Negative Results of Fusing Code and Documentation for Learning to Accurately Identify Sensitive Source and Sink Methods
An Application to the Android Framework for Data Leak Detection

Jordan Samhi*, Maria Kober‡§, Abdoul Kader Kabore*, Steven Arzt†, Tegawendé F. Bissyandé*, Jacques Klein*

* SnT, University of Luxembourg, firstname.lastname@uni.lu

† Fraunhofer Institute for Secure Information Technology, Darmstadt, Hessen, Germany, steven.arzt@sit.fraunhofer.de

‡ mariakober.research@gmx.com

Abstract—**Apps on mobile phones manipulate all sorts of data, including sensitive data, leading to privacy-related concerns. Recent regulations like the European GDPR provide rules for the processing of personal and sensitive data, like that no such data may be leaked without the consent of the user.**

Researchers have proposed sophisticated approaches to track sensitive data within mobile apps, all of which rely on specific lists of sensitive SOURCE and SINK API methods. The data flow analysis results greatly depend on these lists' quality. Previous approaches either used incomplete hand-written lists that quickly became outdated or relied on machine learning. The latter, however, leads to numerous false positives, as we show.

This paper introduces CODOC, a tool that aims to revive the machine-learning approach to precisely identify privacy-related SOURCE and SINK API methods. In contrast to previous approaches, CODOC uses deep learning techniques and combines the source code with the documentation of API methods. Firstly, we propose novel definitions that clarify the concepts of sensitive SOURCE and SINK methods. Secondly, based on these definitions, we build a new ground truth of Android methods representing sensitive SOURCE, SINK, and NEITHER (i.e., no source or sink) methods that will be used to train our classifier.

We evaluate CODOC and show that, on our validation dataset, it achieves a precision, recall, and F_1 score of 91% in 10-fold cross-validation, outperforming the state-of-the-art SUSI when used on the same dataset. However, similarly to existing tools, we show that in the wild, i.e., with unseen data, CODOC performs poorly and generates many false positive results. Our findings, together with time-tested results of previous approaches, suggest that machine-learning models for abstract concepts such as privacy fail in practice despite good lab results. To encourage future research, we release all our artifacts to the community.

I. INTRODUCTION

Given the ubiquity of mobile devices nowadays and the proliferation of apps installed and used by end users, Android apps' analysis has become a common topic in software engineering research. Numerous approaches have been proposed to check security properties, detect malicious code, and detect program bugs. These approaches leverage techniques such as dynamic analysis [56], [40], [14], static analysis [31], [48], [23], [58], [38], [45], [4], [20], [46], or both (i.e., hybrid analyses) [8], [7].

The previously mentioned analysis approaches usually consist of several specific techniques that are applied to apps. One of them is taint analysis, which checks whether data obtained from a given SOURCE method (or any kind of data derived from it, e.g., after some computation) is passed to a SINK method. In the context of Android-based privacy research, a SOURCE is an API method that provides privacy-sensitive data. A SINK is an API method that writes data to the outside, e.g., via the network.

The need for sensitive SOURCE and SINK lists is ubiquitous in taint analysis. Indeed, the literature is full of approaches and techniques that set up privacy strategies building on sensitive API methods that allow retrieving sensitive data and/or API methods allowing to expose this kind of data. The range of approaches relying on sensitive SOURCE and SINK methods that could benefit from a complete and precise list of SOURCE and SINK methods is large: *sensitive data leak detection* [31], [19], [14], [4], [25], [47], [49], *Android component leak detection* [27], *dynamic policy enforcement* [41], *malware detection* [67], [51], [43], *hidden behavior detection* [39], [66], *inter-app communication analysis* [10], [29], *component hijacking vulnerabilities detection* [34], [65], the *uncovering of run-time sensitive values* [42], as well as *GDPR compliance checks* [17], [16].

In its public-facing API – i.e., methods contained in the official online documentation for Android –, Android 11[1] contains more than 34 000 methods. However, it has been shown [30] that developers have access to many more methods inside the Android framework that are not directly available in the public-facing API (e.g., hidden to developers using the annotation "@hide"). In Android 11, for example, more than 210 000 methods are available to developers in total, e.g., through the use of reflection[2]. These large numbers render a manual classification of sources and sinks infeasible. Furthermore, additional methods are added in every new release. New frameworks, such as Android Auto or Chromecast, bring

[1]API level 30, which was released in September 2020

[2]Note that reflection can also be used to make private methods accessible since Android does not provide a Java security manager.

in new methods, and thus potentially new sources and sinks as well. Therefore, automatic approaches for identifying sources and sinks are needed. Several approaches have been proposed in the literature to solve this problem [19], [36], [14]. SuSi [3], which is based on supervised machine-learning, is currently one of the most popular approaches and the state-of-the-art [4], [60], [39]. To the best of our knowledge, it is also the most comprehensive and state-of-the-art approach for deriving lists of sources and sinks from frameworks like Android.

However, the sources list generated by SuSi is neither precise, nor specific for privacy analysis. As we show in Section IV, SuSi classifies methods as sources, even though they are clearly irrelevant for privacy analysis. Secondly, SuSi relies on technical categories (network information, unique identifiers, etc.) to structure its output. Selecting all categories that could be relevant for privacy analysis leads to a large number of irrelevant APIs being selected as well. These observations are not surprising because SuSi makes no upfront assumptions on the sensitivity of the sources yielded.

Further, some methods are misclassified entirely by SuSi, e.g., the method `getScrollIndicatorBounds` of the `android.view.ViewGroup` class is categorized as a source in the "SMS_MMS" category. As we show in Section VI-E, these issues lead to a profusion of false positives for any data flow tracker that relies on SuSi's source/sink lists. Several works in the literature [57], [24], [35] have come to similar conclusions. Luo et al. show that SuSi's sources list leads to a false positive rate of almost 80% while trying to detect sensitive data leaks [35]. Further, in accordance with our own findings, they state the following regarding the sources yielded by SuSi:

> the root cause of these false positives is that their sources [...] are actually inappropriate, i.e., they do not return **sensitive data**.

We note that SuSi's evaluation in the original paper [3] does not highlight these issues, and that SuSi performs well on its training data and select examples. However, in the real-world, the false positive rate is much higher.

To the best of our knowledge, no other approach has tackled the problem of automatically identifying sources and sinks in Android since the release of SuSi in 2014. In other words, SuSi is still the most relevant approach despite its shortcomings. Since the problem remains highly relevant and unsolved regarding sensitive data, we attempted to improve the SuSi approach. The SuSi features for the supervised machine learning rely entirely on static code analysis on the Android bytecode implementation (the Android platform JAR file). SuSi considers individual properties as features, such as method names, parameter types, or method/class modifiers which do not capture the entire semantic of the code.

Our approach, that we named CoDoC, on the other hand, captures the entire semantics of a given method by taking its' complete source code into account. Additionally, CoDoC also considers the JavaDoc documentation of the Android API. We observed that the Android documentation, which is fairly extensive for most classes and methods, provides enough guidance to the developer to correctly use the API. We then assumed that analyzing this documentation would also help in more precisely discovering sensitive SOURCE and SINK methods in the Android API.

Lastly, CoDoC is an attempt to incorporate the ground-breaking advances that have been made in text and code embedding [2], [44] and, thus, machine learning since 2014 when SuSi was published.

While our evaluation shows that CoDoC outperforms SuSi in the lab, CoDoC's real-world performance, likewise, is still lacking. Manually inspecting the SOURCE and SINK methods identified in a set of previously unseen API methods from the Android framework reveals many false positives. We, therefore, argue that even adding documentation and improving the machine learning techniques do not solve the problem of accurately identifying privacy-related sources and sinks in Android. Even with more and better training data and careful optimization of the training, the overall goal remains elusive. We argue that the semantic gap between an individual API method (code or documentation) and an abstract concept such as user privacy are unlikely to be closed by supervised machine learning. Instead, novel approaches are necessary.

Further, we call for a more careful evaluation of machine learning results. In the lab studies based on 10-fold cross-validation, SuSi is sufficient, and CoDoC is even better. Still, on real-world data, i.e., previously unseen methods from the Android framework, both fail to meet expectations.

Overall, we make the following contributions:

- we propose CoDoC: a novel, fully-automated, deep-learning-based approach to detect sensitive SOURCE and SINK methods in the Android framework based on API method source code and documentation;
- we release a new ground-truth of methods labeled as sensitive SOURCE, SINK, or NEITHER;
- we evaluate CoDoC and show that it outperforms the state-of-the-art SuSi on a small evaluation dataset, reaching a precision, recall, and F_1 score of 91% in the lab;
- we apply CoDoC on public methods from the Android framework and show that, likewise SuSi, it yields a high rate of false positives;
- We release our open-source prototype CoDoC to the community and all the artifacts used in our study at:

https://github.com/JordanSamhi/CoDoC

II. BACKGROUND

In this section, we provide the reader with context for the work presented in this paper.

A. Taint Analysis

Taint analysis is a particular dataflow analysis that tracks data through the control flow graph of a program. If a variable V is assigned the return value of a specific function, like a SOURCE method, it becomes tainted. If a tainted value is assigned to some other variable W, this variable W gets tainted as well. The same applies if W is assigned the result of some operation on an already tainted variable V. In other words:

the taint is propagated. When a tainted variable is passed to a `SINK` function as a parameter, a leak is reported, as the value derived from the `SOURCE` reached a `SINK`. In the case of data leak detection in the context of privacy analysis, an example of a `SOURCE` in the Android Framework is `getImei()` and an example of a `SINK` is `sendTextMessage()`.

B. Text Embedding

Our work relies on methods' source code and documentation to train a machine learning model and infer `SOURCE` and `SINK` methods. In order to be processed by machine learning algorithms, these textual representations need to be transformed (*embedded*) into numerical representations, i.e., numerical vectors. In this section, we briefly describe two state-of-the-art techniques for this transformation, namely Sentence-BERT [44] and Code2Vec [2].

1) Sentence-BERT: Method documentation embedding requires efficient natural language processing techniques. SENTENCE-BERT is a modified and more computationally efficient version of the well-known BERT neural network [13]. It relies on siamese and triplet network structures to obtain meaningful sentence embeddings.

2) CODE2VEC: Similarly to natural language embedding, making predictions from source code requires code embedding to have a homogeneous representation of different source code inputs. CODE2VEC embeds Java methods to predict method names. Methods are transformed into ASTs (Abstract Syntax Trees) to construct path representations between different leaf nodes. Then, using the attention mechanism [5], bag of path-contexts are aggregated into a single vector that represents the method body.

III. DEFINITIONS

In the literature, there is no consensus on the definitions of *sensitive* `SOURCE` and `SINK` methods, which leads to a lack of clarity in papers related to taint analysis. As described in Section II-A, taint analysis tracks the flow of data from a given `SOURCE` to a given `SINK`, no matter the type of data. However, in most of the papers, the authors mix *sensitive* `SOURCE` with `SOURCE`, which makes taint analysis appear as tracking sensitive data, which is not always the case. Tracking sensitive data is an instance of the more general task of tracking data.

To cope with this problem and provide state-of-the-art approaches that aim at tracking sensitive data with clear terms, we propose the following definitions:

Definition 1 (Data). Any value or reference to a value.

Definition 2 (Composite Data). Any structure composed of multiple Data (e.g., an object).

Definition 3 (Sensitive Data). Any data or composite data that holds private value(s) that:

- can identify users, i.e., usernames and personally identifying data like email address or name
- can identify the device, i.e., unique device identifiers
- are related data to personal information (of the phone user), e.g., photographs and files, phone calls, SMS

- represent data owned by users holding information about other users, e.g., contacts and phone lists, emails, etc.
- represent environment and sensor information, including geolocation data, camera, and microphone.

Definition 4 (Sensitive `SOURCE`). A function that returns a Sensitive Data. Note that functions that return constant values are never sensitive sources.

Definition 5 (`SINK`). A function that sends out one or more Data, Composite Data, or values derived from a Data or a Composite Data from the application memory space. There is no notion of sensitivity for sinks. The nature of the data (more precisely: the `SOURCE` from which the data was originally obtained) passed to the sink determines whether a leak of sensitive data occurs.

IV. MOTIVATION

Tracking sensitive data within Android apps is of high interest since it is used in numerous security-related approaches [62], [50], [61], [4], [31] and part of legal compliance, e.g., according to the GDPR. Therefore, there is a need to provide analysts and researchers with sources and sinks lists that precisely enclose sensitive data (cf. Definition 6).

Android API: As briefly explained in Section I, the number of public and documented API methods intended for use by Android developers amounts to tens of thousands and increases with every new API version (see Figure 1). Still, even identifying all sources and sinks in these documented public API methods is not sufficient, as developers also call methods not intended for direct use, yet present in the Android framework [30]. With tens of thousands of public API methods and hundreds of thousands of overall methods, manual classification for every new release is obviously infeasible.

Therefore, automated solutions are needed to produce sensitive sources and sinks lists for every new release. In the following, we explain why the existing state of the art is inappropriate for this task.
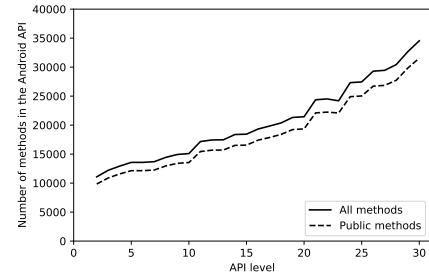

Fig. 1: Number of methods in the public-facing Android API by API level

Problem with the existing state of the art: The state-of-the-art approach SUSI [3] uses machine learning to automatically classify Android `SOURCE` and `SINK` methods. However, it has been shown several times [57], [24], [35] that SUSI's lists are inappropriate since it is not specific to a particular analysis like tracking sensitive data. Therefore it produces many false positives and forces analysts to manually select appropriate `SOURCE` and `SINK` methods.
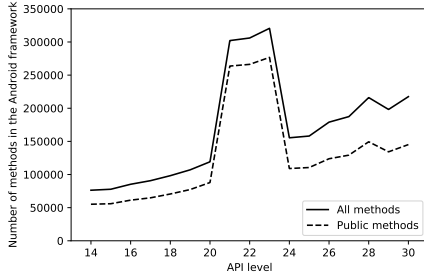
Fig. 2: Number of methods in the entire Android framework code by API level

```
1  public void method() {
2      int p = 7;
3      int q = 4;
4      Rational r = new Rational(p, q);
5      int value = r.intValue();
6      SmsManager s = SmsManager.getDefault();
7      s.sendTextMessage("0", null, value, null, null);
8  }
```

Listing 1: Example of SUSI non-sensitive data leak

Consider the example in Listing 1. In line 4, a `Rational` object is created from two integers. In line 5, the integer representation of the `Rational` is retrieved using the method `intValue()` and stored in variable `value`. Eventually, this `value` is sent out of the device via SMS. Since SUSI wrongly considers the method `intValue()` as a SOURCE and the method `sendTextMessage()` as a SINK, a taint analysis based on SUSI will report a leak. This leak, however, is irrelevant in the context of security and privacy as the SOURCE is not sensitive. Thus, analysts will consider it a false positive when aiming to detect sensitive data leaks in Android apps.

We aim to improve upon the state of the art by *producing a more adequate ground truth to train an improved machine learning model based on method documentation and source code*, unexplored until now.

V. APPROACH

In this paper, we aim to automatically identify sensitive SOURCE and SINK methods in the Android framework among all API methods available to developers (i.e., $> 210\,000$ in Android 11) using supervised machine learning. Figure 3 shows an overview of our approach.

Similar to SUSI, we build our training data by manually labeling Android methods. We consider a method as a sensitive SOURCE if it matches Definition 4[3], a SINK if it matches Definition 5, and NEITHER otherwise. In contrast to SUSI, our approach then uses features extracted from the code as well as the documentation to train a machine-learning model on our ground truth. This is a key difference to SUSI, which only uses distinct properties extracted from parts of the code, such as method names and parameters or class and method modifiers. Further, SUSI completely disregards the method's documentation, which CODOC includes. Moreover, as we rely on the entire source code of a method, we are able to capture the entire semantic of it.

[3]in the following, when we refer to SOURCE, we mean "Sensitive" SOURCE

We finally use our generated model to predict new sensitive SOURCE and SINK methods from the Android framework methods. We explain the individual steps in the following sections. First, we give details about our manual labeling of Android methods in Section V-A. Then, in Section V-B, we explain what features were chosen for training our models. Lastly, in Section V-C, we explicit on what machine learning models our approach builds upon.

A. Manual Labeling

Since our approach relies on supervised machine learning algorithms, labeled data is needed to train our model. However, manual labeling is a challenging and time-consuming task, especially if we randomly chose methods from the Android framework to label one by one. Further, finding a SOURCE or a SINK through random picking is highly unlikely as most methods in the Android framework are neither. Therefore, we opted for a better strategy divided into three phases:

Phase 1: The authors first constituted a *golden dataset* based on well-known methods that return sensitive data described in the literature [18], [4], [14], [39], [15], [64]. These methods span across classes such as: `TelephonyManager`, `AccountManager`, `LocationManager`, `SmsMessage`, or `SensorManager`. This step yielded an initial set of 39 SOURCE and 35 SINK methods.

Phase 2: As explained in Section IV, SUSI can generate lists of sources and sinks (from its own definition, i.e., not restricted to sensitive methods). We applied SUSI on Android API version 30 to generate additional pre-selected input that we hand-labeled as training data for CODOC. As described previously, randomly picking methods from the Android API would mostly lead to methods that are neither sources nor sinks. Therefore, we opted to focus hand-labeling on methods that are more likely to be relevant. We concatenated the list of sources and the list of sinks computed by SUSI to obtain a full list of methods M that SUSI considers relevant. Note that we did not manually post-process methods that SUSI classified as neither a source or a sink. Two of the authors then applied manual post-processing as follows.

One author started from the top of each list manually classify each method in the respective list. The other author started from the bottom of each list with same task. For each method $m \in M$, the authors independently read the documentation and the source code to be able to classify it as a SOURCE, a SINK, or NEITHER based on the definitions described in Section III.

This step leads to three lists per author: a ① SOURCE list; a ② SINK list; and ③ a NEITHER list.

Phase 3: The third phase aimed at calculating the inter-rater agreement between the data labeled by both authors in phase 2. We use inter-author agreement as a quality measure for our hand-labeled dataset, which is later used for training the CODOC classifier. To do so, both authors together alternately verified the results of each other and noted the agreement. Eventually, a Cohen's Kappa coefficient [11] was computed to evaluate the level of agreement. Due to the clear definitions
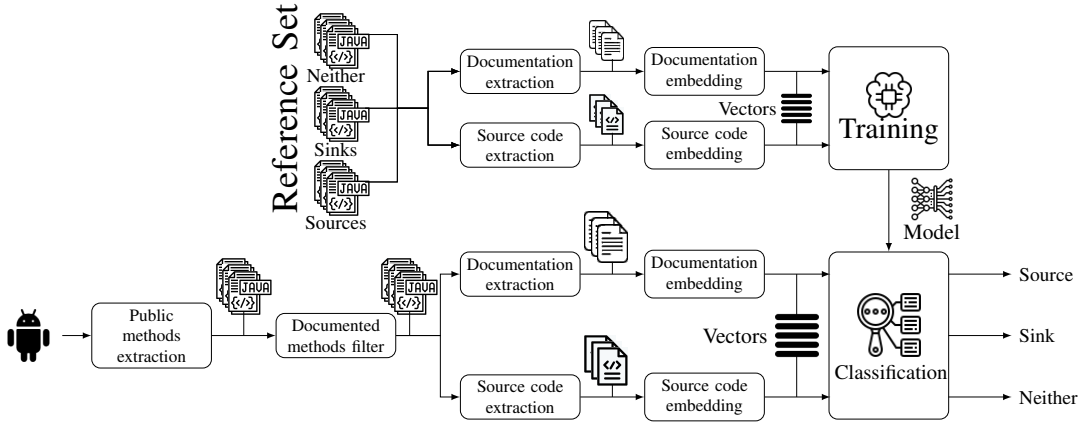
Fig. 3: Overview of the CoDoC approach

given in Section III, both authors reached a perfect agreement level of 1. In phase 2, both authors classified 192 methods as sources, resulting in a total of 231 SOURCE methods from phase 1 and phase 2; 95 methods as sinks, resulting in a total of 130 SINK methods; as well as 654 NEITHER methods. In total, we have a set of 1015 API methods for model training.

B. Data collection and representation

Our approach relies on two different types of input: ① the documentation of a method; and ② the source code of a method. This section explains how these data were gathered and transformed into numerical value vectors.

Data Collection: As an open-source project, the Android source code is directly available on the Internet [4]. We downloaded and parsed it using JAVAPARSER [55] to extract *public* methods that were *documented*, and *implemented* (i.e., concrete methods). A method that is documented means either that: ① the method itself is documented; or ② in one of its parent classes/interfaces is documented. For each method in the so derived dataset, we extracted ① its source code; and ② its documentation. Eventually, our dataset consists of 46 034 methods from the Android framework.

Source Code Representation: Source code must be preprocessed as a piece of textual information before it can serve as input for machine-learning algorithms. In our work, we rely on CODE2VEC [2] (also see ection II). Since the samples have different sizes, they must be transformed into fixed-size numerical vectors. CODE2VEC relies on a neural network that needs to be trained in order to generate source code vectors. As the source code of the Android framework is Java code and the original pre-trained model available in the CODE2VEC project repository[5] has been trained on Java source code as well, we could have used this model. However, since Android code contains platform-specific semantic tokens that cannot be found in regular Java source code (e.g., Activity, BroadcastReceiver, etc.), we decided against this approach. Instead, we trained the model with the source code from the
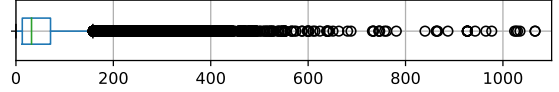

Fig. 4: Distribution of the number of words in the documentation collected per Android method

Android framework to assure that our model properly captures the platform-specific tokens prevalent in Android.

After training the CODE2VEC model with Android framework data, we fed the model with the 46 034 Android methods previously extracted to generate their numerical value vectors. Eventually, 46 034 vectors of size 384 were generated.

Documentation representation: In the same way as the source code, the documentation has to be embedded into fixed-sized numerical value vectors to be fed into machine learning algorithms. We relied on SENTENCE-BERT [44] to generate those vectors. We leave experimentation with other models such as BERT [13] or RoBERTa [32] to future work.

We used SENTENCE-BERT with the "paraphrase-mpnet-base-v2" pre-trained model, which is the one achieving the best performances[6] at the time of writing. Eventually, all documentation of the 46 034 previously gathered methods were converted into 768-value long vectors. The distribution of the number of words in the documentation collected is available in Figure 4. Note that on average, the number of words in the documentation collected is 56, and the median is 32. In future work, we will investigate the effect of text summarizing, i.e., learning from more compact texts.

C. Deep Learning Architecture

Our deep learning model architecture is straightforward and aims at combining documentation and source code vectors into a single representation. The overall architecture is available in Figure 5. Since we are working on two different inputs (i.e., the documentation and the source code) of two different sizes, we decided to rely on two parallel and identical sub-neural networks and combine their output into a single vector that, in turn, is used for a classification task. Each of those

---

[4] https://android.googlesource.com/

[5] https://github.com/tech-srl/code2vec

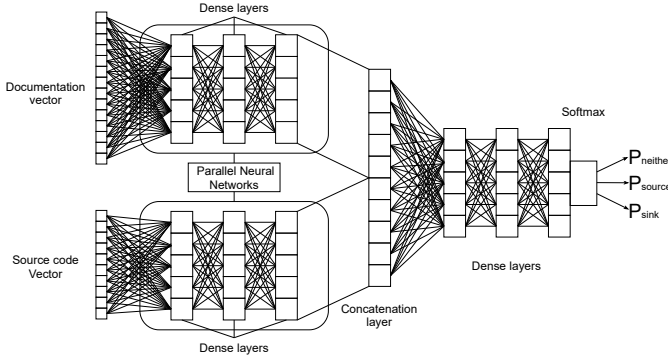[6] https://www.sbert.net/docs/pretrained_models.html

Fig. 5: CODOC neural network architecture.

two parallel networks is built using a stack of three dense [22] layers with ReLU [6] as activation function. They are used for extracting fixed-size features from the two inputs. Thus, the first and the second sub-networks take as input the 768-long documentation vector and the 384-long source code vector, respectively, and provide as output two vectors of size 128 each. Those outputs are combined using a concatenation layer that produces an unique 256-long vector. We use this unique vector for a classification task carried out in 3 additional dense layers. A softmax [52] loss function is considered in the last dense layer in order to perform a multi-class classification, resulting in a classification as SOURCE, SINK, or NEITHER.

VI. EVALUATION

To evaluate CODOC, we address the following research questions:

RQ1: Do documentation and source code features provide complementary input for classification?

RQ2: How does CODOC perform in 10-fold cross-validation and how does it compare with SUSI?

RQ3: Can CODOC make better predictions than SUSI in the wild, i.e., with unseen data?

RQ4: How does CODOC perform on previously unseen methods?

RQ5: How do the source and sink lists created by CODOC and by SUSI compare in data flow analysis?

A. RQ1: Features complementarity

Objective: In this section, we aim at evaluating to what extent both the source code and the documentation are needed to predict sensitive SOURCE and SINK methods. Intuitively, the source code and the documentation should contribute complementary pieces of semantic information.

Experimental Setup: To experimentally evaluate this hypothesis, we run and compare 4 configurations of CODOC:

1) Binary classification with SOURCE and ¬SOURCE
   a) Only documentation
   b) Only source code
2) Binary classification with SINK and ¬SINK
   a) Only documentation
   b) Only source code

Note that we tested these configurations on multiple classifiers, i.e., we exchanged the dense classification layer in Figure 5 with other classifiers. We did so to ensure that RQ1 is answered in-depth and not dependent on a single classification approach. For each binary classification described above, we performed a stratified 10-fold cross-validation [26], [54], and for each iteration we retained the methods that were mis-classified considering only the documentation and well-classified using only the source code, and vice-versa. The results using only code resp. only documentation are available in Table I. We notice that predictions using only documentation yield better results both for source and sink predictions.

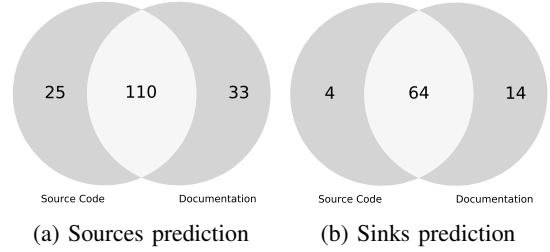(a) Sources prediction   (b) Sinks prediction

Fig. 6: Number of sources and sinks correctly predicted with the KNN algorithm.

Further, we computed the overall number of methods that were mis-classified with the documentation but well-classified with the source code, and vice-versa for sources and sinks and for each classifier. Figures 6a and 6b illustrate the results of this experiment for the KNN algorithm. There we can see that classification with the source code can correctly predict 25 sources that classification with documentation cannot, and 33 vice-versa. In the same way, classification with the source code can correctly predict 4 sinks that classification with documentation cannot, and 14 vice-versa. This shows that, although classification with documentation is better, it misses some samples that classification with source code does not. This shows the need to use both features for our work to improve our model capabilities.

> **RQ1 answer:** Both the documentation and the code independently bring additional information with regard to the other, i.e., several sources/sinks could only be found with the documentation, or only with code features.

B. RQ2: 10-fold cross validation on CODOC's model and comparison with the state-of-the-art SUSI

Objective: In this section, we investigate whether our approach can better identify sensitive SOURCE and SINK methods than the state-of-the-art SUSI approach. We use CODOC with source code and documentation as input. Note that classifying SOURCE and SINK methods is not a binary classification problem since a method can either be: ① a SOURCE; ② a SINK; or ③ NEITHER. Hence, our model relies on a multiclass classification.

	SOURCE prediction										SINK prediction									
	Code					Documentation					Code					Documentation				
	A	P	R	F	K	A	P	R	F	K	A	P	R	F	K	A	P	R	F	K
XGB	0.86	0.74	0.62	0.67	0.58	**0.91**	**0.87**	**0.72**	**0.78**	**0.70**	0.94	0.91	0.58	0.70	0.66	**0.95**	0.91	**0.67**	**0.76**	**0.68**
SVC	0.82	0.61	0.64	0.62	0.49	**0.90**	**0.75**	**0.86**	**0.80**	**0.72**	0.88	0.55	0.69	0.60	0.54	**0.94**	**0.75**	**0.83**	**0.78**	**0.73**
DT	0.80	0.56	0.60	0.57	0.49	**0.85**	**0.66**	**0.66**	**0.66**	**0.52**	**0.88**	**0.53**	**0.54**	**0.53**	0.46	0.87	0.50	0.50	0.48	**0.50**
RF	0.85	0.74	**0.56**	0.62	0.58	0.88	0.92	0.55	0.68	0.58	0.93	0.94	0.51	0.65	0.57	0.92	0.93	0.46	0.60	0.51
GNB	0.81	0.55	0.82	0.65	0.53	0.87	0.66	0.85	0.74	0.65	0.82	0.40	0.77	0.52	0.42	**0.88**	**0.54**	0.76	**0.62**	**0.56**
SGD	0.82	0.61	0.57	0.58	0.50	**0.91**	**0.83**	**0.79**	**0.80**	**0.70**	0.90	0.60	0.65	0.61	0.52	**0.93**	**0.75**	**0.74**	**0.73**	**0.70**
KNN	0.85	0.73	0.56	0.63	0.56	0.89	0.82	0.68	0.74	0.67	0.91	0.66	0.64	0.64	0.60	**0.92**	**0.67**	**0.82**	**0.73**	**0.69**
BDT1	0.85	0.74	0.50	0.59	0.51	0.88	0.86	0.58	0.68	0.58	0.92	0.84	0.49	0.61	0.54	0.91	0.73	0.45	0.55	0.49
BDT2	0.85	0.74	0.50	0.59	0.51	0.88	0.86	0.58	0.68	0.58	0.92	0.84	0.49	0.61	0.54	0.91	0.73	0.45	0.55	0.49
ET	0.86	0.75	**0.56**	0.63	**0.58**	0.87	0.89	0.51	0.64	0.54	**0.94**	**0.95**	**0.60**	**0.73**	**0.67**	0.93	0.93	0.49	0.62	0.57
ADA1	0.84	0.66	0.62	0.63	0.50	0.89	0.78	0.70	0.73	0.70	0.91	0.74	0.58	0.63	0.55	**0.94**	**0.78**	**0.73**	**0.74**	**0.70**
ADA2	0.84	0.66	0.62	0.63	0.50	0.89	0.78	0.70	0.73	0.70	0.91	0.74	0.58	0.63	0.55	**0.94**	**0.78**	**0.73**	**0.74**	**0.70**
GB	0.86	0.72	0.62	0.66	0.57	**0.90**	**0.83**	**0.68**	**0.75**	**0.69**	0.93	0.90	0.52	0.64	0.59	0.92	0.85	**0.53**	0.64	0.59
NN	0.84	0.69	0.56	0.60	0.50	**0.90**	**0.86**	**0.68**	**0.75**	**0.64**	0.90	0.77	0.34	0.45	0.33	0.87	0.00	0.00	0.00	0.00

TABLE I: Results of binary classification on multiple classifiers with code and documentation. (A = Accuracy, P = Precision, R = Recall, F = F_1 score, K = Kappa score)

	Precision	Recall	F_1 score
SOURCE	0.82	0.88	0.85
SINK	0.93	0.87	0.90
NEITHER	0.95	0.93	0.94
Macro Average	0.90	0.89	0.89
Weighted Average	**0.91**	**0.91**	**0.91**

TABLE II: CODOC performances

	Precision	Recall	F_1 score
SOURCE	0.83	0.85	0.84
SINK	0.86	0.71	0.78
NEITHER	0.89	0.91	0.90
Macro Average	0.86	0.82	0.84
Weighted Average	**0.87**	**0.87**	**0.87**

TABLE III: SUSI performances with our ground-truth

CODOC performances: To evaluate our approach, we apply a stratified 10-fold cross-validation and compute the following metrics: ① precision ($\frac{|TP|}{|TP|+|FP|}$); ② recall ($\frac{|TP|}{|TP|+|FN|}$); and ③ F_1 score ($2 \times \frac{precision \times recall}{precision + recall}$), with TP = True Positive, FP = False Positive, and FN = False Negative.

In Table II, we present the results of CODOC. Note that we retain the activation function that yielded the best results for our neural network, i.e., ReLU [1] (we tested the following activation functions: ReLU, sigmoid, tangent, and several combinations of these functions).

Comparison with SUSI: SUSI is not intended to generate *privacy-sensitive* SOURCE and SINK methods, as explained in Section I. Therefore, a direct comparison between our approach CODOC and the pre-trained SUSI would be unfair [12]. We therefore trained SUSI on our own ground truth to evaluate it and compare it with CODOC.

Table III shows the results of a 10-fold cross validation. First, we notice that both CODOC and SUSI independently yield better results for the NEITHER class compared to SOURCE and SINK classes. This is expected since the training data set is highly imbalanced towards NEITHER methods – most of the Android API is not a source or sink. While this imbalance may be relieved using over-/undersampling techniques or class weights, it does not affect the comparative performance of the tools.

Second, CODOC clearly yields slightly better performance than SUSI to classify sensitive SOURCE and SINK methods. CODOC outperforms SUSI by 4 points for the precision, the

Lists	# SOURCE	# SINK
CODOC	15 105	1061
SUSI_ORIGINAL	25 369	5913
SUSI_NEW	12 082	1010

TABLE IV: Number of SOURCE and SINK methods in the lists generated by CODOC and SUSI

recall, and the F_1 score.

> **RQ2 answer:** The performance of our approach CODOC achieves an F_1 score of 91% to identify SOURCE and SINK methods in the Android framework. Furthermore, on the same training set, CODOC achieves a slightly better score than the state-of-the-art SOURCE and SINK classifier SUSI.

C. RQ3: CODOC *and* SUSI *comparison in the wild*

Objective: This research question aims to compare which methods are identified as sensitive SOURCE or SINK methods by CODOC, and which ones by SUSI. To do so, we check the (non-)overlap to judge the performance of both approaches outside of the 10-fold cross-validation. We apply SUSI and CODOC on the full Android SDK, containing mostly non-labeled methods.

Experimental setup: To compare CODOC against SUSI, we generate the following lists of SOURCE and SINK methods (sizes shown in Table IV):

1) **CODOC**: the lists generated by CODOC on Android 30 (API version 11), trained on our ground truth.
2) **SUSI_ORIGINAL**: the lists generated by SUSI on a more recent version of Android (i.e., 30), trained on SUSI's original ground truth.
3) **SUSI_NEW**: the lists generated by SUSI on a more recent version of Android (i.e., 30), trained on our ground truth.

We first analyze the overlap between SUSI's and CODOC's lists in Figure 7. We notice that for both SUSI_ORIGINAL and SUSI_NEW, and both SOURCE and SINK methods, CODOC and SUSI do not have much in common.

We manually curated 6 data sets for further inspection:

1) We select 100 non-sensitive SOURCE methods classified by SUSI_ORIGINAL and compare CODOC's predictions
2) We select 100 misclassified SINK methods predicted by SUSI_ORIGINAL and compare CODOC's predictions

```
1   /**
2    * Always returns 0
3    *
4    * @return 0
5    * @hide
6    */
7   public int describeContents() {
8       return 0;
9   }
```

Listing 2: Source code of method "describeContents" of class "android.bluetooth.BluetoothCodecStatus"

3) We select 100 non-sensitive SOURCE methods classified by SUSI_NEW and compare CODOC's predictions
4) We select 100 misclassified SINK methods predicted by SUSI_NEW and compare CODOC's predictions
5) We select 100 misclassified SOURCE methods predicted by CODOC and compare with both SUSI_ORIGINAL and SUSI_NEW
6) We select 100 misclassified SINK methods predicted by CODOC and compare with both SUSI_ORIGINAL and SUSI_NEW

Methods selection: Regarding the sources, the authors randomly browsed the SOURCE methods yielded by SUSI_ORIGINAL, SUSI_NEW, and CODOC, and consulted the source code and the documentation to ensure the sensitiveness of the methods. In case the methods did not return sensitive values, they were retained for this research question. The authors stopped after 100 SOURCE methods (which is statistically significant at a 95% confidence level and a confidence interval $\pm$ 10% from the 25 369 sources of SUSI_ORIGINAL, the 12 082 of SUSI_NEW, and the 15 105 of CODOC). Regarding the sinks, the same procedure was applied as for sources, except the authors checked if the methods could send data out of the app (100 sinks are statistically significant at a 95% confidence level, and a confidence interval $\pm$ 10% from the 5913 sinks of SUSI_ORIGINAL, the 1010 of SUSI_NEW, and the 1061 of CODOC). An example of a non-sensitive SOURCE method misclassified is: "android.bluetooth.BluetoothCodecStatus.describeContents()" that can be seen in Listing 2. Indeed, the documentation and the source code are very explicit: this method only returns 0. An example of a SINK method misclassified is: "com.android.internal.util.FastMath.round(float)", the source code and documentation of which are available in Listing 3. Indeed, this method is only intended to round a float number. It does not write any value outside the app.

Eventually, we gathered 6 datasets with 100 methods each.
Results: The overall results are available in Table V. From the 100 non-sensitive SOURCE methods classified as sources by SUSI, CODOC only classified 13 as sensitive SOURCE methods. From the 100 non-SINK methods classified as sinks by SUSI, CODOC only classified 4 as SINK methods.
Discussion: These results, in the wild, do not confirm that CODOC is better than SUSI (contrary to what is shown in Section VI-B), nor the other way around, due to the very low overlap between the SOURCE and SINK methods predicted by
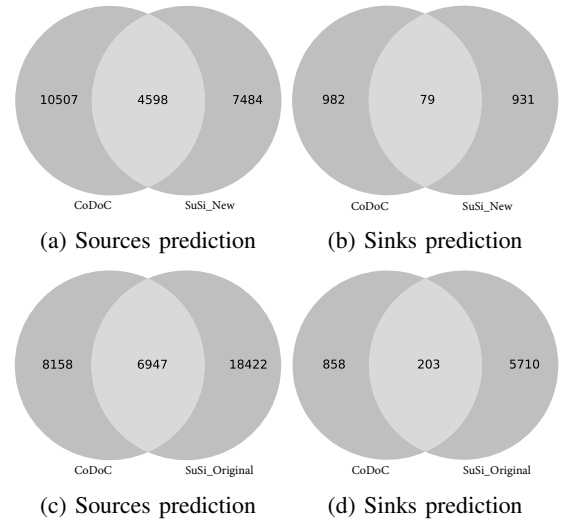

(a) Sources prediction (b) Sinks prediction

(c) Sources prediction (d) Sinks prediction

Fig. 7: Overlap between SOURCE and SINK methods lists of SUSI and CODOC.

```
1   /**
2    * Fast round from float to int. This is faster than
     ↪  Math.round()
3    * thought it may return slightly different results. It
     ↪  does not try to
4    * handle (in any meaningful way) NaN or infinities.
5    */
6   public static int round(float value) {
7       long lx = (long) (value * (65536 * 256f));
8       return (int) ((lx + 0x800000) >> 24);
9   }
```

Listing 3: Source code of method "round" of class "com.android.internal.util.FastMath"

both tools. Tables V and VI do not show that any approach is stronger. Better, taken independently, Table V shows that CODOC is preferable, while Table VI shows that SUSI is preferable. Hence, even if intuitively, by relying on both source code and documentation, better results are expected, in practice, it is not true according to our experiments and observations.

> **RQ3 answer:** While the outputs of CODOC and SUSI differ, neither is strictly superior over the other. Though CODOC excels on SUSI's misclassified results, the inverse is also correct.

Prediction	SUSI_ORIGINAL	CODOC	SUSI_NEW	CODOC
SOURCE	100	13	100	26
SINK	100	4	100	18

TABLE V: CODOC performance on SUSI's misclassified SOURCE and SINK methods

Prediction	CODOC	SUSI_ORIGINAL	CODOC	SUSI_NEW
SOURCE	100	32	100	6
SINK	100	3	100	0

TABLE VI: SUSI performance on CODOC's misclassified SOURCE and SINK methods

	False positives	True positives
SOURCE	90	10
SINK	56	44

TABLE VII: Source and sink methods' false and true positives' rates predicted by CODOC

D. RQ4: Real-world performance of CODOC

Objective: In this section, we aim to qualitatively evaluate CODOC's predicted lists of SOURCE and SINK methods, which is the main goal of this approach, i.e., *generating lists of SOURCE and SINK methods actionable to other tools, e.g., data leak detectors.* To do so, we randomly selected 100 SOURCE and 100 SINK methods in the lists generated by **CODOC**, which is statistically significant for the dataset at a 95% confidence level and a confidence interval $\pm$ 10%, and inspected each method based on two criteria: ① the sensitiveness of SOURCE methods for user privacy; and ② the fact that SINK methods can make data leave the app space.

Results: The results of our manual analyses are available in Table VII. Although, as seen in Section VI-B, CODOC achieves a score of 82% precision in the lab to classify SOURCE methods, in the wild, it reaches a false positive rate of 90%. Same conclusion for SINK methods for which though CODOC achieves a precision score of 93% to classify SINK methods, in the wild, it reaches a false positive rate of 56%.

> **RQ4 answer:** Results indicate that though CODOC achieves high-performance scores when assessing its underlying deep learning model, in the wild, it achieves poorly with over 90% false positive for SOURCE methods and 56% false positive for SINK methods.

E. RQ5: False positive measurement in sensitive data leak detection

To evaluate the effect of the false positives generated by CODOC on real-world applications, we utilize FLOW-DROID [4] to find data leaks in Android apps. Intuitively, a false positive in a list of sources or sinks is only relevant if it leads to spurious leaks in the data flow analysis. A method that is never used in an app might be on a source or sink list, but does not have any negative effect in practice.

For this evaluation, we randomly selected 500 popular apps from the 2022 GOOGLE PLAY across all available categories. For each app analysis, we set FLOWDROID timeouts to 5 min for data flow analysis and 3 min for callback collection and configured the JVM with 768GB maximum heap size. We ran the analysis on a system with 144 logical cores backed by four physical Intel Xeon Gold 6254 CPUs. Note that we focus on the quality of sources and sinks and not the performance of the data flow analysis. We, therefore, opted for a system with sufficient resources to scale to large apps.

We configure FLOWDROID with three different lists:

1) CODOC: The list generated by CODOC on Android 30.
2) SUSI_NEW: The list generated by SUSI, where SUSI was trained on the ground truth presented in this paper and classified methods of Android version 30.

3) SUSI_ORIGINAL: The list generated by SUSI, where SUSI was trained on the original SUSI training data and classified methods of Android version 30.

For each of these lists (Src_{NN}, Snk_{NN} with $NN \in$ {SUSI, SUSI_NEW, CODOC}), FLOWDROID yields a set of data flows $Flow_{NN}$. We then use the data flows $Flow_{NN}$ to remove all sources and sinks from the lists that are not used in at least one data flow. This leads to a reduced list of sources and sinks, i.e. $\widehat{Src_{NN}}, \widehat{Snk_{NN}}$. We validate these lists by hand and count the number of methods that lead to leaks, but that are not actually privacy-sensitive.

For CODOC, we find `StringBuilder.toString()`, which is clearly a false positive, to be the most commonly used "source" in the data flow analysis (72% of all flows). The second most common source was `StringBuffer.toString()` with around 8% of all flows. The sinks are more reasonable, with the Android log methods being the most prevalent ones (21% of all flows).

The SUSI_NEW results lead to far fewer flows (3,410 instead of 71 211 for CODOC). The used sources and sinks are more widely distributed, i.e., the top source only accounts for 10% of all sources. Still, the used sources and sinks are mostly false positives.

For SUSI_ORIGINAL, we find the most flows (153 558). The structure of used sources and sinks resembles SUSI_NEW, i.e., a wide variety of methods, most of which are false positives.

> **RQ5 answer:** The results show that the false positives generated by SUSI and CODOC have a major negative impact on the precision of the data flow analysis that uses these lists of sources and sinks.

VII. DISCUSSION

We designed our study and approach under the hypothesis that adding more semantic, i.e., using code and documentation together, would provide better results than the current state of the art to classify SOURCE and SINK methods. Further, we integrated recent advances in machine learning. Unfortunately, empirical results show that CODOC performs poorly in practice. More precisely, our investigations show that:

- Although code and documentation are complementary to predict SOURCE and SINK methods, CODOC performs poorly in the wild.
- Although CODOC achieves good lab results and better lab results than SUSI, i.e., precision, recall, and F_1 score of 91% compared to a precision, recall and F_1 score of 87%, it performs poorly in the wild.
- CODOC generates many false positives when applied to Android framework methods, i.e., classifies SOURCE and SINK methods that are not SOURCE and SINK methods.
- The false positives in the SOURCE and SINK lists lead to false positives in the data flow analysis which renders the lists unfit for real-world data leak detection scenarios.

VIII. LIMITATIONS AND THREATS TO VALIDITY

CODOC relies on two inputs to make its prediction, namely the source code and the documentation of Android methods.

We acknowledge that the Android framework contains undocumented methods which cannot be taken into account by CoDoC. The lack of method documentation makes CoDoC miss some methods to classify, hence, sensitive SOURCE and SINK methods are certainly missed. However, the proportion of methods missed is too low (i.e., 18.6%) to fully explain the poor real-world performance of the approach.

CoDoC relies on supervised machine learning techniques which, by definition, need labeled data. Therefore, we performed manual labeling based on our expertise to label Android framework methods as SOURCE, SINK, or NEITHER. Consequently, though we observed a strict and consistent procedure, our labels can be influenced by human subjectivity. Nonetheless, we make public all of our artifacts to the research community to mitigate this threat to validity.

Our training set is limited to 1015 samples across three classes. This might not be enough training data. We will explore the use of data augmentation in future work.

Sensitiveness is a concept not well-defined, especially for technical frameworks, and exposed to human subjectivity since there is no formal definition of what it is. Also, the authors noticed during manual labeling that sometimes there is only a fine line between a sensitive value and a non-sensitive one. Therefore, the choices regarding sensitiveness can be biased based on human subjectivity.

As already described and motivated in Section V-A, our manual labeling process was performed without taking into account SuSi's NEITHER methods list. However, we note that since SuSi yields very good results on the NEITHER category [3], there is a high chance that this list contained well-classified samples, hence being more representative than the ones misclassified as SOURCE and SINK methods.

IX. RELATED WORK

In this section, we present the related works available in the literature that are close to our work.
Taint analysis, which requires proper lists of sources and sinks, is used for a variety of purposes: vulnerability detection [33], [9], [37], sensitive data leak detection [4], [28], [45], [14], [58], [20], [58], [45], [61], hidden behavior detection [66], [53], malware detection [51] or bad practices detection [63]. All of these works require lists of SOURCE and SINK methods that have to be defined in advance. If these lists are not complete, the approaches may miss important data flows. Therefore, automated techniques were proposed to catch as many SOURCE and SINK methods as possible, aiming to reach completeness. In 2012, Gibler & al. [19] proposed an approach to automatically detect SOURCE and SINK methods based on mappings between methods and the permissions needed to invoke those methods. Methods requiring sensitive permissions were considered sources. Methods requiring the INTERNET permission were considered sinks. However, not all sensitive methods need permissions in the Android framework [21]. Thus, permission-based approaches miss relevant sources and sinks. Our work, on the other hand, considers all methods in the Android framework regardless of required permissions.

Two years later, Arzt & al. [3] proposed SuSi, an automated approach that relies on machine learning to classify SOURCE and SINK methods in the Android framework. SuSi relies on features based on the method signature (e.g., the method name, the parameter types, the return value, the modifiers, etc.) and based on dataflow features. In contrast to SuSi, our approach CoDoC relies on a multi-input deep-learning classifier based on: ① the source code, and ② the documentation of a method.

More recently, Wongwiwatchai & al. [59] proposed an approach to detect privacy leaks in Android apps. The authors did not rely on existing lists of SOURCE and SINK methods. Rather, they defined their own lists. To do so, the authors study well-known frameworks on data protection (e.g., the GDPR) and constitute a list of personal information commonly defined in these regulations. The process of mapping personal information (e.g., an age) to Android APIs is opaque in the paper. Hence it is difficult to judge the approach's comprehensiveness and the rate of false positives. In contrast, our approach aims to automatically and systematically map sensitive API methods to sensitive data with machine learning techniques.

X. CONCLUSION

As described in Section VI-D, CoDoC, likewise SuSi, does not provide actionable results in the wild. Indeed, although we have shown in Section VI-B that CoDoC outperforms SuSi with a score of 91% on our ground truth, our manual evaluations have shown that CoDoC performs poorly on the Android framework methods. Hence, the resulting SOURCE and SINK methods' list produced cannot be relied upon in real-world data leak detection scenarios. This negative result and the literature [57], [24], [35] show: ① the problem of classifying SOURCE and SINK methods is not trivial; and ② there is an urgent need of a community effort to produce an actionable list of SOURCE and SINK methods for sensitive data leak detection in Android apps carrying highly sensitive data about end users.

XI. DATA AVAILABILITY

For the sake of Open Science, we provide to the community all the artifacts used in our study. In particular, we make available the datasets used, the source code of our prototype and the scripts to execute CoDoC, our manually labeled datasets, the vector representation of source code and documentation, and SuSi related artifacts. The project's repository including all artifacts is available at: https://github.com/JordanSamhi/CoDoC

REFERENCES

[1] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[3] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013-0114*, 2013.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, June 2014.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[6] Chaity Banerjee, Tathagata Mukherjee, and Eduardo Pasiliao Jr. An empirical study on generalizations of the relu activation function. In *Proceedings of the 2019 ACM Southeast Conference*, pages 164–167, 2019.

[7] Luciano Bello and Marco Pistoia. Ares: triggering payload of evasive android malware. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 2–12. IEEE, 2018.

[8] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pages 65–88. Springer, 2008.

[9] Jun Cai, Peng Zou, Jinxin Ma, and Jun He. Sworddta: A dynamic taint analysis tool for software vulnerability detection. *Wuhan University Journal of Natural Sciences*, 21(1):10–20, Feb 2016.

[10] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, page 239–252, New York, NY, USA, 2011. Association for Computing Machinery.

[11] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[12] Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *J. Mach. Learn. Res.*, 7:1–30, December 2006.

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

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

[15] William Enck, Damien Octeau, Patrick D McDaniel, and Swarat Chaudhuri. A study of android application security. In *USENIX security symposium*, volume 2, 2011.

[16] Ming Fan, Le Yu, Sen Chen, Hao Zhou, Xiapu Luo, Shuyue Li, Yang Liu, Jun Liu, and Ting Liu. An empirical evaluation of gdpr compliance violations in android mhealth apps. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 253–264, 2020.

[17] Pietro Ferrara and Fausto Spoto. Static analysis for gdpr compliance. In *ITASEC*, 2018.

[18] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 377–396, 2016.

[19] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In Stefan Katzenbeisser, Edgar Weippl, L. Jean Camp, Melanie Volkamer, Mike Reiter, and Xinwen Zhang, editors, *Trust and Trustworthy Computing*, pages 291–307, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[20] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[21] Sigmund Albert Gorski, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. Acminer: Extraction and analysis of authorization checks in android's middleware. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, CODASPY '19, page 25–36, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[23] Hao Jiang, Hongli Yang, Shengchao Qin, Zhendong Su, Jian Zhang, and Jun Yan. Detecting energy bugs in android apps using static analysis. In Zhenhua Duan and Luke Ong, editors, *Formal Methods and Software Engineering*, pages 192–208, Cham, 2017. Springer International Publishing.

[24] Mohsin Junaid, Donggang Liu, and David Kung. Dexteroid: Detecting malicious behaviors in android apps using reverse-engineered life cycle models. *Computers & Security*, 59:92–117, 2016.

[25] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, Junbum Shin, and SWRD Center. Scandal: Static analyzer for detecting privacy leaks in android applications. *MoST*, 12:1, 2012.

[26] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[27] Li Li, Kevin Allix, Daoyuan Li, Alexandre Bartel, Tegawendé F. Bissyandé, and Jacques Klein. Potential component leaks in android apps: An investigation into a new feature set for malware detection. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 195–200, 2015.

[28] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 280–291. IEEE, 2015.

[29] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Apkcombiner: Combining multiple android apps to support inter-app analysis. In Hannes Federrath and Dieter Gollmann, editors, *ICT Systems Security and Privacy Protection*, pages 513–527, Cham, 2015. Springer International Publishing.

[30] Li Li, Tegawendé F. Bissyandé, Yves Le Traon, and Jacques Klein. Accessing inaccessible android apis: An empirical study. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 411–422, 2016.

[31] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88:67 – 95, 2017.

[32] Zhuang Liu, Wayne Lin, Ya Shi, and Jun Zhao. A robustly optimized bert pre-training approach with post-training. In Sheng Li, Maosong Sun, Yang Liu, Hua Wu, Liu Kang, Wanxiang Che, Shizhu He, and Gaoqi Rao, editors, *Chinese Computational Linguistics*, pages 471–484, Cham, 2021. Springer International Publishing.

[33] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, page 18, USA, 2005. USENIX Association.

[34] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 229–240, New York, NY, USA, 2012. Association for Computing Machinery.

[35] Linghui Luo, Eric Bodden, and Johannes Späth. A qualitative analysis of android taint-analysis results. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 102–114, 2019.

[36] Yuhong Nan, Zhemin Yang, Xiaofeng Wang, Yuan Zhang, Donglai Zhu, and Min Yang. Finding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps. In *NDSS*, 2018.

[37] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[38] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to

android inter-component communication analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 77–88, 2015.

[39] Xiaorui Pan, Xueqiang Wang, Yue Duan, XiaoFeng Wang, and Heng Yin. Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps. In *NDSS*, 2017.

[40] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: Hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, EuroSec '14, New York, NY, USA, 2014. Association for Computing Machinery.

[41] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. Droidforce: Enforcing complex, data-centric, system-wide policies in android. In *2014 Ninth International Conference on Availability, Reliability and Security*, pages 40–49. IEEE, 2014.

[42] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature antianalysis techniques. In *NDSS*, 2016.

[43] Dhruv Rathi and Rajni Jindal. Droidmark: A tool for android malware detection using taint analysis and bayesian network. *arXiv preprint arXiv:1805.06620*, 2018.

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

[45] J. Samhi, A. Bartel, T. F. Bissyande, and J. Klein. Raicc: Revealing atypical inter-component communication in android apps. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1398–1409, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[46] J. Samhi, L. Li, T. F. Bissyande, and J. Klein. Difuzer: Uncovering suspicious hidden sensitive operations in android apps. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 723–735, Los Alamitos, CA, USA, May 2022. IEEE Computer Society.

[47] Jordan Samhi, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein. A first look at android applications in google play related to covid-19. *Empirical Software Engineering*, 26(4):57, April 2021.

[48] Jordan Samhi and Alexandre Bartel. On the (in)effectiveness of static logic bomb detector for android apps. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, August 2021.

[49] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. Jucify: A step towards android code unification for enhanced static analysis. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1232–1244, Los Alamitos, CA, USA, May 2022. IEEE Computer Society.

[50] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Mohamed Ali Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *SECRYPT*, volume 96435, 2013.

[51] Venkatesh Gauri Shankar, Gaurav Somani, Manoj Singh Gaur, Vijay Laxmi, and Mauro Conti. Androtaint: An efficient android malware detection framework using dynamic taint analysis. In *2017 ISEA Asia Security and Privacy (ISEASP)*, pages 1–13, 2017.

[52] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *towards data science*, 6(12):310–316, 2017.

[53] Dawei Shi, Xiucun Tang, and Zhibin Ye. Detecting environmentsensitive malware based on taint analysis. In *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 322–327, 2017.

[54] M. Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society. Series B (Methodological)*, 36(2):111–147, 1974.

[55] F. Tomassetti. Javaparser, https://github.com/javaparser/javaparser. Accessed August 2021.

[56] Victor Van Der Veen, Herbert Bos, and Christian Rossow. Dynamic analysis of android malware. *Internet & Web Technology Master thesis, VU University Amsterdam*, 2013.

[57] Weiping Wang, Jianjian Wei, Shigeng Zhang, and Xi Luo. Lscdroid: Malware detection based on local sensitive api invocation sequences. *IEEE Transactions on Reliability*, 69(1):174–187, 2020.

[58] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS

'14, page 1329–1341, New York, NY, USA, 2014. Association for Computing Machinery.

[59] Nattanon Wongwiwatchai, Phannawhat Pongkham, and Kunwadee Sripanidkulchai. Comprehensive detection of vulnerable personal information leaks in android applications. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 121–126, 2020.

[60] Songyang Wu, Pan Wang, Xun Li, and Yong Zhang. Effective detection of android malware based on the usage of data flow apis and machine learning. *Information and software technology*, 75:17–25, 2016.

[61] Z. Yang and M. Yang. Leakminer: Detect information leakage on android with static taint analysis. In *2012 Third World Congress on Software Engineering*, pages 101–104, 2012.

[62] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054, 2013.

[63] Sergio Yovine and Gonzalo Winniczuk. Checkdroid: A tool for automated detection of bad practices in android applications using taint analysis. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 175–176, 2017.

[64] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 1105–1116, New York, NY, USA, 2014. Association for Computing Machinery.

[65] Mu Zhang and Heng Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *NDSS*. Citeseer, 2014.

[66] Qingchuan Zhao, Chaoshun Zuo, Brendan Dolan-Gavitt, Giancarlo Pellegrino, and Zhiqiang Lin. Automatic uncovering of hidden behaviors from input validation in mobile apps. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1106–1120. IEEE, 2020.

[67] Hao Zhou, Wei Zhang, Fengqiong Wei, and Yunfang Chen. Analysis of android malware family characteristic based on isomorphism of sensitive api call graph. In *2017 IEEE Second International Conference on Data Science in Cyberspace (DSC)*, pages 319–327, 2017.