DocFlow: Extracting Taint Specifications from Software Documentation

Marcos Tileria
Marcos.tileria.2016@live.rhul.ac.uk
Royal Holloway
University of London
United Kingdom

Jorge Blasco
jorge.blasco.alis@upm.es
Universidad Politécnica de Madrid
Spain

Santanu Kumar Dash
Santanu.Dash@rhul.ac.uk
Royal Holloway
University of London
United Kingdom

ABSTRACT

Security practitioners routinely use static analysis to detect security problems and privacy violations in Android apps. The soundness of these analyses depends on how the platform is modelled and the list of sensitive methods. Collecting these methods often becomes impractical given the number of methods available, the pace at which the Android platform is updated, and the proprietary libraries Google releases on each new version. Despite the constant evolution of the Android platform, app developers cope with all these new features thanks to the documentation that comes with each new Android release. In this work, we take advantage of the rich documentation provided by platforms like Android and propose `DocFlow`, a framework to generate taint specifications for a platform, directly from its documentation. `DocFlow` models the semantics of API methods using their documentation to detect sensitive methods (sources and sinks) and assigns them semantic labels. Our approach does not require access to source code, enabling the analysis of proprietary libraries for which the code is unavailable. We evaluate `DocFlow` using Android platform packages and closed-source Google Play Services libraries. Our results show that our framework detects sensitive methods with high precision, adapts to new API versions, and can be easily extended to detect other method types. Our approach provides evidence that Android documentation encodes rich semantic information to categorise sensitive methods, removing the need to analyse source code or perform feature extraction.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; • **Software and its engineering** → *Development frameworks and environments.*

KEYWORDS

Taint analysis, Documentation, Android, Natural Language Processing

ACM Reference Format:
Marcos Tileria, Jorge Blasco, and Santanu Kumar Dash. 2024. DocFlow: Extracting Taint Specifications from Software Documentation. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3623312

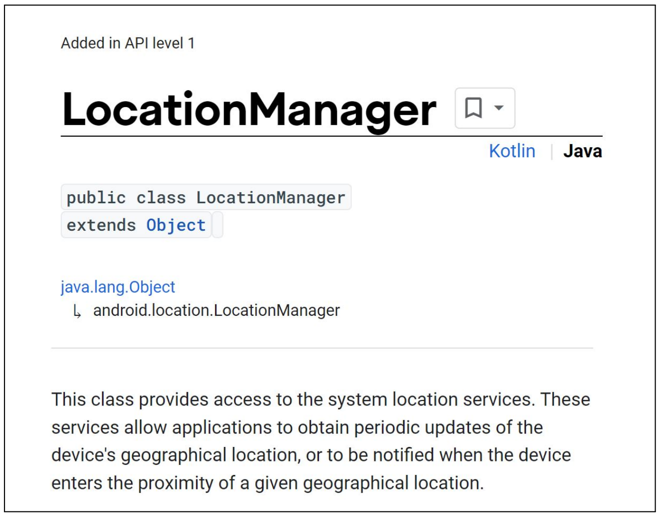
1 INTRODUCTION

Smart devices enable an app-based ecosystem where users can run a myriad of applications seamlessly on any device. Android is arguably the most popular platform with more than 3 billion active devices [22]. Although convenient for users, this ecosystem exposes user's data and Personal Identifiable Information (PII) to data leaks and other privacy violations [5, 40, 33]

Security analysts use static and dynamic analysis frameworks to detect vulnerabilities, bugs, and privacy violations [7, 5, 46, 12, 18]. These frameworks often require an initial configuration to define the scope of the analysis. For instance, taint tracking requires a list of sources and sinks (taint specifications) to connect data flows. Taint specifications are critical to detect security violations. Manually collecting this information is time-consuming and prone to errors. Moreover, malware developers can use less known methods to bypass code analysers [5, 44]. Therefore, automatic techniques to collect security-sensitive methods are desirable.

Several approaches have been proposed in the past to find sensitive methods in Android [5, 33, 47, 12, 18]. These works rely on either code analysis or language specification to find candidate methods from a large collection of methods. These techniques present limitations when dealing with proprietary code and obfuscation tools, which raise the complexity of automated code analysis [17].

Like many other platforms, the Android development team produces and maintains documentation about the different APIs they offer to app developers (even when some of these APIs are offered by proprietary libraries). The documentation of the Android framework and Google libraries are available in Javadoc format and on the developer's website. The documentation of a method describes the purpose, the inputs and outputs, and sometimes other considerations. Figure 1 shows an extract of the documentation corresponding to the `LocationManager` class. This online documentation is generated from the Javadoc description in its source code. The class description is complemented by permissions details (omitted in the figure) and class hierarchy. In Figure 2, we can observe the `getLastKnownLocation` documentation that consist of the method signature followed by a description and permission information (parameters and return type details are omitted). The full signature includes the access modifier (public), the return type (Location), the method name (getLastKnowLocation), and the parameters (String provider). Overall, Android documentation is a great source for extracting syntactic and semantic information about API methods and classes. In this work, we demonstrate that this documentation can be leveraged to classify Android methods and extract taint specifications without the need to process program code.


Added in API level 1

LocationManager

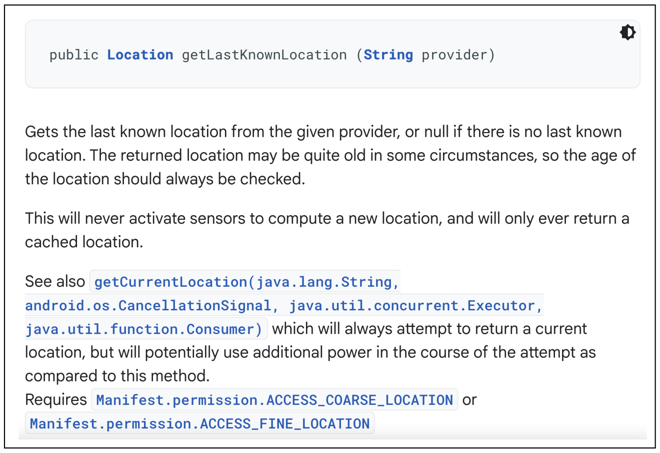
Kotlin | Java

```
public class LocationManager
extends Object
```

java.lang.Object
↳ android.location.LocationManager

This class provides access to the system location services. These services allow applications to obtain periodic updates of the device's geographical location, or to be notified when the device enters the proximity of a given geographical location.

Figure 1: `LocationManager` class online documentation


```
public Location getLastKnownLocation (String provider)
```

Gets the last known location from the given provider, or null if there is no last known location. The returned location may be quite old in some circumstances, so the age of the location should always be checked.

This will never activate sensors to compute a new location, and will only ever return a cached location.

See also `getCurrentLocation(java.lang.String, android.os.CancellationSignal, java.util.concurrent.Executor, java.util.function.Consumer)` which will always attempt to return a current location, but will potentially use additional power in the course of the attempt as compared to this method.
Requires `Manifest.permission.ACCESS_COARSE_LOCATION` or `Manifest.permission.ACCESS_FINE_LOCATION`

Figure 2: `getLastKnowLocation()` method documentation

NLP techniques have been used before to analyse app metadata [24, 48], permissions [40, 32], user reviews and privacy policies[38, 26, 63, 35]. Previous works have highlighted the importance of software documentation to extract semantic information about programs [25, 57, 58, 42]. Other works have shown how API documentation is a rich source for understanding the purpose of software components [29, 45], and the interaction between them [16]. However, little attention has been given to API documentation to detect sensitive methods for security analysis.

Our Work. This work proposes a novel approach to detect sensitive API methods based on the analysis of their documentation via NLP. `DocFlow` uses the API documentation to represent Android methods in high-dimensional embeddings. The distributed embeddings encode semantic information that enables the inference of security and semantic properties that can be used to generate taint specifications and perform semantic search operations for different purposes. `DocFlow` can be used to detect sensitive methods (sources and sinks), classify methods into semantic categories (location, network, etc.), and it is able to find semantic-similar methods across Android versions. Thus, `DocFlow` can be used to generate taint analysis specifications and extract semantic properties from software documentation.

Our framework implements the full pipeline, from crawling and parsing the documentation to the classification and semantic tasks. We use `DocFlow` to detect sensitive methods for the Android API version 32, and several Google proprietary libraries. Moreover, we show that `DocFlow` can detect sensitive methods on each API release and different versions of one method based on its semantics. We compare our results with other tools and find that `DocFlow` achieves comparable or better results. In summary, this paper's contributions are:

- We demonstrate that API documentation in well-documented libraries can be reliably used for auto-identification of sensitive methods and classify them into semantic categories.
- We present `DocFlow`, an end-to-end automatic tool that uses NLP to classify sensitive methods, perform semantic search operations, and generate taint specifications.
- We compare our approach to published and well-cited baselines and significantly outperform them. We do this without access to the source code of the libraries, unlike the baselines. Moreover, our approach incorporates clustering and semantic search capabilities, whereas the baselines only offer classification capabilities.
- We show that our approach is robust against the Android framework and documentation evolution by showing consistently accurate classification results across multiple versions of software documentation.
- Both `DocFlow` and the corresponding artefacts for this work are available online [1].

2 INFORMATION FLOWS IN ANDROID

Android apps consist of a set of components that sit on the top of the Android Application Framework [34]. Developers call APIs in this framework to request resources and interact with the Android Runtime and native libraries. Additional components are normally available via Google Play Services (GPS) libraries. These bundle libraries expose stub methods implemented in the proprietary app *Google Play Services*. App developers use these libraries by invoking their public APIs. Both, the Android Application Framework and libraries such as GPS offer methods that access sensitive resources within the system (camera, contacts, account information, etc.).

2.1 Sensitive methods

Android apps manipulate a variety of sensitive information, such as unique identifiers, geolocation, and sensor readings. These data can be exposed to third parties via APIs that access shared resources.

[1]https://gitlab.com/s3lab-rhul/android/docflow

Taint analysis is normally used to detect such situations[19, 33, 5, 12]. This analysis aims to find a connection between source and sink methods. Taint analysis tools require a taint specification that defines the list of sensitive sources and sensitive sinks.

Android apps do not have access to resources outside their sandbox by default and rely on API calls to access external resources. Consider the code snippet in Listing 1. In this simplified example, the method `getLastKnownLocation()` in line 2 reads data from the `LocationManager`, which is the resource that manages location services. Here, the location variable is leaked through the file system using the `writeFile` method (line 8) and the method `post` that executes an HTTP request (line 12). In this example, if only the HTTP sinks are included in the taint specification, any leak through the file system will remain undetected.

```
1  public void onCreate(Url url, String filePath){
2      loc = LocationManager.getLastKnownLocation()
3      ...
4      //some code
5      ...
6      FileOutputStream file;
7      file = new FileOutputStream(“file_name");
8      file.writeFile(filePath,loc);
9
10     HttpURLConnection connection;
11     connection = new HttpURLConnection(url);
12     connection.post(url, loc);
13 }
```

Listing 1: Code showing sources and sinks use

In Android, sources and sinks are selected from API methods exposed in the Application Framework API and Google Play Services libraries. Sources can be defined as methods that read shared resources and return non-constant values into the application code [5]. Likewise, sink methods write a non-constant value to a shared resource outside the application context. Examples of shared resources in Android include the file system, network connections, and system broadcasts. Note that the idea of sensitive methods depends on the goals of the analysis. For instance, an analyst might be only interested in PII sources and network sinks. Therefore, a fine-grained categorisation is usually required.

Looking at the documentation of the `getLastKnowLocation` method and the corresponding class, we can infer that this method is a source. The class documentation describes the location services it provides, and the method documentation describes what type of location information is retrieved. This example also shows the rich semantic information that class and method names can provide if good development practices are in place. However, manually selecting sensitive methods does not scale well in practice.

2.2 Challenges

Android receives constant updates to cope with new features and hardware. Currently, there are 13 major Android versions and 33 API levels. While Android versions usually refer to features visible to users, API releases are related to changes in the Android framework. Each release contains hundreds of thousands of methods, and app developers struggle to keep up with the pace [34, 8]. On top of API levels, updates from 40 closed-source Google Play Services libraries make the ecosystem even more complex. Google libraries have received between 45 and 62 updates per year in the period 2018-2022.

With the ever-changing Android framework and Google libraries, collecting a comprehensive list of sources and sinks becomes critical for analysis completeness. It has been shown that static analysis tools miss a large number of data leaks due to incomplete configurations [5, 46]. Moreover, apps tend to have more sensitive data leaks with newer Android releases [8], augmenting the importance of complete taint specifications. Therefore, it is desirable to automate this task due to the abundance of Java APIs and Google libraries available.

Software documentation is a rich source of information for learning about programs, and it is instrumental to the success of software that relies on APIs [29]. For instance, representative names, perceptible relations between API types, and accurate descriptions are fundamental for good API usability [45]. At the same time, these good practices enable complex inferences. For instance, Chaudhary *et al.* developed a compiler extension that searches bugs based on rules extracted from document specifications [11]. Similarly, Gorla *et al.* identified several abnormalities by checking apps' behaviour against their description [24]. The Android architecture is designed for reusing library components accessed through their APIs. As Android is intended to be used by millions of third-party developers, its source code is accompanied by rich and detailed documentation. This makes Android API documentation a good candidate for using a learning-based approach to infer semantic properties of the underlying implementation.

3 OVERVIEW OF DOCFLOW

In this section, we describe `DocFlow`, a system that leverages documentation to generate sink and source definitions for the Android platform. `DocFlow` takes the description of an API method and generates a vector representation that is then used to classify and extract semantic information about that method. We first present our design principles, and then `DocFlow` implementation details. We evaluate `DocFlow` in Section 4.

3.1 Design Principles

`DocFlow` uses NLP techniques to detect sensitive methods using Android documentation. Word embedding techniques generate distributed representation of words by mapping text to high dimensional vectors[37, 36]. This representation aims to capture the word/sentence semantics in vectors of real numbers. Thus, the meaning is “distributed” across multiple components instead of local representation where each element represents exactly one component [2]. `DocFlow` uses this technique to calculate the vector representation of Android methods based on their documentation. In particular, our methodology is based on the following observations about Android documentation:

Maturity. The Android source code is well documented. All methods from the Application and Java frameworks have their corresponding documentation. Android AOSP and Google Play Services developers do not accept commits without a proper code style, including the Javadoc of every non-trivial method. We provide more details when describing the dataset in the next section.

Structure. Android developers follow a convention when writing documentation. For instance, every class must have a description and methods documentation should start with a third-person descriptive verb. These guidelines allow us to learn patterns from Android Javadoc. The style guide for contributors is described in Android's website [23].

Semantics. A method's documentation reflects the contract between the caller and callee methods. An API is defined by the combination of the caller interface and the description of the usage protocol (signature) and semantics [45], while the implementation details are hidden in the callee method. In general, the documentation specifies the purpose of a method, parameters, and return type. Additionally, a class description includes the purpose of the class and the abstraction it provides.

These properties make Android a suitable platform to automate method classification via documentation analysis (instead of a code-based approach). All this documentation provides rich semantic context that can be used to make natural language inferences. Note that this approach does not apply to every framework because it requires well-documented software. Nevertheless, most popular frameworks need to offer well-documented APIs so `DocFlow` is specially suited for these very popular and well-documented frameworks such as Android.

3.2 `DocFlow` Methodology

Figure 3 presents an overview of `DocFlow`. We briefly describe the overall approach, and then we go through each step in detail. First, our framework crawls the Android website to download and parse the Android and Google Play Service documentation. The second step sanitises the raw data and generates different input formats for each method. These differ in the amount of information that describes each method and are used to find the best representation. The Classification module encompasses the components to perform the classification tasks. First, we classify sources and sinks using a Transfer Learning [4] approach which consists of two layers. First, the *Embedding Layer* calculates the vector embeddings using a Language Model and transforms each method representation into a high-dimensional distributed vector. The *Method Classifier* layer receives the input from the Embedding Layer and separates security-sensitive from non-sensitive methods. The *Semantic Classifier* module further classifies methods into semantic categories, e.g., file system or network connection. `DocFlow` includes a *Semantic Search* module that reuses the pre-calculated embeddings from the method classifier to search for methods that are close in the vector space using metrics such as cosine similarity. Last, we use the output of the Classification Module to generate taint specifications that can be used with flow analysis tools. With this pipeline, `DocFlow` relieves security analysts of much of the substantial manual effort of generating taint specifications. In the next sections, we describe each component in detail.

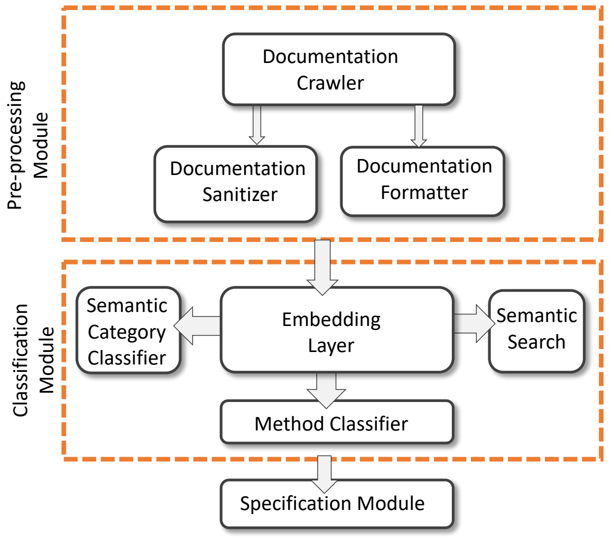


Figure 3: `DocFlow` overview.

3.3 Pre-processing Module

This module offers three main tasks: Download the documentation, sanitise the text, and generate the input for the Classification Module. `DocFlow`'s crawler downloads the reference section of the Android developers and Google Play Services websites[2]. The first step consists of extracting all the Android packages from the website map. Then, `DocFlow` iterates over all packages, classes, nested classes, interfaces, and methods to extract the documentation. For each method, we extract the signature and description. The signature includes the class name, method name, return type and parameters. Additionally, we download the description of all classes. All this information is stored in JSON format. The next step processes the JSON files to generate the representation for each method.

3.3.1 ***Sanitisation step.***

This module cleans the data from the JSON files and then generates different representation formats. The raw data from the files contain special characters, numbers, and sometimes HTML elements. All these characters must be cleaned before generating the method representation that is passed to the embedding layer for further processing. For this purpose, the `DocFlow` formatter module implements the following three steps:

1. Sift and Uniquify. We remove methods with short description, e.g., less than 20 characters or 3 tokens. We also remove duplicated methods to avoid the duplication effect [1]. The Android framework contains many classes that offer the same functionality across different packages. We considered two methods duplicated if they have the same method name and description.

2. Sub-tokenise. We sanitise tokens for rare words, e.g, camel case tokens[28], and tokens with special characters (.,$, or numbers). NLP models are unlikely to produce good embeddings for tokens unseen during training or tokens with very low occurrence. Consider the following method description: "Returns the `serverClientId` that was set in the request". While for a human specialist, it is evident

[2]https://developer.android.com/reference/packages
https://developers.google.com/android/reference/packages

the meaning of the token `serverClientId`, the embedding layer will benefit from splitting this token into three sub-tokens (server, client, id). This transformation takes more relevance because of the Android source code writing style.

3. Deprecation-check. We remove all deprecated documents. These contain the deprecated annotation or the word deprecated in the description. While these methods may still be used, their documentation is no longer maintained under the same principles as non-deprecated methods, so they normally contain non-relevant descriptions or point to the newer alternative.

3.3.2 ***Input Generator***. `DocFlow` can generate multiple formats for each method to find its best representation. We aim to use the best representation for a particular classification or clustering task. In the evaluation (Section 4), we explore the effect of using multiple granularities to generate the method representations. For instance, consider the `getLastKnownLocation()` documentation and its class description from Figure 1. The format below corresponds to **Format A** in the evaluation and consists of the method description only.

Gets the last known location from the given provider or null if there is no last known location. The returned location may be quite old in some circumstances, so the age of the location should always be checked.

While a more complex representation, such as **Format E**, can append the class description (shown below) to the method description.

This class provides access to the system location services. These services allow applications to obtain periodic updates of the device's geographical location, or to be notified when the device enters the proximity of a given geographical location.

3.4 Methods Classifier

The Sources-Sinks classifier consists of an *encoding step* and a *classification step*. First, the encoding step calculates the embedding representation of each method using the Sentence-BERT [50] network (`DocFlow` is flexible regarding the specific embedding model). Sentence-BERT can use a siamese or triplet network architecture to produce sentence embeddings efficiently. In this architecture, a pre-trained BERT network [15] (Bidirectional Encoder Representations from Transformers) is connected to a pooling layer to derive fixed-size embeddings (768 dimensions). The network is then fine-tuned with several tasks depending on the available data, e.g. the Semantic Textual Similarity (STS) or question-answering tasks. This model achieved state-of-the-art performance over several NLP task benchmarks[50].

We propose an optional fine-tuning step for the embedding layer to better separate sources/sinks methods in the vector space. For this, we adopt a contrastive learning [13] approach and the STS task. We use an inference architecture and a dataset composed of pair of methods and a similarity score to continue fine-tuning the model. The aim of this step is to minimise the distance between semantically similar sources/sinks and maximise the distance between unrelated methods.

The second step consists of a feature-based classifier that separates sources and sinks from other methods. We connect the embedding layer with a customisable classifier where the final decision is up to the analyst. We tested with different Machine Learning approaches and provide the results in Section 4. This step requires a dataset with annotated labels, .e.g, Android methods with a source, sink, or neither label (See Section 4 for details on the dataset we used). Note that our approach easily generalises to classify other method types. The only requirement is to produce an annotated dataset and (optionally) fine-tune the embedding layer. Moreover, `DocFlow` is flexible regarding the specific model choice for the embedding and classification layer.

3.5 Semantic Category Classifier

This module further assigns a semantic category to each method such as Network and Identifiers. This division is useful in case the analyst is only interested in a subset of methods, which is the case in practice as taint analysers do not scale to cover all possible sources/sinks. For this classification task, we use an unsupervised classifier based on a zero-shot approach [60] that does not require a labelled dataset of Android methods with their semantic categories for training.

The classification problem is modelled as a text entailment problem where the task is to decide if the premise entails the hypothesis. Figure 4 shows an overview of this approach. The method's documentation to be classified (*doc_x*) is used as the premise, and each label (file, location, and network) is converted into a hypothesis, e.g., "is doc_x about location?". Then, an embedding layer encodes the pair and passes the output to a sequence-pair classifier that predicts if the premise entails, contradicts, or is neutral to the hypothesis. The probabilities for entailment are converted into category probabilities, and we finally take the probability of entailment as the probability of the label being true. For instance, if we want to classify the method `getLastKnowLocation`, the classifier first constructs one hypothesis for each possible category (e.g., geolocation, file system), then it passes to the sequence-pair classifier and selects the label with higher probability as the category for the method.

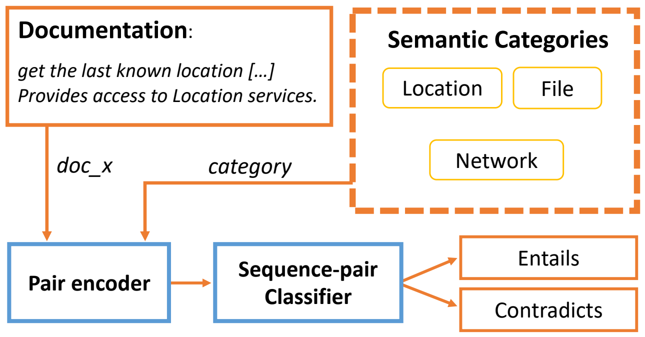


Figure 4: Semantic Category Classifier Overview.

Note that this step is independent of the Method Classifier module and can be used to select sensitive methods from specific semantic categories. This approach enforces label and document understanding and does not rely on task-specific training, so previous observations of labels are not required. Moreover, the DocFlow zero-shot classifier is flexible regarding labels. The user only needs to add or remove categories from the list of labels. Last, this module is also flexible regarding the sequence-pair classifier. In our evaluation, we use a pre-trained model fine-tuned with the MultiNLI dataset [56], but the decision is up to the user.

3.6 Semantic Search

The Semantic Search module solves the problem of finding semantically similar methods. This step reuses the embeddings calculated in the Methods Classifier module (it can also calculate the embeddings from scratch). Figure 5 illustrates our approach for symmetric semantic search (query and corpus entries are roughly the same length). The module first embeds all method representations in the corpus dataset. Then, a query is embedded into the same vector space, and the closest method representations are found using a similarity metric. Overall, the embedding layer derives semantically meaningful vectors that can be compared using similarity measures, e.g., cosine similarity and Euclidean distance. This property allows us to use the embeddings for semantic similarity search and clustering efficiently.

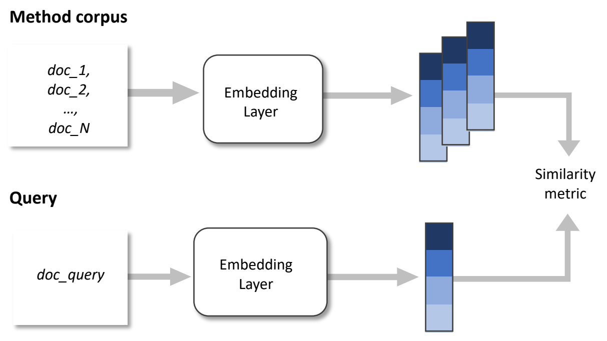


Figure 5: Semantic Search module overview

3.7 Specification Module

The last module generates taint specifications based on the outputs of the Classification Module. In particular, we convert the list of sources/sinks into configuration files that can be used by taint analysers to define security policies. We use the same format as FlowDroid[6] to generate the output files by default. However, the format can be customised as needed.

Additionally, we use the output of the Semantic Classifier to refine the list based on preferred categories. DocFlow's Semantic Search module allows us to easily adapt taint specifications to new API levels or library updates, e.g., the same method with different parameters, a new method that offers similar functionality, and methods that replace deprecated APIs or wrappers.

4 EVALUATION

We evaluate DocFlow with a corpus of Android AOSP and Google Play documentation obtained in March 2022. This corresponds to the API level 32 and Android version 12. The scripts and dataset are available in the repository, including the taint specifications and information flow results.

4.1 Experimental Setup

We first provide details of our corpus, followed by details about the formats we use to represent APIs, their neural embeddings and the details of our classification.

Dataset. We download a corpus of 46227 methods. Android AOSP methods form the bulk of this dataset with 37692 methods from 4538 classes. All framework methods contain a description, and 2% correspond to deprecated methods. We collected 8535 methods from 1191 classes from Google Play Services, of which 80% include a description and 4% are deprecated methods. A further 15% corresponds to methods from deprecated classes. For the remaining cases, we use the method name as the description.

We rely on expert hand labelling to get the ground truth for our dataset. For the training phase, we randomly selected 3K methods from the SuSi dataset and manually annotated each with one of three categories: sources 35%, sinks 21% and neither 43%. Three raters with expertise in taint analysis and Android performed this task. We calculated the Fleiss' kappa [20] reliability metric to measure the degree of agreement between the raters and obtain $k = 0.65$, which shows a good level of agreement [3]. All the mismatches involved source-neither or sink-neither labels. The raters discussed such conflicts until they reached an agreement to ensure the quality of the labels. There were no source methods classified as sink or vice-versa. Still, the fact that around 20% of the methods were assigned with different labels shows the limitations of manually collecting sensitive methods, even when the volume of methods is small. We rely on previous works [53, 52] to obtain the labels for Google libraries (523 methods).

Representations. We use four method representation formats (A-D) to evaluate the source-sink classifier and three formats (E-G) for the semantic category classifier. Table 1 shows how each format is constructed. Format A is the simplest case where only the method's description is used. Each format adds more information to the method representation, such as class description and return types. Format D is the one that includes more information and includes the full method signature, method description, and class description. For the semantic category experiment, we emphasise class documentation (instead of method description or signature) because class documentation is a more natural choice to encode semantic categories information.

Format	Method representation
A	method description
B	method name + description
C	method signature + description
D	method signature + description + class description
E	method description + class description
F	class description
G	method name + class description

Table 1: Method representation formats used during DocFlow evaluation for sources and sinks detection.

Embeddings. For the Embedding Layer, we use the pre-trained Sentence-BERT model `all-mpnet-base-v2`, which has the best overall performance [49]. We use a set of 96 hand-annotated methods for fine-tuning this model. First, we assign a similarity score to each possible combination of methods (4k pairs). The score is automatically calculated based on the category (source, sink, neither) and how similar the representations are in the vector space. For instance, if both methods are sources, we add the cosine similarity to a base value. If the pair contains a source and sink, the base value is zero. With this approach, we model similarities based on the category and the method's semantics. Finally, we fine-tune the model with the following configurations: 10% of the training data for warm-up, 4 epochs, batch size of 16, cosine similarity loss, and evaluation every 1000 steps against another set of hand-annotated methods with their corresponding score. We tested with different settings recommended by the Sentence-BERT authors and selected the one with the best performance.

Classification. We split our dataset into training (75%) and test (25%) sets, and we use four classifiers: Logistic regression, SVM, XGboost, and a 1-layer neural network. These classifiers have produced good results with high dimensional embeddings [50, 38, 15, 5]. We evaluate the performance of the classifiers using the following metrics: accuracy, precision, recall and F1-score. We use the cosine similarity metric for semantic search operations.

4.2 Efficacy of Representations

We first evaluate the impact of the four method representation formats on the precision of the classifiers. The intuition is that the classification modules will benefit from a representation that contains more semantic information. Figure 6 illustrates the performance of the classifiers with the different input formats. The results indicate that the precision tend to increases as we add more information to the method representation. The simplest format (A) performs worst in all cases, and the most complex (D), which includes the full signature and class description, achieves the best performance. The difference is less significant between formats (B) and (C). The latter uses more information to represent the signature, but the descriptions remain equal. We use Format D for the following experiments unless otherwise stated.

Next, we explore how the performance of DocFlow changes using different classifiers. We use the four classifiers mentioned above and a custom classifier built on top of the Semantic Search module. For this, we use the embeddings from the training set to classify embeddings from the test set based on the cosine similarity. This approach assigns the label of the most similar method from the training set.

Table 2 reports DocFlow's evaluation metrics using the format D and the fine-tuned Sentence-BERT model. The high precision and recall indicate that the models rarely make mistakes when predicting sources and sinks. While all classifiers achieve comparable results, XGboost achieves the best results considering all metrics. Therefore, we use XGboost as the default classifier for the remaining experiments. Even though the Semantic Search approach achieves the worst results, the performance is still comparable to the other classifiers using a relatively simple approach.

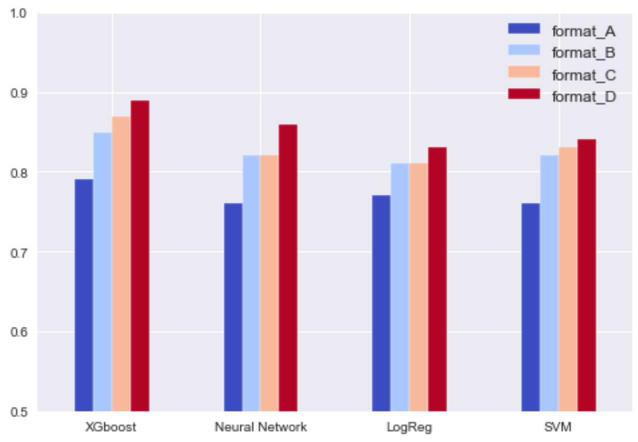


Figure 6: Formats evaluation. The pre-trained model `all-mpnet-base-v2` was used to encode the documentation

Classifiers	Acc.	Prec.	Rec.	F1
LR	0.86	0.85	0.85	0.85
SVM	0.85	0.84	0.84	0.84
XGboost	**0.89**	**0.88**	**0.87**	**0.87**
NN	0.86	0.85	0.85	0.85
S-Search	0.82	0.80	0.79	0.79

Table 2: DocFlow performance using the fine-tuned Sentence-BERT in the embedding layer.

Running DocFlow on our entire corpus produced a total of 11456 sources and 6367 sinks from 37692 framework methods from the API level 32. While DocFlow scales to analyse the entire corpus, we use a subset[3] of the `android.Location` package to manually validate the results. We use this package because it is a popular API that contains many sensitive methods and has significant churn in its API. In total, these classes contain 195 unique methods. DocFlow detects sources with a precision of 90% and 89% recall. The precision and recall for sinks are 70% and 96% respectively. The false positive rate is larger for sinks. This is because methods such as callbacks and constructors are classified as sinks, e.g., `addNmeaListener` and `registerGnssStatusCallback` from the `LocationManager` class. Currently, our dataset has annotations for sources, sinks and all other methods are grouped as neither. The dataset can be extended to model other method types by adding more granular labels.

4.3 Baseline Comparison

We compare DocFlow results against SuSi[5] and SWAN [46]. Both tools use a similar approach (with access to source code) to classify sources and sinks. First, they extract syntactic and semantic features from the source code and then train a Machine Learning classifier. The main difference is that SWAN uses more general

[3]The subset consists of methods from the following classes: `Location`, `LocationProvider`, `LocationManager`,`GnssAntennaInfo`, `PhaseCenterOffset`, `SphericalCorrections`, `GnssMeasurementsEvent`, `GnssMeasurementsEvent`, `GnssNavigationMessage`, `GpsSatellite`, and `GpsStatus`

features while SuSi includes Android-specific features such as permissions. We use two settings for the DocFlow Embedding Layer: (DF) uses a default pre-trained Sentence-BERT, and (FT) performs the optional fine-tuning. Table 3 shows that DocFlow (DF) outperforms both tools in precision and recall. Moreover, we can see that fine-tuning the Embedding Layer consistently improves the results for all classifiers.

	Acc.	**Prec.**	**Rec.**	**F1**
SuSi	0.78	0.79	0.77	0.78
SWAN	0.89	0.76	0.67	0.71
DocFlow (DF)	0.84	0.83	0.82	0.83
DocFlow (FT)	**0.89**	**0.88**	**0.87**	**0.87**

Table 3: DocFlow vs SuSi comparison. (DF) default pre-trained Sentence-BERT and (FT) the fine-tuned model

Identification of Semantic Categories. We evaluate our Semantic Category classifier with a dataset of 358 methods and compare the results with SuSi category classifier (SWAN does not offer this feature). The dataset consists of Android methods annotated with their corresponding category. We use the same set of labels as SuSi for easy comparison. For the Pair-encoder, we use the pre-trained model `bart-large` [60] fine-tuned with the multi_nli dataset [56]. This corpus consists of 433k sentence pairs annotated with textual entailment information.

Table 4 shows the precision and recall of the classifiers. Precision and recall metrics are calculated for each label, and their weighted average is displayed. Format E (method and class description) achieves the best overall performance. Format F (only class description) still gets an equivalent precision compared to SuSi and improves the recall by 10%. By manually analysing the predictions per category, we observe that SuSi fails to assign a semantic category (no-category label) to a large percentage of methods. This performance contrasts with SuSi, where the results are comparable to DocFlow under many settings. One reason for this might be that extracting features from the code does not lead to good models to predict semantic categories. In contrast, our Semantic Category classifier uses natural language descriptions, which are a more natural choice for this task. Another disadvantage of SuSi is that its feature extraction needs to be modified to add a new category. DocFlow zero-shot approach does not require any internal change.

	Acc.	**Prec.**	**Rec.**	**F1**
DocFlow (E)	**0.86**	**0.91**	**0.86**	**0.88**
DocFlow (F)	0.83	0.89	0.83	0.86
DocFlow (G)	0.79	0.89	0.70	0.78
SuSi	0.59	0.88	0.60	0.71

Table 4: Docflow and SuSi semantic category classification

4.4 Platform libraries

As already mentioned, the code for Google Play Services (GPS) libraries is not available. Thus, frameworks that use program analysis techniques to detect sensitive methods cannot model these libraries properly. As DocFlow does not present this limitation, we run DocFlow classifier on all GPS libraries and evaluate four of them (Wearable, TV, Analytic, and Ads). We first run DocFlow for all GPS methods and then validate the performance with the four libraries with labels. In total, DocFlow detected 2787 sources and 2283 sinks from 8535 Google Play Services methods. Table 5 shows the percentage of sources and sinks detected on each GPS library. These results show that DocFlow can detect most of the sources and sinks for these libraries. We further analysed the cases where DocFlow misclassified the methods and found that these cases usually involve callbacks and constructors. We provide an example of how DocFlow can model callbacks below.

	Package	**Srcs**	**Sinks**
Wearable	`com.google.android.gms.wearable`	0.95	0.96
TV	`com.google.android.gms.cast.tv`	0.92	1.00
Analytics	`com.google.android.gms.analytics`	1.00	0.93
Ads	`com.google.android.gms.ads`	0.92	0.85

Table 5: Sources and Sinks detected by DocFlow per library.

Semantic Search. We evaluate the DocFlow Semantic Search module on the task of completing a partial taint specification using the dataset of wearable methods (266 methods with labels) from Google Play Services. In this case, we added the *callback* label to the previous labels (source, sink and neither). Callback methods are important as they allow precise call graph construction for Android apps. We simulate the partial specification by randomly splitting our dataset into two halves: a corpus and a query dataset. This experiment aims to assign labels to the query dataset using the existing taint specification (corpus). The semantic search takes each method from the query dataset and searches for the top three most similar methods in the corpus dataset. We use the cosine similarity as the metric and the fine-tuned Sentence-BERT encoder from previous experiments. We tested all representation formats and two approaches for assigning labels: 1) the closest in the vector space (best-match). 2) The most common of the top three (top-3).

DocFlow achieves an overall precision and recall of 90% using the best-match approach with Format C. Table 6 shows the results per category using these settings. In general, the results using the top-3 approach are worse than the best-match by a margin of 2-10%, depending on the method representation format. The only exception is Format E which shows slightly better performance with the top-3 approach, but the precision and recall are worse than other formats (84%). These results show that DocFlow can be used to either classify sensitive methods or complement existing specifications using the Semantic Search module.

	Total	**Acc.**	**Prec.**	**Rec.**	**F1**
Sources	101	0.97	0.92	0.94	0.93
Sinks	33	0.71	0.88	0.78	0.83
Callbacks	34	0.94	1.00	0.97	0.98

Table 6: Method categorisation using Semantic Search. Labels are assigned according to the best-match approach.

Taint Specification for Libraries. We have shown that DocFlow can classify Google API methods for which no source code is available. Now, we use DocFlow to generate taint specifications for these APIs and feed the specification to FlowDroid [6], a popular taint analyser. The input consists of the list of sources and sinks produced by DocFlow and the output is a configuration file with the signature and classification of the Google API methods. As previously mentioned, our artefacts are available in the replication package. We ran FlowDroid on a dataset of 65 popular streaming apps with the configuration file created by DocFlow. The goal was to detect the presence of sensitive flows that use Google libraries methods. We detected several such flows in the results. For instance, the app com.tru reads custom data from the gsm.cast.MediaInfo package. This data flow reaches the Freshchat SDK via Bundle objects. We found similar flows for the CleverTap SDK. Another example is the com.ted.android app that calls sources from the *gms.analytics.Tracker* package to read sensitive data. While these flows do not necessarily indicate abusive behaviour, they clearly show that developers use Google proprietary packages to read sensitive data. This experiment shows how FlowDroid can leverage DocFlow to study information flows in Android apps. DocFlow enables the automatic detection of sources/sinks and can generate taint specifications. FlowDroid (and other static analysers) can use such specifications to detect sensitive flows, even for Google proprietary libraries.

4.5 Robustness against Software Evolution

One motivation to develop DocFlow is to automatically generate taint specifications for new Android or Google libraries versions. Before evaluating how well DocFlow adapts to new framework versions, we look at how documentation changes across versions (API methods and classes). For this, we use the subset of the android.Location package from four API levels. Table 7 shows the evolution of this package from API level 29 to 32. The values in column New Methods indicate that methods are constantly added on each release. The following three columns indicate how recurrent methods change across API levels. For instance, looking at the API 29 ⟶ 30 update, 18 % of the classes description differ (C), 14% of method descriptions present changes (M), of which 30% are due to methods being deprecated (D). These numbers indicate that the documentation of recurrent methods is also subject to modifications, and only a portion of these changes are due to deprecation. These results align with a recent work [30] that studied changes in code and API documentation across versions.

API Change	New Methods	C	M	D
29 ⟶ 30	14	18	14	30
30 ⟶ 31	21	18	25	16
31 ⟶ 32	9	35	34	5

Table 7: Location package evolution. C (% of classes modified), M (% of methods modified) and D (% of deprectaed methods).

We run the DocFlow classifier (default settings) against the four versions of the *android.Location* package to evaluate how well DocFlow performs across versions. Figure 7 shows the increase in correctly predicted sources and sinks per version. This result indicates that DocFlow correctly adapts to changes in the Android framework, while the missed sources and sinks remain almost constant across versions (reduced for sources). The increased number of sources and sinks aligns with the data shown in Table 7.

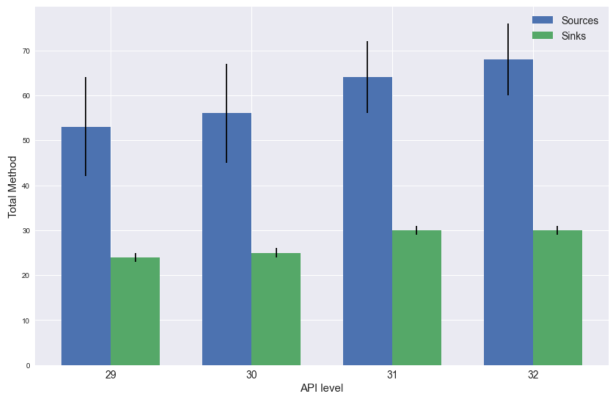


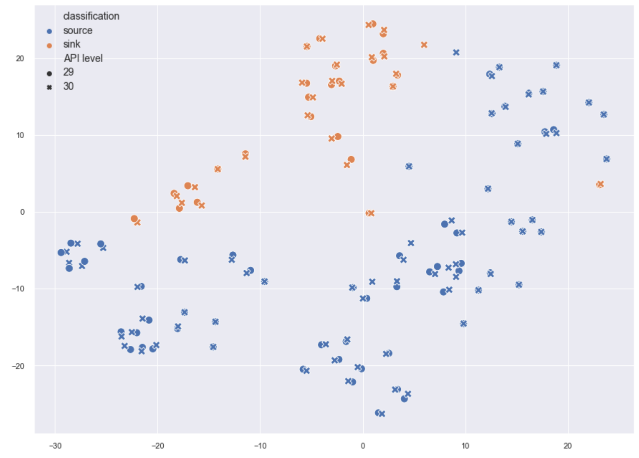
Figure 7: Number of detected sources and sinks across API levels. The sticks indicate the missed sources and sinks.

We analysed a projection of the embeddings in 2-dimensional space using t-SNE [55]. This technique stands out by producing visualisations that reveal structural information for high-dimensional data. Figure 8a illustrates the sources and sink for API levels 29 and 30, while Figure 8b does it for APIs 31 and 32. These figures, complemented by manual analysis of individual points, show how the Embedding Layer models the method's documentation. Firstly, it shows that sources and sinks tend to be separated in the vector space and explains the accuracy of our method classifier module. Secondly, different versions of the same method (including across different releases) are very close in the vector space. As can be seen in Figures 8a and 8b, most projections are done in pairs (corresponding to either the same method from different API levels or the same method with a different signature). The findings demonstrate that DocFlow has the ability to identify sensitive methods across different versions and effectively captures the semantic modifications in the documentation. These capabilities help in creation of reliable taint specifications for newer versions of Android.

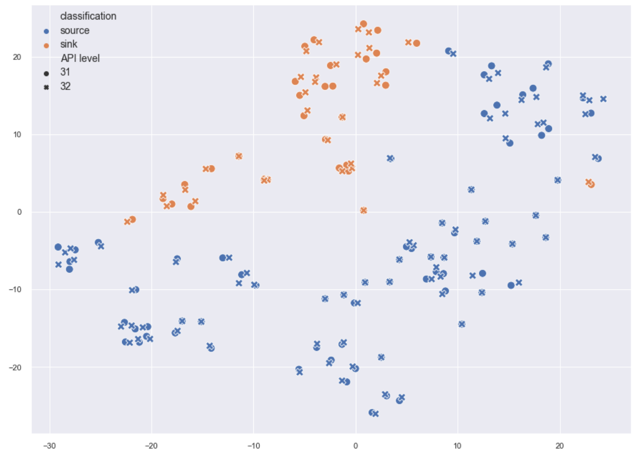
5 DISCUSSION

Our work generates taint specifications by classifying methods as sources and sinks. Security specifications can include other methods types, e.g., callback and sanitisers. Piskachev *et al.* [46] use a broader definition of *Security-Relevant Methods* to include methods types such as sanitisers and authentication. They modified SuSi to include more general features that can be used for other Java-based projects. DocFlow is flexible regarding the method types and can be easily adapted to meet new requirements. One example is Table 6, where DocFlow detected callback methods with high precision.

Design Decisions. DocFlow is parametric in the embedding and classification algorithms. We tested it with Word2Vec [37] and Universal Sentence Encoder [10], but the overall results were better with Sentence-BERT. Instead of a transfer learning approach,



(a) API 29 and API 30



(b) API 31 and API 32

Figure 8: 2D-projection of embeddings for sources & sinks from Location package for API 29-32. Spilt to aid readability.

other works [38, 15, 51] use a fine-tuning approach for the classification task. This approach adds an output layer after calculating the embeddings and retrains the whole network, optionally fixing the weights of one or more layers. Peters *et al.* showed that both approaches produce similar results in most cases [43]. Our transfer learning based approach can easily reuse the embeddings for other tasks, such as semantic search, while allowing fine-tuning.

Documentation Quality. `DocFlow` assumes that Android APIs are well documented in terms of completeness and quality as Android is a popular platform with millions of developers worldwide. However, short or missing descriptions methods can affect the accuracy of our approach. We use Format G (Table 1) to quantify the effect of missing or short descriptions. The sources/sinks detection rate for the libraries are affected by a margin of 3-9 %. Even though this experiment overestimate the number of methods with missing description (all methods), the results show that `DocFlow` can predict sources and sinks in the absence of some information. This is likely because of good coding practices such as representative names complemented by the semantic extracted from the class description and method signature. Recently, Liu *et al.* studied the problem of *silently evolved methods* [30], whose behaviour changes without an update of its documentation. This can introduce noise in our framework. However, the differences are unlikely to change the method's semantic, as Android APIs need to be compatible with previous versions. An extension of our work can model this situation by clustering semantically close APIs through their embedding before feeding the classifier. Zhang *et al.* showed that this approach is effective at slowing down the ageing of learning-based malware detectors [62]. Therefore, `DocFlow` is not suited for all types of projects and focuses on well-documented projects. Still, `DocFlow` adapts well to small changes in the documentation, as shown in the evaluation.

Deprecation and Generalisability. `DocFlow` ignores methods with deprecation tags but it can easily classify the replacement for the deprecation if they are linked in the description of a deprecated method. `DocFlow` achieves good performance across time by analysing different versions of the Location package. However, this may not generalise to other framework packages or Google libraries. This limitation is associated with the complexity of manual validation, as we need to produce labels for each package. Moreover, the labelling process may add errors to the evaluation. We address this problem by cross-labelling the dataset with three experts.

Generative models. In this work, we have not experimented with popular generative LLMs such as ChatGPT[39]. However, we feel that integration with such models can potentially achieve better contextual representation of API documentation as these models are multi-modal and collect information about APIs from a variety of sources. Having said that, there are significant overheads in the Prompt Engineering that needs to be done to make these LLMs useful for downstream tasks [31]. Another disadvantage is that these models are usually trained on a corpus that is a few months old. This makes the Prompt Engineering for our task more complex, as we need to provide intelligence on a new Android release.

6 RELATED WORK

NLP techniques have been widely used to study issues in apps such as privacy policies [38, 26, 63, 35], permissions [40, 32], and app descriptions [24, 48]. There is a growing body of work that brings together Natural Language Processing (NLP) and Software Engineering (SE) for tasks such as software analysis and security, commonly known as *Dual-channel or Bimodal Software Engineering* [9]. Recent advances in this area have introduced novel control and data-flow analyses guided by identifiers from the program [14, 41]. The focus of our research is *Dual-Channel Software Security.* We apply NLP to seed taint analysis by auto-identification of sources and sinks from software documentation.

Sensitive methods have been detected using Machine Learning [5, 46] or other statistical approaches [33]. One limitation of these works is that they rely on code analysis for feature extraction, but this is not always possible (e.g., Google Play Services libraries). For instance, SuSi extracts 144 features from each application before feeding the feature vectors to an SVM classifier. Obtaining the Android framework code is not straightforward as the Android jar file is shipped with stub methods, and bespoke scripts are required to extract code for different versions due to constant changes in the Android architecture. In contrast, our approach automatically extracts vector representations using sentence embeddings from Android API documentation, which is publicly available.

A different approach proposes a query specification language for taint-flows definition [47]. This approach requires input from developers to specify taint-flow queries, while `DocFlow` takes a different approach and analyses software documentation. Seldon is a semi-supervised method for inferring taint analysis specifications [12], and USpec [18] is an unsupervised tool for discovering aliasing specifications of APIs. Both tools learn from a large dataset of programs. We focus on API documentation instead of programs.

One of the first uses of API documentation was to extract models of APIs in Java libraries [61]. Text processing of API documentation has also been used to identify functionality, structure and quality of APIs [21]. Xie *et al.* proposed using verbs phrases to close the usability gap between API documentation and user queries [59]. Treude *et al.* showed how insights from API documentation can be improved with those from other artifacts such as discussion forums [54]. The recent advent of LLMs has opened up opportunities for analysis of mixed-text that describe APIs. For instance, Huang *et a.* demonstrated the application of LLMs to extract APIs and their relations from large sentences of unstructured text, such as those found on discussion forums [27].

7 CONCLUSION

In this paper, we proposed a novel approach to generate taint specifications. We showed that software documentation contains rich semantic information useful for inferring security and other semantic properties of methods. Security analysts can use `DocFlow` to find methods that are semantically similar across versions of the same platform. This can help them to keep up to date with the constant updates and changes to APIs in Android, and Google Play Services in particular. We believe that our work will help the community to obtain better taint specifications that will result in more robust and complete security and privacy analysis of Android applications. Our approach removes the need to analyse code to extract taint specifications, which is particularly convenient when source code is not available or complete. This will encourage extension of taint analysis to API calls made to closed-source software libraries.

8 ACKNOWLEDGEMENTS

This research has been partially sponsored by the Engineering and Physical Sciences Research Council (EPSRC) and the UK government as part of the Centre for Doctoral Training (CDT) in Cyber Security for the Everyday at Royal Holloway, University of London (EP/P009301/1 and EP/W015927/1).

The authors also thank Dr. Helena Gómez for her valuable comments and the computing resources provided through DGAPA-UNAM PAPIIT project number TA101722.

REFERENCES

[1] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Association for Computing Machinery, New York, NY, USA, 143–153. ISBN: 9781450369954. DOI: 10.1145/3359591.3359735.

[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51, 4, 1–37.

[3] Douglas G Altman. 1990. *Practical statistics for medical research*. CRC press.

[4] Zaid Alyafeai, Maged Saeed AlShaibani, and Irfan Ahmad. 2020. A survey on transfer learning in natural language processing. *arXiv preprint arXiv:2007.04239*.

[5] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2013. Susi: a tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013*, 114, 108.

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49, 6, 259–269.

[7] Shweta Bhandari, Wafa Ben Jaballah, Vineeta Jain, Vijay Laxmi, Akka Zemmari, Manoj Singh Gaur, Mohamed Mosbah, and Mauro Conti. 2017. Android inter-app communication threats and detection techniques. *Computers Security*, 70, 392–421.

[8] Paolo Calciati, Konstantin Kuznetsov, Xue Bai, and Alessandra Gorla. 2018. What did really change with the new release of the app? In *Proceedings of the 15th International Conference on Mining Software Repositories*, 142–152.

[9] Casey Casalnuovo, Earl T. Barr, Santanu Kumar Dash, Prem Devanbu, and Emily Morgan. 2020. A theory of dual channel constraints. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results* (ICSE-NIER '20). Association for Computing Machinery, Seoul, South Korea, 25–28. ISBN: 9781450371261. DOI: 10.1145/3377816.3381720.

[10] Daniel Cer et al. 2018. Universal sentence encoder. *arXiv preprint arXiv:1803.11175*.

[11] Sandeep Chaudhary, Sebastian Fischmeister, and Lin Tan. 2014. Em-spade: a compiler extension for checking rules extracted from processor specifications. *ACM SIGPLAN Notices*, 49, 5, 105–114.

[12] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin Vechev. 2019. Scalable taint specification inference with big code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2019). Association for Computing Machinery, Phoenix, AZ, USA, 760–774. ISBN: 9781450367127. DOI: 10.1145/3314221.3314648.

[13] Sumit Chopra, Raia Hadsell, and Yann LeCun. 2005. Learning a similarity metric discriminatively, with application to face verification. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 1. IEEE, 539–546.

[14] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2018. Refinym: using names to refine types. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ESEC/FSE 2018). Association for Computing Machinery, Lake Buena Vista, FL, USA, 107–117. ISBN: 9781450355735. DOI: 10.1145/3236024.3236042.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[16] Wenbo Ding and Hongxin Hu. 2018. On the safety of iot device physical interaction control. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 832–846.

[17] Shuaike Dong et al. 2018. Understanding android obfuscation techniques: a large-scale investigation in the wild. In *International conference on security and privacy in communication systems*. Springer, 172–192.

[18] Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin Vechev. 2019. Unsupervised learning of api aliasing specifications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2019). Association for Computing Machinery, Phoenix, AZ, USA, 745–759. ISBN: 9781450367127. DOI: 10.1145/3314221.3314640.

[19] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32, 2, 1–29.

[20] Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76, 5, 378.

[21] Davide Fucci, Alireza Mollaalizadehbahnemiri, and Walid Maalej. 2019. On using machine learning to identify knowledge in api reference documentation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ESEC/FSE 2019), 109–119.

[22] Google. 2023. Android blog. Retrieved Feb. 15, 2023 from https://www.blog.google/products/android/android-12-beta/.

[23] Google. 2023. Android contributing page. Retrieved Feb. 15, 2023 from https://source.android.com/docs/setup/contribute/code-style.

[24] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking app behavior against app descriptions. In *Proceedings of the 36th international conference on software engineering*, 1025–1035.

[25] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 631–642.

[26] Hamza Harkous, Kassem Fawaz, Rémi Lebret, Florian Schaub, Kang G Shin, and Karl Aberer. 2018. Polisis: automated analysis and presentation of privacy policies using deep learning. In *27th USENIX Security Symposium (USENIX Security 18)*, 531–548.

[27] Qing Huang, Yanbang Sun, Zhenchang Xing, Min Yu, Xiwei Xu, and Qinghua Lu. 2023. Api entity and relation joint extraction from text via dynamic prompt-tuned language model. *ACM Transactions on Software Engineering and Methodology*. Just Accepted.

[28] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

[29] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. 2021. A systematic review of api evolution literature. *ACM Computing Surveys (CSUR)*, 54, 8, 1–36.

[30] Pei Liu, Li Li, Yichun Yan, Mattia Fazzini, and John Grundy. 2021. Identifying and characterizing silently-evolved methods in the android api. in 2021 ieee/acm 43rd international conference on software engineering: software engineering in practice (icse-seip 2021). (2021).

[31] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: a systematic survey of prompting methods in natural language processing. 55, 9, Article 195, 35 pages.

[32] Xueqing Liu, Yue Leng, Wei Yang, Wenyu Wang, Chengxiang Zhai, and Tao Xie. 2018. A large-scale empirical study on android runtime-permission rationale messages. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 137–146.

[33] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. 2009. Merlin: specification inference for explicit information flow problems. *SIGPLAN Not.*, 44, 6, 75–86. DOI: 10.1145/1543135.1542485.

[34] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An empirical study of api stability and adoption in the android ecosystem. In *2013 IEEE International Conference on Software Maintenance*, 70–79. DOI: 10.1109/ICSM.2013.18.

[35] Nuhil Mehdy, Casey Kennington, and Hoda Mehrpouyan. 2019. Privacy disclosures detection in natural-language text through linguistically-motivated artificial neural networks. In *International Conference on Security and Privacy in New Computing Environments*. Springer, 152–177.

[36] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

[37] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26.

[38] Preksha Nema, Pauline Anthonysamy, Nina Taft, and Sai Teja Peddinti. 2022. Analyzing user perspectives on mobile app privacy at scale. In *Proceedings of the 44th International Conference on Software Engineering*, 112–124.

[39] OpenAI. 2023. Gpt-4 technical report. (2023). arXiv: 2303.08774 [cs.CL].

[40] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. {Whyper}: towards automating risk assessment of mobile applications. In *22nd USENIX Security Symposium (USENIX Security 13)*, 527–542.

[41] Profir-Petru Pârtachi, Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2020. Flexeme: untangling commits using lexical flows. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ESEC/FSE 2020). Association for Computing Machinery, Virtual Event, USA, 63–74. ISBN: 9781450370431. DOI: 10.1145/3368089.3409693.

[42] Profir-Petru Pârtachi, Santanu Kumar Dash, Christoph Treude, and Earl T Barr. 2020. Posit: simultaneously tagging natural and programming languages. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 1348–1358.

[43] Matthew E Peters, Sebastian Ruder, and Noah A Smith. 2019. To tune or not to tune? adapting pretrained representations to diverse tasks. *arXiv preprint arXiv:1903.05987*.

[44] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the seventh european workshop on system security*, 1–6.

[45] Marco Piccioni, Carlo A Furia, and Bertrand Meyer. 2013. An empirical study of api usability. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 5–14.

[46] Goran Piskachev, Lisa Nguyen Quang Do, and Eric Bodden. 2019. Codebase-adaptive detection of security-relevant methods. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 181–191.

[47] Goran Piskachev, Johannes Späth, Ingo Budde, and Eric Bodden. 2022. Fluently specifying taint-flow queries with fluenttql. *Empirical Software Engineering*, 27, 5, 1–33.

[48] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. Autocog: measuring the description-to-permission fidelity in android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 1354–1365.

[49] Nils Reimers. 2023. Sbert pretrained models. Retrieved Feb. 15, 2023 from https://www.sbert.net/docs/pretrained_models.html.

[50] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*.

[51] Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. 2019. How to fine-tune bert for text classification? In *China national conference on Chinese computational linguistics*. Springer, 194–206.

[52] Marcos Tileria and Jorge Blasco. 2022. Watch over your tv: a security and privacy analysis of the android tv ecosystem. *Proceedings on Privacy Enhancing Technologies*, 3, 692–710.

[53] Marcos Tileria, Jorge Blasco, and Guillermo Suarez-Tangil. 2020. Wearflow: expanding information flow analysis to companion apps in wear os. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 63–75.

[54] Christoph Treude and Martin P. Robillard. 2016. Augmenting api documentation with insights from stack overflow. In *Proceedings of the 38th International Conference on Software Engineering* (ICSE '16), 392–403.

[55] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *Journal of machine learning research*, 9, 11.

[56] Adina Williams, Nikita Nangia, and Samuel Bowman. 2018. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. Association for Computational Linguistics, New Orleans, Louisiana, 1112–1122. http://aclweb.org/anthology/N18-1101.

[57] René Witte, Qiangqiang Li, Yonggang Zhang, and Juergen Rilling. 2008. Text mining and software engineering: an integrated source code and document analysis approach. *Iet Software*, 2, 1, 3–16.

[58] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. 2015. Dase: document-assisted symbolic execution for improving automated software testing. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE, 620–631.

[59] Wenkai Xie, Xin Peng, Mingwei Liu, Christoph Treude, Zhenchang Xing, Xiaoxin Zhang, and Wenyun Zhao. 2020. Api method recommendation via explicit matching of functionality verb phrases. In (ESEC/FSE 2020), 1015–1026.

[60] Wenpeng Yin, Jamaal Hay, and Dan Roth. 2019. Benchmarking zero-shot text classification: datasets, evaluation and entailment approach. *arXiv preprint arXiv:1909.00161*.

[61] Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao, and Feng Qin. 2016. Automatic model generation from documentation for java api functions. In (ICSE '16), 380–391.

[62] Xiaohan Zhang, Mi Zhang, Yuan Zhang, Ming Zhong, Xin Zhang, Yinzhi Cao, and Min Yang. 2022. Slowing down the aging of learning-based malware detectors with api knowledge. *IEEE Transactions on Dependable and Secure Computing*.

[63] Sebastian Zimmeck, Peter Story, Daniel Smullen, Abhilasha Ravichander, Ziqi Wang, Joel R Reidenberg, N Cameron Russell, and Norman Sadeh. 2019. Maps: scaling privacy compliance analysis to a million apps. *Proc. Priv. Enhancing Tech.*, 2019, 66.