obfuscation increases, the readability and correctness of the NCS-based models decrease. The length of methods might also negatively affect the results. The length of methods in this test dataset was much higher than the average in the AAPI dataset and those using dangerous permission in the API Source Code dataset. One notable result was that Code2Desc was found to be less affected by obfuscation and long methods, especially for the metrics of *readability* and *correctness*. While the complex architecture of NCS, such as its use of copy-attention, may negatively affect such cases, the more simpler Code2Desc model was able to achieve results comparable with NCS-based models in explaining methods that leak data, and as such this presents as stability.

It should be noted that the amount of naming obfuscation applied to each method in the evaluation was different. While some methods have their names, parameters, return types, and local variables obfuscated, other methods may only have a single referenced variable's package name obfuscated. In the introduced Code2Desc model, the method was able to be coarsely summarized, albeit accurately, when the self-descriptive and relatively more informative parts of a method such as its name, parameter names, and package names responsible for the essential functionality of the method were

Table 14 Analysis of generated sentences for methods using dangerous permissions

Model	Readability (%)	Correctness (%)	Expressiveness of vulnerability (%)	Obfuscated (%)	Audience user (%)	Audience developer (%)
Code2Desc	46	28	31	60	13	31
Fine-tuned NCS	75	35	39	56	15	62
AAPI	76	44	56	56	13	58

Table 15 Analysis of generated sentences for methods leaking data

Model	Readability	Correctness	Expressiveness of vulnerability	Obfuscated	Audience user	Audience developer
Code2Desc	48	28	4	72	4	36
Fine-tuned NCS	64	32	5	74	0	59
AAPI	59	38	9	74	3	53

not obfuscated. Moreover, since API calls are difficult to obfuscate [63] and self-descriptive, the amount of API calls in a method also presents an important factor. It is observed that most of the correctly summarized obfuscated methods (63% for methods leaking data, 95% for methods having dangerous permissions) have non-obfuscated Android API calls and strings such as permission names that helps to summarize the functionality of the method. For example, the proposed model was able to successfully summarize a method named "k"after naming obfuscation was applied, since the method has responsibility for checking the enablement of the network connection by using an API call. Thus, the proposed model's success in summarizing obfuscated methods depends on the level of obfuscation applied and which method components are renamed. Therefore, the proposed model does not guarantee to perform well when a large amount of obfuscation is applied.

As shown in both Tables 14 and 15, generated sentences are generally intended for developers. The main reason behind this is that the training dataset was composed of method-summary pairs that had been extracted from Android API documentation targeted at Android developers. In addition, the copy-attention mechanism copies method-distinctive keywords from the given input sequence (source code). These keywords used in the methods were created by developers and only targeted as being understood by other developers, which follows the widely accepted tenets of code writing. Accordingly, the sentences may not always appear that clear for application users to understand. However, the output could be evaluated by developers, and the information regarding third-party libraries that apps use could be summarized by developers. Due to a lack of documentation, even developers are not fully aware of the security vulnerabilities that such libraries could potentially introduce. As future work, we would suggest the use of more user-friendly descriptions in order to better train summarization models. Moreover, replacing some developer-specific words with more user-friendly options may help to resolve this issue. For example replacing the developer-specific word *return* with the *get* phrase could be seen to increase a phrase's readability amongst application users.

Lastly, all of the models were evaluated on a common small dataset consisting of 30 samples for dangerous permission usage and data leakage. The performance of each model using these common datasets are presented Tables 16 and 17 As can be seen, the results are compatible with those of the previous analysis. All of the methods performed better in terms of explaining dangerous permission usage rather than data leakage; and this was due to obfuscation and the API calls used as well as their length. As expected, the explanations mainly targeted developers. Even though the NCS-based models produced better results, the introduced Code2Desc model showed a distinct level of stability and robustness against obfuscated or long/complex methods. Table 18 presents examples of the explanations generated.

4.4 Limitations and future work

The most significant limitation that faced the current study was that TPLs generally do not share their metadata or they lacked sufficient detail. On the other hand, this proves the necessity for the proposed approach, given its ability to handle more complex data. For this reason, the metadata used in many natural language generation studies is considered insufficient to be used as the only or essential source according to the current study. Metadata often contains only generalized introductory information, and lacks details of API documentation. Therefore, such metadata was only able to be used within two components in the current study: extraction of general information and TPL classification. The performance of the second component, which outputs the library category, is significantly affected by the

Table 16 Analysis of generated sentences for 30 common methods using dangerous permissions

Model	Readability (%)	Correctness (%)	Expressiveness of vulnerability (%)	Obfuscated (%)	Audience user (%)	Audience developer (%)
Code2Desc	36	23	20	56	16	46
Fine-tuned NCS	50	36	40	56	23	60
AAPI	70	53	53	56	20	60

Table 17 Analysis of generated sentences for 30 common methods leaking data

Model	Readability (%)	Correctness (%)	Expressiveness of vulnerability (%)	Obfuscated (%)	Audience user (%)	Audience developer (%)
Code2Desc	40	16	0	73	6	60
Fine-tuned NCS	40	23	3	73	0	46
AAPI	40	46	6	73	10	46

Table 18 Exemplar generated texts

Library	Generated text
Adyen SDK for Android	Adyen Components for Android allows you to accept in-app payments. Its category is UTILS. It gets the current device location.
Logback	It is the reliable, generic, fast, and flexible logging framework. Its category is UTILS. It starts the configuration for a user.
Baidu	It is a route planner for traveling by foot, car, or public transportation. Its category is UTILS. It gets the current network connection from the device.
Maps SDK for Android	It displays maps inside of your Android application. Its category is LOCATION. It sets the current location of the last known location.
AndroidSimpleLocation	It is for easy access to the device location on Android. Its category is LOCATION. It gets the current location in the list, if applicable.
Kofax Android SDK	It engages your customers on their preferred channel. Its category is UTILS. It has a setter for the last known location.
Mapzen Android SDK	Mapzer Android SDK makes your life easier. Its category is UTILS. It returns whether the location of this class has been set or not.

size of the dataset, where it consists of library metadata in known categories. This was clearly shown in the results. It could be said that the current study, which constructed a dataset of libraries, made the first step in showing how to overcome such limitations.

In terms of code summarization, the source code of Android API calls and their descriptions given in the Android API reference were used to generate summaries for Android source methods. However, the source code of API calls and their documentation is short. Therefore, in future works, this dataset could be extended with methods-summary pairs collected from software repositories such as GitHub. However, it should be emphasized that using API calls in the current study had a positive effect, especially for summarizing methods using dangerous permissions since some APIs are directly related to certain permissions.

The primary aim of the current study was to generate explanations on TPLs based on security sensitive behaviors. In future research, these explanations could be extended with additional information, such as adding TPL versions and their best known vulnerabilities according to vulnerability databases.

5 Conclusion

The current study aimed to enhance app descriptions with information about the TPLs they use. In order to achieve that, a library dataset was introduced and subsequently analyzed. The dataset highlighted the limitations of the documentation provided by TPL developers, and thereby the importance of the proposed work. The study proposed three components in order to generate descriptions automatically. The first component extracted general information about TPLs from their documentation, whilst the second component detected their categories. Finally, the third component generated a textual summary of library methods that utilize dangerous permissions or may leak data. While the first model is based on BERT, the second and third components propose new/adapted approaches to address the problem. The

results were analyzed extensively, and a manual evaluation was conducted on the descriptions generated. The study's results were promising in what is arguably still a new field, as the study approached the problem of app description generation from a different perspective. We believe that researchers will benefit from both the introduced datasets and also the proposed new approach.

Acknowledgements We would like to thank TUBITAK for its support. This study is supported by the Scientific and Technological Research Council of Turkey (TUBITAK-118E141).

Declarations

Conflict of interest Author Beyza Cevik declares that he has no conflict of interest. Author Nur Altiparmak declares that she has no conflict of interest. Author Murat Aksu declares that she has no conflict of interest. Author Sevil Sen declares that she has no conflict of interest.

Ethical approval This article does not contain any study with human participants or animals performed by any of the authors.

References

1. Sen, S., Can, B.: Android security using nlp techniques: a review. Preprint arXiv:2107.03072, (2021)
2. Qu, Z., Rastogi, V., Zhang, X., Chen, Y., Zhu, T., Chen, Z.: Autocog: measuring the description-to-permission fidelity in android applications. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1354–1365 (2014)
3. Feng, Y., Chen, L., Zheng, A., Gao, C., Zheng, Z.: Ac-net: assessing the consistency of description and permission in android apps. IEEE Access **7**, 57829–57842 (2019)
4. Alecakir, H., Kabukcu, M., Can, B., Sen, S.: Discovering inconsistencies between requested permissions and application metadata by using deep learning. In: 2020 International Conference on Information Security and Cryptology (ISCTURKEY), pp. 56–56, IEEE (2020)
5. Alecakir, H., Can, B., Sen, S.: Attention: there is an inconsistency between android permissions and application metadata!, pp. 1–19 (2021)
6. Andow, B., Nadkarni, A., Bassett, B., Enck, W., Xie, T.: A study of grayware on google play. In: 2016 IEEE Security and Privacy Workshops (SPW), pp. 224–233, IEEE (2016)
7. Wang, H., Guo, Y.: Understanding third-party libraries in mobile app analysis. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pp. 515–516, IEEE (2017)
8. Privacygrade: Grading the privacy of smartphone apps. (2021). (Visited September 2021) [Online]. Available: http://privacygrade.org/
9. Wang, H., Guo, Y., Ma, Z., Chen, X.: Wukong: a scalable and accurate two-phase approach to android app clone detection. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, pp. 71–82 (2015)
10. Book, T., Pridgen, A., Wallach, D.S.: Longitudinal analysis of android ad library permissions. Preprint arXiv:1303.0857, (2013)
11. Stevens, R., Gibler, C., Crussell, J., Erickson, J., Chen, H.: Investigating user privacy in android ad libraries. In: Workshop on Mobile Security Technologies (MoST), vol. 10, Citeseer (2012)
12. Zhang, M., Duan, Y., Feng, Q., Yin, H.: Towards automatic generation of security-centric descriptions for android apps. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 518–529, ACM (2015)
13. Wu, T., Tang, L., Zhang, R., Wen, S., Paris, C., Nepal, S., Grobler, M., Xiang, Y.: Catering to your concerns: automatic generation of personalised security-centric descriptions for android apps. ACM Trans. Cyber-Phys. Syst. **3**(4), 36 (2019)
14. Ahmad, W.U., Chakraborty, S., Ray, B., Chang, K.: A transformer-based approach for source code summarization. CoRR, arXiv:abs/2005.00653, (2020)
15. Liu, X., Leng, Y., Yang, W., Zhai, C., Xie, T.: Mining android app descriptions for permission requirements recommendation. In: 2018 IEEE 26th International Requirements Engineering Conference (RE), pp. 147–158, IEEE (2018)
16. Wu, T., Tang, L., Zhang, R., Wen, S., Paris, C., Nepal, S., Grobler, M., Xiang, Y.: Catering to your concerns. ACM Trans. Cyber-Phys. Syst. **3**, 1–21 (2019)
17. John, O., Naumann, L., Soto, C.: Paradigm shift to the integrative big five trait taxonomy: History, measurement, and conceptual issues, pp. 114–158. 01 (2008)
18. Yu, L., Zhang, T., Luo, X., Xue, L.: Autoppg: towards automatic generation of privacy policy for android applications. In: Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 39–50 (2015)
19. Chen, W., Aspinall, D., Gordon, A.D., Sutton, C., Muttik, I.: A text-mining approach to explain unwanted behaviours. In: Proceedings of the 9th European Workshop on System Security, p. 4, ACM (2016)
20. Tibshirani, R.: Regression shrinkage and selection via the lasso. J. R. Stat. Soc. Ser. B (Methodol.) **58**(1), 267–288 (1996)
21. Grace, M.C., Zhou, W., Jiang, X., Sadeghi, A.-R.: Unsafe exposure analysis of mobile in-app advertisements. In: Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks, pp. 101–112 (2012)
22. He, Y., Yang, X., Hu, B., Wang, W.: Dynamic privacy leakage analysis of android third-party libraries. J. Inf. Secur. Appl. **46**, 259–270 (2019)
23. Gorla, A., Tavecchia, I., Gross, F., Zeller, A.: Checking app behavior against app descriptions. In: Proceedings of the 36th International Conference on Software Engineering, pp. 1025–1035 (2014)
24. Zhang, C., Wang, H., Wang, R., Guo, Y., Xu, G.: Re-checking app behavior against app description in the context of third-party libraries. In: SEKE, pp. 665–664 (2018)
25. Narayanan, A., Chen, L., Chan, C.K.: Addetect: automated detection of android ad libraries using semantic analysis. In: 2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), pp. 1–6, IEEE (2014)
26. Liu, B., Liu, B., Jin, H., Govindan, R.: Efficient privilege de-escalation for ad libraries in mobile apps. In: Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, pp. 89–103 (2015)
27. Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C.: A survey of machine learning for big code and naturalness. ACM Comput. Surv., vol. 51 (2018)
28. Haiduc, S., Aponte, J., Marcus, A.: Supporting program comprehension with source code summarization. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10, (New York, NY, USA), pp. 223–226, Association for Computing Machinery (2010)
29. LeClair, A., Jiang, S., McMillan, C.: A neural model for generating natural language summaries of program subroutines. CoRR, arXiv:abs/1902.01954 (2019)
30. Sridhara, G., Pollock, L., Vijay-Shanker, K.: Automatically detecting and describing high level actions within methods. In: 2011

33rd International Conference on Software Engineering (ICSE), pp. 101–110 (2011)
31. McBurney, P., McMillan, C.: Automatic documentation generation via source code summarization of method context. In: 2nd International Conference on Program Comprehension, ICPC 2014 - Proceedings, 06 (2014)
32. Alon, U., Levy, O., Yahav, E.: code2seq: Generating sequences from structured representations of code. CoRR, arXiv:abs/1808.01400, (2018)
33. Hu, X., Li, G., Xia, X., Lo, D., Jin, Z.: Deep code comment generation. In: Proceedings of the 26th Conference on Program Comprehension, ICPC '18, (New York, NY, USA), pp. 200–210, Association for Computing Machinery (2018)
34. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. CoRR, arXiv:abs/1409.3215, (2014)
35. Wang, W., Zhang, Y., Zeng, Z., Xu, G.: Trans^3: A transformer-based framework for unifying code summarization and code search. CoRR, arXiv:abs/2003.03238, (2020)
36. Shi, E., Wang, Y., Du, L., Chen, J., Han, S., Zhang, H., Zhang, D., Sun, H.: Neural code summarization: How far are we? (2021)
37. Hu, X., Li, G., Xia, X., Lo, D., Lu, S., Jin, Z.: Summarizing source code with transferred api knowledge. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18, pp. 2269–2275, International Joint Conferences on Artificial Intelligence Organization, 7 (2018)
38. Rodeghero, P., McMillan, C., Shirey, A.: Api usage in descriptions of source code functionality. In: 2017 IEEE/ACM 1st International Workshop on API Usage and Evolution (WAPI), pp. 3–6, IEEE (2017)
39. Shahbazi, R., Sharma, R., Fard, F.H.: Api2com: On the improvement of automatically generated code comments using API documentations. CoRR, arXiv:abs/2103.10668, (2021)
40. Android arsenal: Android developer portal with tools, libraries, and app. https://android-arsenal.com/. Online; last accessed on April 4 (2022)
41. Sonatype, Maven central repository search. https://search.maven.org/, 2017. Online; last accessed on November 2 (2021)
42. JFrog, I.: Spring.io. https://repo.spring.io/, 2013. Online; last accessed on November 2 (2021)
43. JFrog, I.: Jcenter is the place to find and share popular apache maven packages. https://bintray.com/bintray/jcenter, 2016. Online; last accessed on November 2 (2021)
44. JitPack, Jitpack | publish jvm and android libraries. https://jitpack.io/, 2015. Online; last accessed on November 2 (2021)
45. Backes, M., Bugiel, S., Derr, E., McDaniel, P., Octeau, D., Weisgerber, S.: On demystifying the android application framework: Re-visiting android permission specification analysis. In: 25th {USENIX} security symposium ({USENIX} security 16), pp. 1101–1118 (2016)
46. Ma, Z., Wang, H., Guo, Y., Chen, X.: Libradar: Fast and accurate detection of third-party libraries in android apps. In: 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), pp. 653–656 (2016)
47. Li, M., Wang, W., Wang, P., Wang, S., Wu, D., Liu, J., Xue, R., Huo, W.: Libd: scalable and precise third-party library detection in android markets. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 335–346 (2017)
48. Zhang, Y., Dai, J., Zhang, X., Huang, S., Yang, Z., Yang, M., Chen, H.: Detecting third-party libraries in android applications with high precision and recall. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 141–152 (2018)
49. Derr, E., Bugiel, S., Fahl, S., Acar, Y., Backes, M.: Keep me updated: An empirical study of third-party library updatability on android. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17, (New York, NY, USA), pp. 2187–2200, ACM (2017)
50. Zhang, J., Beresford, A.R., Kollmann, S.A.: Libid: reliable identification of obfuscated third-party android libraries. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, (New York, NY, USA), p. 55-65, Association for Computing Machinery (2019)
51. Backes, M., Bugiel, S., Derr, E., McDaniel, P., Octeau, D., Weisgerber, S.: On demystifying the android application framework: Re-visiting android permission specification analysis. In: 25th {USENIX} security symposium ({USENIX} security 16), pp. 1101–1118 (2016)
52. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Notices **49**(6), 259–269 (2014)
53. Devlin, J., Chang, M., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. CoRR, arXiv:abs/1810.04805, (2018)
54. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., Drame, M., Lhoest, Q., Rush, A.M.: Transformers: State-of-the-Art Natural Language Processing 10 (2020)
55. Bird, S.: Nltk: the natural language toolkit. In: Proceedings of the COLING/ACL 2006 Interactive Presentation Sessions, pp. 69–72 (2006)
56. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al.: Scikit-learn: machine learning in python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
57. Stevens, R., Gibler, C., Crussell, J., Erickson, J., Chen, H.: Investigating user privacy in android ad libraries. In: Workshop on Mobile Security Technologies (MoST), vol. 10, Citeseer (2012)
58. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. CoRR, arXiv:abs/1706.03762, (2017)
59. Hu, X., Li, G., Xia, X., Lo, D., Lu, S., Jin, Z.: Summarizing source code with transferred api knowledge. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18, pp. 2269–2275, International Joint Conferences on Artificial Intelligence Organization, 7 (2018)
60. LeClair, A., McMillan, C.: Recommendations for datasets for source code summarization. CoRR, arXiv:abs/1904.02660, (2019)
61. Feizollah, A., Anuar, N.B., Salleh, R., Wahab, A.W.A.: A review on feature selection in mobile malware detection. Digital Investig. **13**, 22–37 (2015)
62. Qu, Z., Rastogi, V., Zhang, X., Zhu, T., Chen, Z.: Autocog: measuring the description-to-permission fidelity in android applications. In: Proceedings of the ACM Conference on Computer and Communications Security, pp. 1354–1365, 11 (2014)
63. Zhang, F., Huang, H., Zhu, S., Wu, D., Liu, P.: Viewdroid: towards obfuscation-resilient mobile application repackaging detection. In: Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks, pp. 25–36 (2014)

Terms and Conditions

Springer Nature journal content, brought to you courtesy of Springer Nature Customer Service Center GmbH (“Springer Nature”).
Springer Nature supports a reasonable amount of sharing of research papers by authors, subscribers and authorised users (“Users”), for small-scale personal, non-commercial use provided that all copyright, trade and service marks and other proprietary notices are maintained. By accessing, sharing, receiving or otherwise using the Springer Nature journal content you agree to these terms of use (“Terms”). For these purposes, Springer Nature considers academic use (by researchers and students) to be non-commercial.
These Terms are supplementary and will apply in addition to any applicable website terms and conditions, a relevant site licence or a personal subscription. These Terms will prevail over any conflict or ambiguity with regards to the relevant terms, a site licence or a personal subscription (to the extent of the conflict or ambiguity only). For Creative Commons-licensed articles, the terms of the Creative Commons license used will apply.
We collect and use personal data to provide access to the Springer Nature journal content. We may also use these personal data internally within ResearchGate and Springer Nature and as agreed share it, in an anonymised way, for purposes of tracking, analysis and reporting. We will not otherwise disclose your personal data outside the ResearchGate or the Springer Nature group of companies unless we have your permission as detailed in the Privacy Policy.
While Users may use the Springer Nature journal content for small scale, personal non-commercial use, it is important to note that Users may not:

1. use such content for the purpose of providing other users with access on a regular or large scale basis or as a means to circumvent access control;
2. use such content where to do so would be considered a criminal or statutory offence in any jurisdiction, or gives rise to civil liability, or is otherwise unlawful;
3. falsely or misleadingly imply or suggest endorsement, approval , sponsorship, or association unless explicitly agreed to by Springer Nature in writing;
4. use bots or other automated methods to access the content or redirect messages
5. override any security feature or exclusionary protocol; or
6. share the content in order to create substitute for Springer Nature products or services or a systematic database of Springer Nature journal content.

In line with the restriction against commercial use, Springer Nature does not permit the creation of a product or service that creates revenue, royalties, rent or income from our content or its inclusion as part of a paid for service or for other commercial gain. Springer Nature journal content cannot be used for inter-library loans and librarians may not upload Springer Nature journal content on a large scale into their, or any other, institutional repository.
These terms of use are reviewed regularly and may be amended at any time. Springer Nature is not obligated to publish any information or content on this website and may remove it or features or functionality at our sole discretion, at any time with or without notice. Springer Nature may revoke this licence to you at any time and remove access to any copies of the Springer Nature journal content which have been saved.
To the fullest extent permitted by law, Springer Nature makes no warranties, representations or guarantees to Users, either express or implied with respect to the Springer nature journal content and all parties disclaim and waive any implied warranties or warranties imposed by law, including merchantability or fitness for any particular purpose.
Please note that these rights do not automatically extend to content, data or other material published by Springer Nature that may be licensed from third parties.
If you would like to use or distribute our Springer Nature journal content to a wider audience or on a regular basis or in any other manner not expressly permitted by these Terms, please contact Springer Nature at

onlineservice@springernature.com