DAISY: Dynamic-Analysis-Induced Source Discovery for Sensitive Data

XUELING ZHANG, Rochester Institute of Technology, USA
JOHN HEAPS, The University of Texas at San Antonio, USA
ROCKY SLAVIN, The University of Texas at San Antonio, USA
JIANWEI NIU, The University of Texas at San Antonio, USA
TRAVIS D. BREAUX, Carnegie Mellon University, USA
XIAOYIN WANG, The University of Texas at San Antonio, USA

Mobile apps are widely used and often process users' sensitive data. Many taint analysis tools have been applied to analyze sensitive information flows and report data leaks in apps. These tools require a list of sources (where sensitive data is accessed) as input, and researchers have constructed such lists within the Android platform by identifying Android API methods that allow access to sensitive data. However, app developers may also define methods or use third-party library's methods for accessing data. It is difficult to collect such source methods because they are unique to the apps, and there are a large number of third-party libraries available on the market that evolve over time. To address this problem, we propose DAISY, a Dynamic-Analysis-Induced Source discoverY approach for identifying methods that return sensitive information from apps and third-party libraries. Trained on an automatically labeled data set of methods and their calling context, DAISY identifies sensitive methods in unseen apps. We evaluated DAISY on real-world apps and the results show that DAISY can achieve an overall precision of 77.9% when reporting the most confident results. Most of the identified sources and leaks cannot be detected by existing technologies.

CCS Concepts: • **Software and its engineering**; • **Security and privacy** → *Human and societal aspects of security and privacy*;

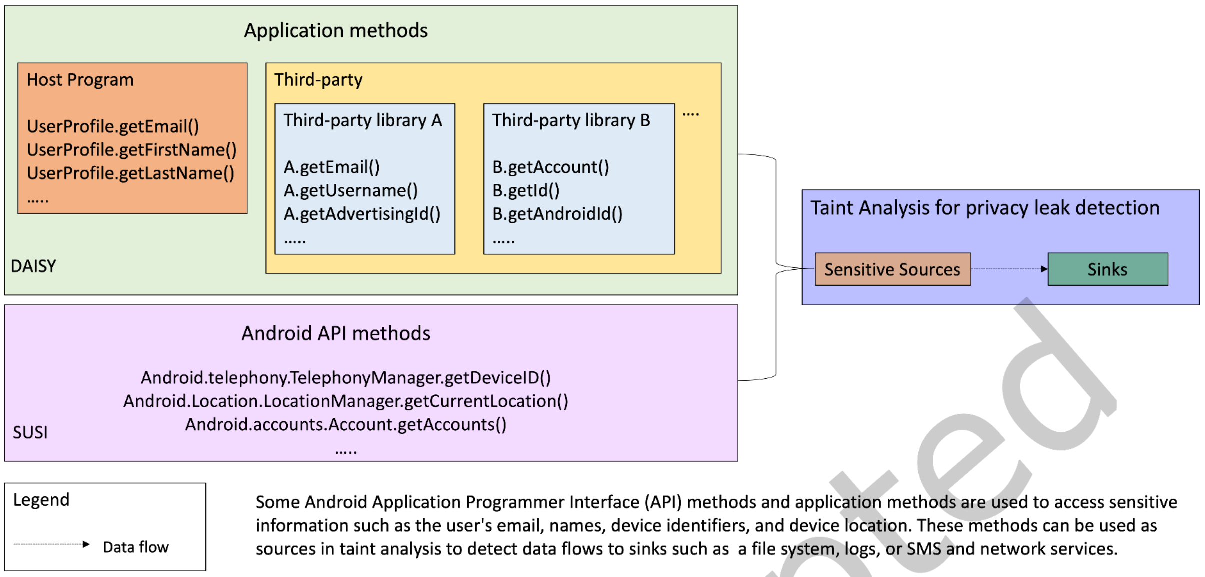
Additional Key Words and Phrases: privacy leak, mobile application, natural language processing

1 INTRODUCTION

Over the last decade, smartphones have become a necessity in people's daily lives, providing access to millions of mobile apps. These apps collect, process, and share sensitive user data for functional or commercial purposes. Widespread access to sensitive data raises privacy concerns in the mobile ecosystem. In response, privacy regulations [3, 4, 15, 23, 52] require software companies to disclose what sensitive data they collect, for what purposes it is used, and with whom it is shared. Inaccurate or misleading disclosures, and improper data processing can lead to privacy law violations and legal consequences. Ensuring alignment between disclosures and data processing is challenging due to the scale and variety of data processing within a mobile app: apps may contain up to tens of thousands of data flows, which can send data to various destinations, such as local storage, and first- and third-party web services. Moreover, apps are built using multiple frameworks, including the mobile

Authors' addresses: Xueling Zhang, Rochester Institute of Technology, Rochester, USA, xueling.zhang@rit.edu; John Heaps, The University of Texas at San Antonio, San Antonio, USA; Rocky Slavin, The University of Texas at San Antonio, San Antonio, USA; Jianwei Niu, The University of Texas at San Antonio, San Antonio, USA; Travis D. Breaux, Carnegie Mellon University, Pittsburgh, USA; Xiaoyin Wang, The University of Texas at San Antonio, San Antonio, USA.

1049-331X/2022/10-ART $15.00
https://doi.org/10.1145/3569936



Some Android Application Programmer Interface (API) methods and application methods are used to access sensitive information such as the user's email, names, device identifiers, and device location. These methods can be used as sources in taint analysis to detect data flows to sinks such as a file system, logs, or SMS and network services.

Fig. 1. Candidate sources for Android apps

platform, third-party libraries, and custom app code. To comply with law, app companies need tools to reliably trace sensitive data through their mobile apps. These tools can be used to verify practices described in a privacy policy or a privacy impact assessment in support of legal compliance.

Taint analyses [8, 20] have been proposed and widely adopted to understand data processing in code. They typically require, as inputs, a list of sensitive *sources*, which describe the starting point in the flow and may be represented as a method signature or data type (e.g., a user's device identifier); and *sinks*, which describe the end point in the flow as a method signature that corresponds to a means of storage or communication (e.g., a local file or network connection), and produce as outputs candidate data flows between the source and sink (see Figure 1). Early research in privacy-sensitive data flow analysis identified sources from the Android platform, which consists of Application Program Interface (API) methods that allow access to sensitive device resources, such as device location (e.g., `Android.Location.LocationManager.getCurrentLocation()` returns the device's location) and sensitive device identifiers (e.g., `Android.telephony.TelephonyManager.getDeviceID()` returns the device's IMEI). These methods can be used as sources by taint analysis tools for privacy leak detection (see Android API Methods in Figure 1). These sources were first identified manually [48] or semi-automatically [45, 73] from the Android API methods. Given the large number of public methods (more than 100,000) in the API, researchers have been unable to classify all sources manually or semi-automatically. To supplement these original source lists, Rasthofer et al. [54] proposed SUSI, an automated machine learning-based approach for classifying sources and sinks from the Android API methods, which significantly increased the number of identified Android API sources. Consequently, SUSI enabled new, large-scale taint analysis applications in privacy leak detection among Android apps to network servers [29, 59], and third-party libraries [31].

While SUSI identifies sensitive sources in the Android API methods, there is a second category of sensitive sources where data comes from user input through graphical user interfaces (GUI). The data input by the user can then be accessed through Android GUI API methods (e.g., android.widget.EditText.getText()), which are part of the Android API methods but are not labeled as sensitive sources by SUSI as they do not always return sensitive data. Recent work [7, 35, 47, 50, 68] demonstrates how user input data can be traced through GUI API method executions. Because user input is not always sensitive, this tracing method requires additional work to classify the input data type by first analyzing the GUI hierarchy [56] and next to classify labels associated with the method invocations.

Aside from Android API / GUI API methods, sensitive user data can be accessed through non-Android API methods, such as the methods defined by third-party libraries or apps themselves. An app may share sensitive user data with third-party services (e.g., social media profiles or mobile app telemetry) or use third-party services to store, process, and retrieve such data. Third-party libraries provide specific methods for accessing user data. Examples include method "`io.sentry.event.User.getUsername()`" from the third-party library "`Sentry`", "`io.branch.SystemObserver.getAdvertisingId()`" from the third-party library "`Branch IO`", and "`mobileapptracker.Parameters.getAndroidId()`" from another library "`MobileAppTracking`". Whereas the Android API methods are used across all Android apps, there are a larger variety of third party libraries that can be reused by some, but not all, apps (e.g., an app using Sentry's user telemetry and error tracking features). In addition, unlike the Android API methods, third-party library APIs are not always well documented.

In addition, an app may obtain data from its own back-end server (e.g., fetching user profiles previously collected through a web portal, through another app from the same organization, or through a hand-written registration form collected by the company). Afterwards, the app can send this data to any sink reachable throughout the app. Apps often define their own methods for accessing this data, such as "`UserProfile.getEmail()`" to return a user's email address or "`User.getFirstName()`" to return a user's first name. Unlike the Android API and third-party method names that are standardized across apps and libraries, these custom method names are unique to each app.

To identify non-Android sources (potential sources of sensitive data through neither the Android API nor GUI API methods), our prior work, ConDySTA [79], utilizes an existing user profile with pre-defined sensitive data and code instrumentation to identify app and third-party library methods that return sensitive data at run-time. We applied ConDySTA to 100 apps and detected 39 non-Android source leaks. Because ConDySTA is limited to test coverage, it will miss any source that is not triggered during testing, and the human effort required to increase test coverage means it cannot easily be applied to a large number of apps.

We address these limitations by proposing **DAISY** (Dynamic-Analysis-Induced Source DiscoverY for sensitive data), a novel machine learning-based approach to automatically identify *sensitive methods* in an arbitrary app or third party library (here a sensitive method refers to a method that returns sensitive data).

In general, the training phase of DAISY uses call stacks that are automatically collected and labeled during dynamic analysis. Next, the trained model is used to predict whether a method in an app or library is sensitive based on the method's calling-context described in the app's static call graph. Below, we summarize how DAISY effectively combines dynamic analysis, static analysis, and machine learning to address three major technical challenges, along with the intuition behind our solutions.

- **Constructing a sufficiently large training set.** While it is easy to automatically extract methods from many Android apps, manually labeling each method as sensitive or non-sensitive becomes prohibitively expensive due to the complexity of code semantics and the sparsity of sensitive methods (hundreds of non-sensitive methods may need to be reviewed before one sensitive method is found).
  Solution: DAISY overcomes this challenge using dynamic-analysis-induced, automatic labeling. During training, we run all of the training apps and collect the run-time return values of the methods that are being executed. Next, all the executed methods can be automatically labeled by checking whether their return values contain planted sensitive data (we can preset sensitive data, such as device ID and account email address, before running the apps in the training set).

- **Handling partially sensitive methods.** Some methods can be *partially sensitive* because they return sensitive values, but only under certain conditions. For example, at the top of the call stack in Listing 1, it is impossible to determine the information type of the `PreferenceHelper.getString()` method, because it is used across multiple contexts and returns either sensitive or non-sensitive data.
  Solution: To address this issue, instead of only classifying a single method, DAISY classifies methods combined with their calling contexts (called *in-context methods*). The same method with a different calling context can be labeled differently during training , thus yielding different label predictions depending on the context. In the example in Listing 1, we consider the calling context of the method `PreferenceHelper.getString(...)`, `UserAuthHelpr.getEmail(...)` to infer that it may return email address.
- **Recognizing text semantics in method signatures.** Method names are frequently composed of natural language words, which allows us to leverage advances in natural language processing (NLP) and machine learning to classify a method. While a word embedding provides a robust semantic representation of a word in a sentence, it can hardly handle words that were unseen in the training data, which are common in method signatures with informal, abbreviated texts.
  Solution: We handle informal texts by taking advantage of the subword embedding feature of the FastText [11, 36] framework. Subword embeddings considers sub-strings of words when constructing word embeddings so that an unseen word with sub-strings seen in training can be properly embedded.

```
//Return Value:
xxxxxxxxxxx@gmail.com
//Call Stack:
com.tubitv.helpers.PreferenceHelper.getString(PreferenceHelper.java:2)
com.tubitv.helpers.PreferenceHelper.getString(PreferenceHelper.java:3)
com.tubitv.helpers.UserAuthHelper.getEmail(UserAuthHelper.java:1)
```

Listing 1. Example of context-method

In the evaluation, DAISY was trained and validated using the call stacks we collected from the 200 top Android apps from Google Play, based on the rankings from PlayDrone [65]. We then applied DAISY to in-context methods extracted statically from the call-graphs of 100 apps ranked from 201 to 300, and 26,927 potentially sensitive in-context methods were identified. Since it is virtually impossible to manually label all discovered sensitive in-context methods, we chose two subsets. The first subset consists of 340 in-context methods which are predicted by DAISY with the highest confidence for different considered context lengths and information types. This high-confidence subset evaluates the effectiveness of DAISY when a user is interested in only the most-likely sensitive methods (e.g., when a user has limited time or resources for scanning a batch of apps). The second subset consists of 452 in-context methods which are randomly sampled, with a 20% rate, from in-context methods of 10 apps in our test set. This randomly sampled subset evaluates the effectiveness of DAISY when a user is interested in all sensitive methods (e.g., when an app developer seeks to avoid privacy violations). The evaluation results show that DAISY is able to achieve an average precision of 77.9% for the high-confidence subset and an average precision of 44.0% for the random subset. Further analyses of the confirmed new sources show that (1) among 464 detected and confirmed new sources, 437 can be detected by neither ConDySTA nor static taint analysis and (2) further considering calling contexts of length 2 and 3 helps to discover 46 and 23 more new sources, respectively.

The contributions in paper are summarized as follows:

- A novel approach, DAISY, to discover sensitive methods in Android apps along with their calling context based on machine learning and subword embeddings.
- An automatic labeling technique based on dynamic exploration of app code to extract large-scale training data sets from real-world apps.

- Viability for Android app marketplaces and developers to discover sensitive sources defined in the third party libraries and apps. Our evaluation shows that DAISY discovered a significant number of manually confirmed new sources that can be used in static and dynamic taint analysis.
- Multiple manually and automatically labeled data sets of in-context methods with sensitive information types that can be leveraged in future research.

The remainder of this paper is organized as follows: in Section 2, we present background knowledge on Android taint analysis, call stacks, and calling context; in Section 3, through an example, we demonstrate the limitations of existing approaches and the motivation for adopting machine learning to mitigate the limitation; in Section 4, we introduce DAISY and our machine learning model, followed by our evaluation setup and results in Section 5; finally, we discuss important issues in Section 6 and related works in Section 7 before concluding in Section 8.

2 BACKGROUND

In this section, we will introduce some of the concepts and techniques used in this work, including taint analysis, sources and sinks, call stack, and calling context.

2.1 Taint analysis for privacy leak detection

To ensure that users' data is only used in accordance with the relevant confidentiality policies, it is necessary to analyze how data flows within the program in question. Taint analysis is a type of information flow analysis in which objects are tainted and tracked using data flow analysis. There is a large body of work towards enforcing secure data flow through static [9, 12, 25, 27, 30, 38, 43, 45, 46, 48, 62, 70, 73, 74], or dynamic[20, 60, 63, 76] program analysis. Static taint analysis is performed prior to execution by considering all possible execution paths. Dynamic taint analysis is more precise than static taint analysis as it only propagates taint along the real path taken at run time. For smartphone apps, a data leak occurs when sensitive sources (phone numbers, device identifiers, contact data) flows to sinks (Internet, SMS transmission). Taint analysis is most frequently used to detect privacy leaks: it taints sensitive data at its source, and propagates the taint information through the application (or even a combination of apps), issuing a warning if tainted data reaches a sink.

2.2 Sources and sinks

Taint analysis requires sources and sinks as input and aims to discover connections between the provided sources and sinks. When using taint analysis to detect privacy leaks, we are interested in whether user's sensitive data flows to untrusted parties. *Source* is a statement retrieving sensitive data from the system and *Sink* is a statement saving data to storage or sending data outside of the application. In the code example from Listing 2, the user's device ID is read and send out as the text of an SMS message. In this sample code, the device ID is accessed though the Android API method `getDeviceId()` (on Line 4), which is the source, and the the device ID is then sent out though the Android API method `sendTextMessage` (on Line 7), which is the sink.

```
1  void onCreate() {
2      // Get the data
3  TelephonyManager mgr = (TelephonyManager) this.getSystemService(TELEPHONY_SERVICE);
4  String deviceId = mgr.getDeviceId();
5  // Leak the data SmsManager sms = SmsManager.getDefault();
6 sms.sendTextMessage("+49␣1234", null , deviceId , null , null);
7 }
```

Listing 2. Simple Data Leakage Example

2.3 False negatives in taint analysis

A false negative in taint analysis occurs when there's a data leak in the application but the analysis tool is unable to detect it. The cause of false negatives in taint analysis can be attributed to 1) inaccessibility in static taint analysis. There are several types of code that are statically inaccessible, such as dynamically loaded code, reflection code, native code, code executed on a remote server, and so on. When a data flow passes those code, a static taint analysis tool won't be able to detect it. 2) code coverage in dynamic analysis. Dynamic analysis relies on runtime execution, which is limited by the code coverage. It can only observe the data flow while the app is running and may miss the taint flows that are not triggered. 3) incomplete sources and sinks in all taint analysis. No matter how effective the analysis tool is, it can only guarantee to detect all privacy leaks when its list of sources and sinks is complete. If a source is missing, an app can still retrieve sensitive data from the source without being detected by the analysis tool. DAISY aims to address the third problem of incomplete source lists by automatically identifying the sensitive methods from apps and third-party libraries.

2.4 Call stack

DAISY uses method's call stacks as training data. Call stack is the sequence of active method invocations that lead to a program location during runtime. Developers can print a call stack at any point in the app code using provided API methods. For example, a call stack can be generated when the app crashes due to an error or exception, providing a list of method calls that led up to the thrown exception. Call stacks provide valuable information to developers for locating the cause of the crash [40]. Listing 3 is an example of a call stack. In DAISY, we automatically collect call stacks as training set for our learning models.

```
--------- beginning of crash
FATAL EXCEPTION: main
Process: com.zenga.zengatv, PID: 13218
android.view.WindowManager$BadTokenException: Unable to add window -- token android.os.
    BinderProxy@189f3cf is not valid; is your activity running?
 at android.view.ViewRootImpl.setView(ViewRootImpl.java:679)
 at android.view.WindowManagerGlobal.addView(WindowManagerGlobal.java:342)
 at android.view.WindowManagerImpl.addView(WindowManagerImpl.java:93)
 at android.widget.Toast$TN.handleShow(Toast.java:459)
 at android.widget.Toast$TN$2.handleMessage(Toast.java:342)
 at android.os.Handler.dispatchMessage(Handler.java:102)
 at android.os.Looper.loop(Looper.java:154)
```

Listing 3. A sample call stack

2.5 Call graph and calling context

DAISY generates test set from apps' call graphs. A call graph is a directed graph in which nodes represent methods and edges represent calls from one method to another. Call graphs are often used to help people understand programs, such as in taint analysis for tracking the flow of values. Call graphs can be dynamic or static. A dynamic call graph is a record of an execution of the program, such as the output of a profiler. Thus, a dynamic call graph can be precise, but it only describes one run of the program. A static call graph is one that is intended to represent every possible run of the program. That is, every call relationship that occurs is represented in the static call graph. In this work, in order to identify as many potential sources as possible , an app's call graph is generated statically. Figure 2 shows a sample call graph of an Android app's Login Activity. In this example, the Login Activity is launched when a user opens the app on their device. When the user clicks (or taps) the ”Login” button on the screen, the `onCreate()` callback is invoked, and an `onClickListener` is set to invoke `onClick()`. Once the user clicks the button, `onClick()` will be invoked. It will then call the `Authentication()`

and `getUserInput()` methods, which will invoke `getPasswords()` or `getUsername()` to collect a username or a password, respectively.

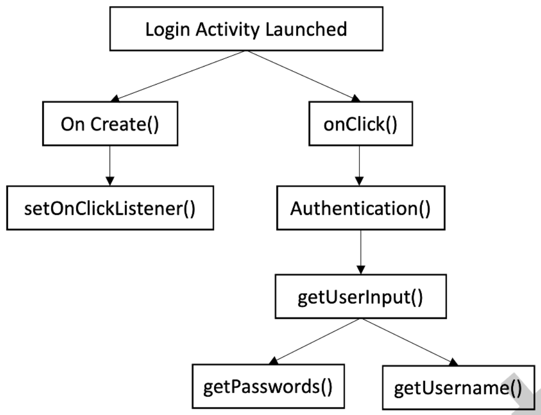


Fig. 2. A sample call graph in login activity

A calling context is a single path in the call graph that represents a method execution sequence. In Figure 2, the path from `onClick()` to `getPassword()` is a calling context of `getPassword()`, and the path from `onClick()` to `getUsername()` is a calling context of `getUsername()`. In this paper, the goal of DAISY is to identify potential sources in an unknown app based on static calling contexts without running the app. That is, we expect DAISY to predict whether a method from an unknown app is sensitive or not based on the method's static calling contexts. A method is sensitive if it extracts sensitive information from the application or device. In Figure 2, DAISY intents to predict whether `getUsername()` or `getPassword()` are sensitive or not based on their calling context.

3 MOTIVATION EXAMPLE

Our previous work, ConDySTA [79], relies on dynamic testing to trigger sensitive data-related events, which is limited by the testing coverage. With such a strategy, data accesses can be missed if a method is not triggered during testing. ConDySTA also requires human effort to register and login to the app, making it infeasible for use on a large scale. In the sample call graph in Figure 3, when an app launches the login activity, there are three options for users to login, `loginWithUsername`, `loginWithEmail`, and `loginWithPhoneNumer`. During testing, if only one path is executed, only the sensitive methods in that path will be identified. In the example, if `loginWithEmail` is executed, then only `getEmail()` and `getPassword()` will be triggered, which can be identified as sensitive methods (execution path highlighted in blue). However, in the other two branches, the methods `getUsername()` and `getPhoneNumber()` are also sensitive methods, but they will be missed because the branches were not triggered during testing. As another example, our evaluation results in Section 5.5 show that among the 464 sensitive methods identified by DAISY, only 23 of them are triggered in dynamic testing.

To avoid the limitation on test coverage, we propose DAISY, which utilizes machine learning techniques to identify sensitive methods from a statically generated call graph. DAISY will examine each method along with its calling context in the call graph and predict whether it may return sensitive information or not. In this example, the untouched leaf nodes, such as `getUsername()`, and `getPhoneNumber()`, along with their calling contexts, can be fed into a machine learning model for prediction. To train the machine learning model, we take advantage of the dynamically covered methods and their call stacks. For example, `getEmail()`, `getPassword()` in the graph (together with their dynamic call stacks) can be used as positive instances in the training set. Note that, once the model is trained, DAISY can use it to predict any methods in a new app without executing the new app. The advantages of using this machine learning strategy to identify sensitive methods in an app include: 1) every method can be examined; 2) no need to instrument and run the app; 3) no human effort required for registration or login, which makes it scalable for a large number of apps.

4 APPROACH

Our approach aims to identify sensitive methods defined in Android apps and third-party libraries. To achieve this goal, DAISY uses dynamic analysis to generate call stacks and automatically label them to train classification models. Given an unclassified method from an app, its calling context is extracted from the app's call graph, which is then input to the classification models to predict whether it returns sensitive information. Figure 4 shows the overview of our approach using four different layers: input, data set preparation, classification, and output. In the figure, the ovals represent processes or actions, the trapezoids represent artifacts of our classification model, and the parallelograms denote data. Section 4.1 describes the input data for our approach, which includes the apps used for training and prediction, as well as the user profile we created for labeling. Section 4.2 describes how we use dynamic analysis to generate training data and automatically label them. Section 4.3 describes the process of training our classifiers. Section 4.4 discusses how we use static analysis to generate the data set for prediction. The trained classifiers are then used to predict whether a given method is sensitive or not. Furthermore, we use the outputs of the classification as input sources of taint analysis for data leak detection.

4.1 Input Layer

DAISY takes different inputs for the training and prediction phases. The input for the training phase includes a collection of apps (i.e., apk files) and a user profile. This profile, labeled **Sensitive Data User Profile** in Figure 4, serves as an input for the auto-labeling process. The details of how we use it for labeling will be discussed in

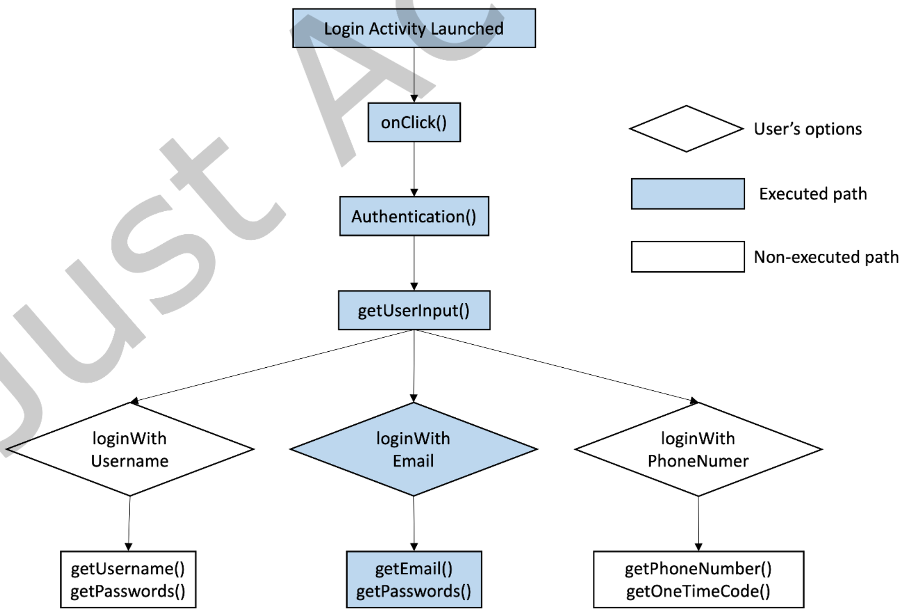


Fig. 3. A sample call graph in login activity

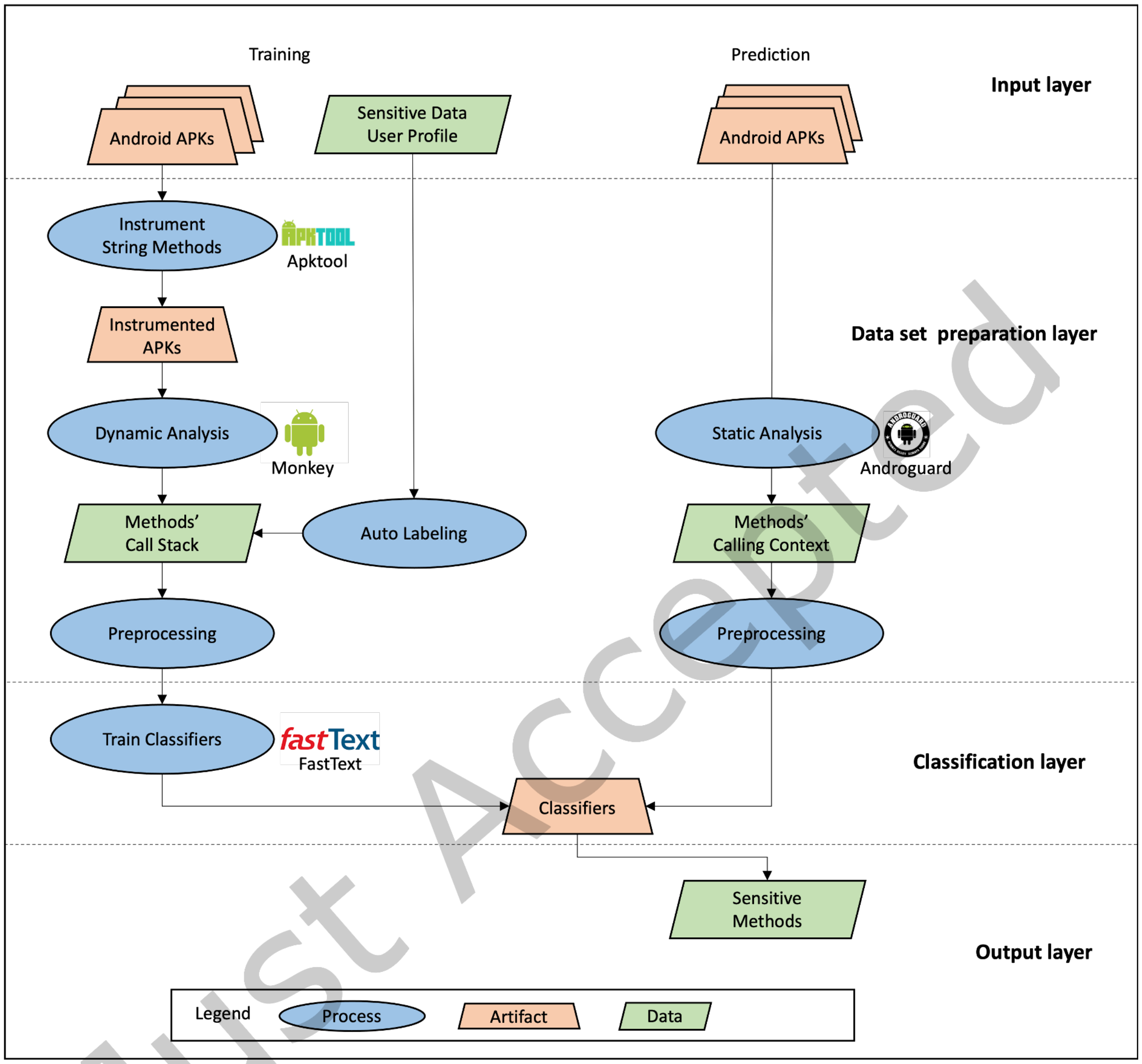


Fig. 4. Approach Overview

Section 4.2. The profile includes six unique identifiers: advertising ID, device ID, android ID, email address, IMEI, serial number, and user name. The profile values will be used as sources in value-based dynamic taint analysis for generating training sets. To reduce noise, each predefined source value must be unique so that other values do not accidentally contain them. Table 1 shows the six information types and their corresponding values in the user profile. We obtain the device identifiers from our test device, and we created a Gmail account and username. Many apps collect and access profile data, which ensures this step is broadly applicable across a large number of apps. We chose those six types of information because they are personally identifiable information (PII) as

Table 1. Sensitive Data User Profile

Info Type	Sample Value
Advertising Id	"fc1303d8-7fbb-44d8-8a68-a79ffac06fea"
Android Id	"a54eccb914c21863"
Email	"********@gmail.com"
IMEI	"355458061189396"
Serial	"ZX1G22KHQK"
User Name	"*******"

defined by the EU General Data Protection Regulation (GDPR) [1, 2]. In addition, these six information types are well studied, prior works [42, 78, 79] consider them as important sensitive data in privacy protection. The input of the prediction phase is a set of apps to be analyzed for identifying sensitive methods in them.

4.2 Data Set Preparation Layer (Training)

To prepare the training set, we collect methods' call stacks and automatically label them based on methods' return values. We use value-based dynamic taint analysis to identify data flows by checking whether run-time values of variables contain predefined sensitive values (e.g., username). We choose value-based dynamic taint analysis because it is robust enough to handle native code and out-of-scope data flows (e.g., information from third-party services such as Facebook, manual / web-based registration information from remote servers, information from file / user interface). The major limitation of value-based dynamic taint analysis is its inability to handle encrypted / obfuscated data, but since we are monitoring variables inside app code, we believe such inability may not cause much noise as encryption and obfuscation are typically performed only when data is sent out. The automatic labeling process consists of four steps describe below.

4.2.1 App Instrumentation. To build and label our training set, we need to collect methods' return values and call stacks at run time. In order to observe the return values and call stacks of methods at run time, we instrument all `String`-type methods in the smali code by inserting Android logging invocations at their return statements (**Instrument String Methods** in Figure 4). Each inserted invocation prints the run-time value of the returned variable and the call stack at the return statement. In particular, we acquire the call stack by throwing an exception and catching it immediately, while fetching the call stack saved in the exception variable. It should be noted that we instrument only `String`-type methods because their return values are readable, and although sensitive data is often packaged into objects, they are typically accessed through String-type methods. For example, a method named `getUserProfile()` may return an object of type `UserProfile` which contains username and Android Id as fields, but the actual values of username and Android Id are typically accessed through `UserProfile.getName()` and `UserProfile.getID()`, which are both String-type methods. For this process, we use APKTOOL [71] to decompile APK files into smali code[1]. As seen in Figure 4, the resulting **Instrumented APKs** are then used as input to the next step to generate call stacks.

4.2.2 Call Stack Generation. The resulting instrumented code is then rebuilt back into the APK format for our value-based dynamic taint analysis (**Dynamic Analysis** in Figure 4). We use the Android Debug Bridge (adb) to automatically install the rebuilt apps onto our test device and run MONKEY[17] to perform the testing on the apps. MONKEY is a tool developed as a part of the Android toolkit to perform GUI testing on apps. The tool generates different types of UI events that can interact randomly with the activity components of an app such as clicks, drags, and touches. We use MONKEY to perform testing because it is the most well-established event

[1]Assembler for the dex format used by Dalvik

generator used in Android dynamic analysis, and it is fully automated and robust enough to be applied to all apps. Furthermore, existing studies [66] show that Monkey achieves comparable coverage with state-of-the-art tools. For each app, we automatically install, execute, test, uninstall, and save the system log into the local file system for later inspection. For apps that require registration and login during testing, we manually create accounts using the user profile data we predefined in Table 1 to complete the login process. This ensures that DAISY will identify sensitive call stacks by searching for the values from the profile data. During testing, for each method that is triggered, its return value and call stacks are saved into the system log. Listing 4 shows an example of saved data, where line 1 shows the return value (`fc1303d8-7fbb-44d8-8a68-a79ffac06fea`), line 2 shows the method that has been triggered (`com.facebook.internal.AttributionIdentifier.getAndroid.AdvertiserId()`), and the rest of lines show the call stack of this method.

When run time testing is completed, the system log contains the collected call stacks of the *String*-type methods and their corresponding return `String` values under the context of that call stack (**Methods' Call Stack** in Figure 4), which will then be used as input for the process of **Auto Labeling** and **Preprocessing**.

```
1  16:25:13.442  W System.err: java.lang.Exception: fc1303d8-7fbb-44d8-8a68-a79ffac06fea
2  16:25:13.443  W System.err: at com.facebook.internal.AttributionIdentifiers.
        getAndroidAdvertiserId(AttributionIdentifiers.java:1)
3  16:25:13.443  W System.err: at com.facebook.marketing.internal.RemoteConfigManager.run(
        RemoteConfigManager.java:5)
4  16:25:13.443  W System.err: at java.util.concurrent.ThreadPoolExecutor.runWorker(
        ThreadPoolExecutor.java:1133)
5  16:25:13.443  W System.err: at java.util.concurrent.ThreadPoolExecutor$Worker.run(
        ThreadPoolExecutor.java:607)
6  16:25:13.443  W System.err: at java.lang.Thread.run(Thread.java:761)
```

Listing 4. System log of `String`-type method call stack

4.2.3 Automatic Labeling. We label a call stack based on its return value (**Auto Labeling** in Figure 4). If the return value of a call stack matches the value of a sensitive information type in Table 1, the call stack will be labeled with that sensitive information type. Otherwise, it will be labeled as "non-sensitive". In the example of Table 4, line 2 shows the `String`-type method `com.facebook.internal.AttributionIdentifiers.getAndroid.AdvertiserId()`. Line 1 shows its return `String` value `"fc1303d8-7fbb-44d8-8a68-a79ffac06fea"`, which matches with the information type "AdvertiserID" in our user profile. In this example, the call stack (Line 3-6) will be labeled with the sensitive information type "AdvertisingID". It should be noted that a call stack can be labeled with multiple sensitive information types if its return value contains multiple values in TABLE 1.

As discussed in Section 1, some methods may return different values in different calling contexts. We present an example of such method in Listing 5. In the listing, method `com.crashlytics.android.core.CrashlyticsCore.sanitize.Attribute()` returns the user's email address when it was called by method `com.crashlytics.android.core.CrashlyticsCore.set.UserEmail()`, but it returned non-sensitive information when it was called by other methods. To address the challenge of the same method returning different values in different calling contexts, we label call stacks up to the length being considered, and label a call stack s with a sensitive information type t only when all the observed call stacks with s as their prefix returns the sensitive information type t. As an example in Figure 5, consider three dynamically observed call stacks with method a at the top: [email]$a \leftarrow b \leftarrow c$, [email]$a \leftarrow b \leftarrow d$, and []$a \leftarrow c \leftarrow e$. The first two call stacks return the email address, and the third call stack returns an empty string, and we assume that there are no other call stacks with a at their top. In such a case, if we consider call stacks with length one, we will mark a as non-sensitive because not all call stacks starting with a returns email. If we consider call stacks with

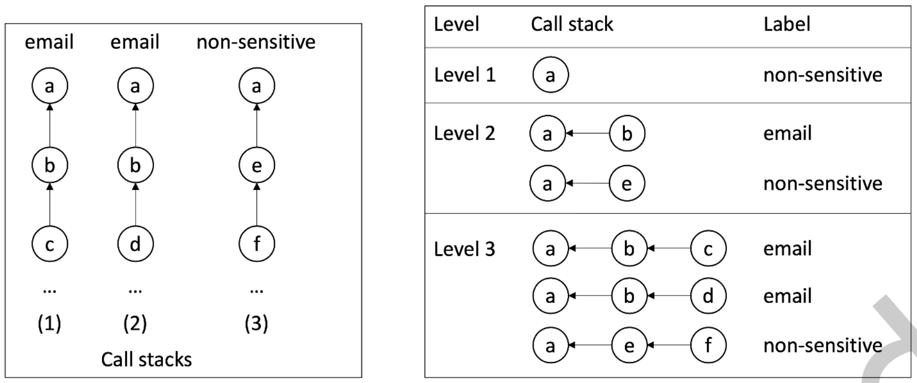


Fig. 5. Auto Labeling Strategy

length two, we will mark $a \leftarrow b$ as sensitive with type email because every $a \leftarrow b$ returns email, and $a \leftarrow e$ as non-sensitive. Using this strategy, we can ensure that we identify both *unconditional* sensitive methods (methods that return sensitive data in any context) and *conditional* sensitive methods (methods that return sensitive data only in certain contexts) together with their calling contexts.

```
//Return sensitive data
Return Value: **********@gmail.com
Call Stack:
com.crashlytics.android.core.CrashlyticsCore.sanitizeAttribute(CrashlyticsCore.java:845)
com.crashlytics.android.core.CrashlyticsCore.setUserEmail(CrashlyticsCore.java:528)
com.fitradio.ui.login.task.BaseLoginJob.handleLoginResponse(BaseLoginJob.java:146)
com.fitradio.ui.login.task.EmailLoginJob.getUserLoginEvent(EmailLoginJob.java:68)
com.fitradio.ui.login.task.BaseLoginJob.onRunRun(BaseLoginJob.java:67)

//Return non-sensitive data
Return Value: expiration_date
Call Stack:
com.crashlytics.android.core.CrashlyticsCore.sanitizeAttribute(CrashlyticsCore.java:845)
com.crashlytics.android.core.CrashlyticsCore.setString(CrashlyticsCore.java:560)
com.fitradio.ui.login.task.BaseLoginJob.handleLoginResponse(BaseLoginJob.java:153)
com.fitradio.ui.login.task.EmailLoginJob.getUserLoginEvent(EmailLoginJob.java:68)
com.fitradio.ui.login.task.BaseLoginJob.onRunRun(BaseLoginJob.java:67)
```

Listing 5. Method Returns Different Data Under Different Context

4.2.4 *Preprocessing.*

After creating call stacks and their corresponding labels, we preprocess call stacks into a format that can be processed by a natural language processing model (**Preprocessing** in Figure 4). In this step, each method signature from the call stacks is converted into a list of words. For each method in a call stack, we remove parameters from the end of the method name and split the method signature on the dot operator (.). Next, each part of the method signature is split on the punctuation allowed in Android method names (i.e., _ and $), followed by word-splits at capitalization changes (i.e., camel case boundaries) using a simple regular expression. Finally, all words are changed to lowercase. After preprocessing, for example, the method signature `android.location.Location.getLatitude()` becomes `[android, location, location, get, latitude]`. The above steps are applied to each method signature in order, from the last call (the method at the top of the stack) to the first call (the method at the bottom of the stack) and then concatenated into one word list. In the same order, the word lists of each method will be concatenated into one word list to represent the entire call stack.

At the end of this step, each of the call stacks that we collected during dynamic analysis will be preprocessed into a list of words, which will then be fed into a natural language model for text classification learning.

4.3 Classification Layer

In this layer, the preprocessed call stacks will be used to train classifiers to predict if a sensitive information type is being accessed (see **Train Classifiers** in Figure 4). We use FastText to represent the word lists and train a supervised classifier for each information type.

4.3.1 Method Signature Word Embeddings. A calling context consists of a list of fully qualified method names denoting the order in which methods were called. These method names are usually composed of words from natural language, e.g., `android.location.Location.getLatitude()` is the method signature and it is preprocessed into a sequence of words: `[android, location, location, get, latitude]`. These words are then represented as real-value vectors which can be used as input to train a machine learning model.

A common problem in word vector representations is how to handle *unknown words*. If a model has never seen a word previously, then finding a semantically relevant vector representation is difficult. *Unknown words* cause severe problems in text extracted from calling contexts and call stacks due to the informal language usage in source code; multiple abbreviated words are often combined to form a code identifier (e.g., "addr" for "address", "droid" for "Android").

To overcome this challenge, we use FastText [36] to produce vector representations for the word sequence of a calling context. FastText is a text classification technique based on the skip-gram Word2Vec[49] model for learning vector representations for words and text classification. The FastText framework helps to overcome the challenge of unknown words by splitting individual words into subwords, which are the set of overlapping character *n*-grams contained in a word. A character *n*-gram is simply a sequence of *n* characters of a word. Each character *n*-gram is assigned a vector representation and learned similarly to a whole word. Whole words are preprocessed by enclosing the words in the"<"and ">" characters to differentiate n-grams from n-length words. For example, the tri-gram "met" can be differentiated from the word "<met>"; these angle brackets are included when splitting a word into subwords. The subword vector representations and the whole word vector representation, if previously seen, are used to compute a final word representation. Thus, any new word representation is approximated using its constituent subwords. For example, if "biometric" was an unseen word with no existing vector representation, it could not be processed by the machine learning model. The tri-gram subwords of "<biometric>" consist of:

```
<bi, bio, iom, ome, met, etr, tri, ric, ic>
```

These subwords have a high probability of being seen before (i.e., they have vector representations) because most subwords are common and shared across other words in the English language. The subword vector representations are then averaged together to derive a semantically relevant whole word vector representation for "biometric".

Subword extraction will be performed only for words and will not process an entire signature as a single word. The data preprocessing (see 4.2.4) will first split full method signatures into word sequences and then subwords are derived for each word in those word sequences.

There are other text classification models that also use subwords, the most notable of which is BERT [18]. This model is a bidirectional transformer and takes advantage of sentence-level information and context to generate vector representations. We chose FastText over BERT because our data consists of method names which are not parts of normal sentences. Methods resemble short sentence fragments (e.g., "get Username", "get User Login Event") and do not maintain the same structure or complexity that BERT is initially trained on. Further, method names often use words in ways that BERT has not been trained on, such as repeating words (e.g., "on Run Run") and words that would not normally be seen together (e.g., "set String"). These and other issues can be solved by retraining the model, however this poses a problem for using BERT. Not only does BERT take many

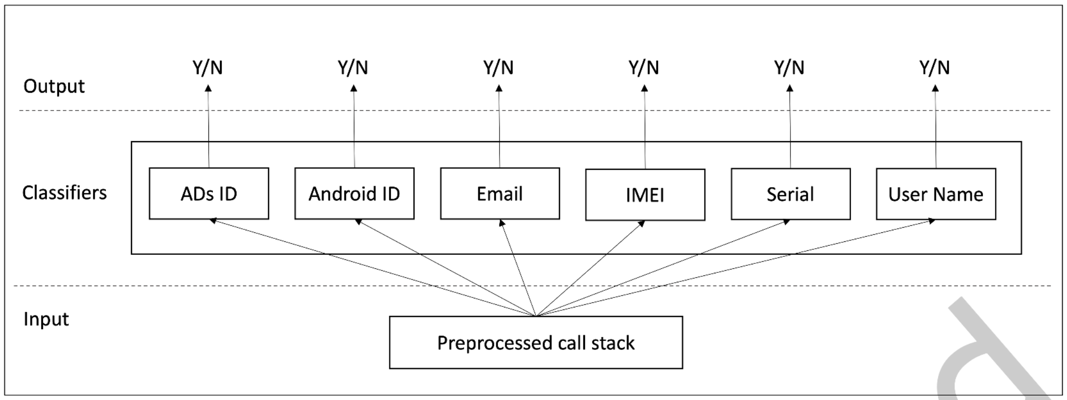


Fig. 6. Model Overview

days to retrain new vector representations, but it also needs a very large corpus to produce adequate vector representations; originally, BERT was trained with a corpus size of about 3.4 billion words and it is recommended to use a corpus of at least that size. For the 200 apps analyzed for training, we were only able to collect 3 million words from method name examples. In this case, we would need to analyze over 220,000 more apps to achieve the 3.4 billion recommended word count, which we do not have the time or resources to perform. To resolve these issues, we chose to use FastText as it can be trained from scratch using much less data and can be trained very quickly.

4.3.2 Sensitive Method Classifier. The learning task for our model is, given a method or sequence of methods from a call stack, classify whether the return value of the top method in the sequence is a sensitive information type. A multi-class classification model would be the standard machine learning approach, but a problem arises for our data because it is possible for a single method to return multiple sensitive information types. A multi-class learning model predicts the single best class from among a set of defined classes, which also usually includes one negative class (i.e., the data was predicted to not be a part of any other defined class). For our model, the classes would be our sensitive information types (Ads ID, Android ID, Email, IMEI, Serial Number, and User Name) and an additional non-sensitive class (i.e., the method did not return any of our sensitive information types). While the multi-class model typically predicts a single best class for each input, our data can return multiple sensitive information types. To mitigate this limitation, we decompose the learning task into separate binary learning problems: for each sensitive information type, a binary classification predicts whether the type is a specific sensitive type or "not that type". If a given method is classified by all models as "not that type", then it is assumed to be non-sensitive. Figure 6 shows the classifiers included in DAISY.

After FastText has completed its subword tokenization and averaging (described in Section 4.3.1), the input vector representation is provided to six different classifiers, where each is a binary logistic regression classifier. The output of each classifier for a single method follows the equation below:

$$f(CallTrace) = g(B \sum_{i \in \Phi}^{m} AX_i/m) \tag{1}$$

$$B \in \mathbb{R}^{c \times h}, A \in \mathbb{R}^{h \times v}, X_i \in \mathbb{R}^{v}$$

where g is the sigmoid function, A is an embedding matrix containing all of the model's whole-word and subword embeddings, Φ is the set of all embedding indices that compose the full call stack representation, and X_i is

Table 2. Table showing the word dimensions and epochs of the different FastText models from auto-tuning

	Android ID	**Email**	**Username**	**IMEI**	**Ads ID**	**Serial**
Level 1	dimension: 169 epochs: 37	dimension: 55 epochs: 47	dimension: 92 epochs: 100	dimension: 169 epochs: 37	dimension: 144 epochs: 100	dimension: 92 epochs: 100
Level 2	dimension: 169 epochs: 37	dimension: 169 epochs: 37	dimension: 92 epochs: 100	dimension: 100 epochs: 5	dimension: 92 epochs: 100	dimension: 92 epochs: 100
Level 3	dimension: 169 epochs: 37	dimension: 98 epochs: 100	dimension: 92 epochs: 100	dimension: 169 epochs: 37	dimension: 169 epochs: 37	dimension: 92 epochs: 100

a one-hot vector indicating the current index of the representation being summed. The matrix B holds the parameters for the logistic regression unit for each class, where c, h, and v represent the number of classes, the hidden size of each embedding vector, and size of the vocabulary (full-word and subword vocabularies together), respectively. For clarity, the summation simply describes the averaging of the full-word and subword embeddings.

Both matrix A and B are randomly initialized, which means the method representations are completely learned during the training phase. We found in our experiments that initializing with embeddings which were pre-trained on our call stack corpus did not improve classification performance for our task.

During training, labels from dynamic testing are given to the model to calculate the loss for each call stack. Each of the six regression unit's losses are calculated separately according to the well-known binary logistic regression loss function:

$$-y_{ic} \cdot log(f(CallStack)) + (1 - y_{ic}) \cdot log(f(CallStack)) \tag{2}$$

where $y_{ic} \in \{0, 1\}$ is an indicator of the i^{th} method's membership of class c. Minimizing the sum of equation 2 over each call stack is solved asynchronously using stochastic gradient descent [36]. Multiple threads are used to optimize performance and the number of threads is a hyperparameter of our model.

During inference, the output of equation 4.3.2 is calculated for each class. Our model predicts class membership using a threshold of 0.5, therefore, multiple classes can be predicted for a method. If no class probability is above 0.5, the call trace is predicted to be "non-sensitive".

The FastText model contains a number of hyperparameters that are auto-tuned. That is, FastText will automatically retrain the model based on many different hyperparameter values and then report the set of hyperparameter values that performed best for the given data. Some of the hyperparameters included in auto-tuning are: word dimension, learning rate, size of the hidden layer, and training epochs. The FastText model consists of an input layer, a single hidden layer, and an output layer. The size of the input layer and hidden layer are based on the the number of dimensions (i.e., the size) of the word vector embeddings. The number of elements in the vector that represents a word is defined by the "word dimension" value in FastText. Since this is auto-tuned, different sizes were determined for different trained models, as seen in Table 2 (shown as "dimension"). The size of the hidden layer is determined by the function $dim \times vocab$, where "dim" is "word dimension" and "vocab" is the size of (i.e., the number of unique words contained in) the vocabulary.

In our experiments with FastText, our vocabulary contained 11,538 unique words. When we trained a model for each information type at levels 1, 2, and 3, the hidden layer had a size range of 634,590 to 1,949,922. The number of epochs (i.e., the number of iterations the entire training data set was processed by the model during training) is also reported in Table 2. A learning rate of 0.0001 was reported for all models, chosen by the auto-tuning. All of the training was performed using the softmax loss function as that is the only loss function available in FastText.

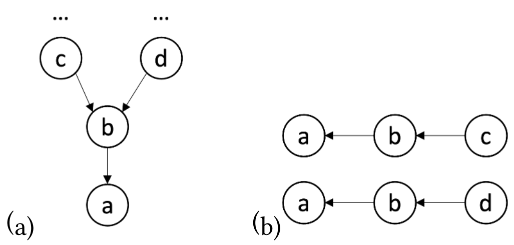


Fig. 7. Sample Call Graph and Calling Context

4.4 Data Set Preparation Layer (Prediction)

After the classification models have been trained, given an arbitrary Android app, DAISY automatically identifies the sensitive methods in that app. In particular, we prepare the app's data for model prediction using **Static Analysis** (as shown in Figure 4) to extract methods with their static calling contexts. The static analysis consists of the following steps:

First, DAISY utilizes the ANDROGUARD[16] to disassemble an app and extract its method call graph. ANDROGUARD is an well-known static analysis tool for Android apps. We chose ANDROGUARD because it is freely available and open-sourced (allow others to confirm our findings). The toolkits provided by ANDROGUARD have the ability to quickly and robustly process a large number of apps, and the ANDROGUARD library (of which the call graph generator is a component) has already been widely used in existing research [5, 14, 19, 22, 28, 37, 39, 41, 51, 58, 67, 69, 77]. ANDROGUARD disassembles an application into classes, methods, basic blocks, and individual instructions. Call relations of methods are identified by the invocation statements, such as `invoke-direct` and `invoke-static`, and then extended conservatively on class inheritance hierarchy. Although ANDROGUARD is not very precise (e.g., not considering object-sensitivity and flow-sensitivity), it fits our requirement due to its robustness, scalability and the comprehensiveness of generated call graph (without considering reflection). When manually inspecting the predicted sources (or detected leaks after applying taint analysis) to remove false positives, the false positives caused by inaccurate call graphs can also be removed (as in our evaluation).

From the method call graph, DAISY extracts only `String`-type methods as *methods for prediction* as they may directly return sensitive information (for the same reason as explained in Section 4.2.1). It should be noted that other methods may still be part of the calling contexts of these `String`-type methods. Then, for each method for prediction, DAISY traverses the call graph starting from its node and traverses backward towards the root (i.e., the program entry method) along all paths for up to two edges. By doing so, we are able to extract a list of static calling contexts for each method for prediction (**Methods' Calling Context in Figure 4**). For example, in Figure 7, (a) shows a sample call graph, where method a is called by method b, and method b is called by both method c and d. Based on this call graph, we extract the two calling contexts of method a, as (b) shows. Each calling context is then preprocessed in the same way as described in Section 4.2.4, and then fed into the trained classifiers. It should be noted that the same method with different calling contexts and different levels of calling contexts is fed into the classification models separately. For example, in Figure 7, method a's two calling contexts, $a \leftarrow b \leftarrow c$ and $a \leftarrow b \leftarrow d$ will generate four inputs: a, $a \leftarrow b$, $a \leftarrow b \leftarrow c$, and $a \leftarrow b \leftarrow d$, based on their length, they will be fed into the level 1, level 2, level 3, and level 3 models respectively.

5 EVALUATION

5.1 Research Questions

Our evaluation aims to answer the following research questions:

- RQ1: How precisely can DAISY identify sensitive methods and categorize them?
- RQ2: How many of the sensitive methods discovered by DAISY can not be found by applying existing static taint analysis or dynamic analysis?
- RQ3: How many additional conditional sensitive methods can DAISY find by considering more levels of calling contexts?
- RQ4: How many leaks are caused by the sensitive methods identified by DAISY?
- RQ5: How effective is the subword embedding feature on handling text from call stacks?

5.2 Data Set

We collected our subject apps from Goole Play based on PlayDrone[65], a collection of Android app meta data on the Google Play store. We used the 300 top-ranked apps whose APKs can be de-compiled and instrumented successfully. Among the 300 apps, the top 150 were used as the training set, the middle 50 apps (ranked 151-200) were used as the validation set, and the remaining 100 apps were used as our testing set[2].

5.3 Experiment Process

We used the approach described in Section 4.4 to prepare our test set. We generated call graphs of the 100 testing apps, from which, we extracted 196,282 methods (i.e., considered as in-context methods with calling context to level 1), 904,476 in-context methods with calling contexts to level 2, and 1,715,993 in-context methods with calling contexts to level 3. We consider calling contexts up to level 3 because longer calling contexts will lead to even more data and huge cost in the prediction process. Please note that in this paper our goal is to validate the usefulness of calling contexts. We leave the determination of an optimal calling-context length (which may be different for various information types and apps) as future work. Meanwhile, we trained and validated 18 classification models (the combination of three calling context lengths and six information types) using the call stacks collected from apps in the training and validation sets.

After that, we fed the collected in-context methods to their corresponding classification models (e.g., an in-context method with calling context to level 2 will be fed to all six classification models trained / validated with call stacks with length 2). After the prediction process, DAISY reported 2,237, 7,214, and 17,476 in-context methods as sensitive for calling context levels 1, 2, and 3, respectively.

5.4 Ground Truth Labelling

The 100 apps in our test set contain over 2.8 million in-context methods. So, it is impossible to label all of them and calculate the recall of DAISY. However, the recall metrics of DAISY do not prevent it from being practically useful. Since no technique can detect all sensitive methods, any approach that can discover sensitive methods that are undetected by existing techniques will be very useful. Since a user may need to manually review the discovered sensitive methods or the information leaks that come from them, the precision of DAISY can be very important, which indirectly measures the additional effort required by the user.

To calculate the precision of DAISY, we need to manually label the reported sensitive in-context methods to confirm the true positives and the false positives. However, in our test set, more than 26,000 in-context methods are reported as sensitive by DAISY, it is infeasible to label them all. Therefore, we manually label the following two sampled subsets to evaluate DAISY in two different usage scenarios.

In the first usage scenario, we consider DAISY to be used in app scanning at app stores / security analysis services where a human user cannot review too many reported sensitive methods due to the large number of apps under analysis. In this case, the user may only review the reported sensitive in-context methods that are most likely to be true positives. Therefore, we sample the first subset (high-confidence subset) by choosing the 20 most

[2]See our dataset and code at https://sites.google.com/view/daisy2022/

Table 3. Precision of DAISY on the High-Confidence Subset

InfoType	Level 1	Level 2	Level 3	Overall
AdsID	18/20	20/20	17/20	55/60 (97.1%)
AndroidID	15/20	15/20	18/20	48/60 (80.0%)
Email	16/20	17/20	10/20	43/60 (71.7%)
IMEI	0/0	6/20	15/20	21/40 (52.5%)
Seriel	15/20	19/20	17/20	51/60 (85%)
Username	13/20	15/20	19/20	47/60 (78.3%)
Overall	77/100 (77.0%)	92/120 (76.7%)	96/120 (80%)	265/340 (77.9%)

confidently reported sensitive in-context methods from each of the 18 classification models (a combination of three different calling-context lengths and six information types). The most confident predictions are chosen based on their probability scores, which range from 0 to 1. In total, this subset contains 340 reported sensitive in-context methods (no positive in-context methods were reported for IMEI information type with calling context to level 1). In the second usage scenario, we consider DAISY to be used by the app developer / app producing organization who wants to avoid privacy policy violations by examining all suspicious methods. For this scenario, we selected 10 apps from our test set based on their rankings (the apps ranked 10, 20, 30, ..., 100 were selected). Then, from each of those 10 apps, we randomly selected 20% of the reported sensitive in-context methods, yielding a subset of 452 in-context methods for labeling. Using this second subset (random subset), we can evaluate DAISY's precision on each of these apps. We manually labeled each of the 648 in-context methods in the two subsets. To reduce potential labeling errors, we have two authors independently labeling all of the methods, and a third author resolving any conflicts. The inter-rater agreement value for our manual labeling is 96.05% (Cohen's kappa). It should be noted that method labeling is a costly task that requires examination of all relevant code of the method in byte-code form, especially considering much of the code is obfuscated. Among the 648 in-context methods, there are 22 of them that we were not able to decide whether they were truly sensitive or not, so we conservatively considered all of them to be false positives in the evaluation.

To answer RQ1, we created Table 3 and Table 4, which show the precision of DAISY on the high-confidence subset and the random subset respectively. In Table 3, column 1 presents the information type, while in Table 4, column 1 presents the app name. Columns 2-4 of both Table 3 and Table 4 present the precision for each information type / app with each length of calling context. In each cell, the number before the / symbol is the number of manually confirmed true positives, while the number after the / symbol is the number of reported in-context methods. Column 5 shows the overall precision, which is computed by combining data from Columns 2-4.

From Table 3, we can see that in the high-confidence subset, DAISY is able to achieve an average precision of 70%, 76.7%, and 80% for calling context of length one, two, and three, respectively. The overall precision of 77.9% indicates that the majority of in-context methods identified by DAISY are the true positives, and a user will be able to identify 265 real sensitive in-context methods from 100 testing apps by reviewing only 340 most confidently reported in-context methods. It should be noted that these 340 in-context methods are from 58 different apps so they cover a large portion of testing apps. Among different information types, DAISY performs best on Android ID, Advertisement ID, and Serial Number, partly because the methods returning these information types tend to have standard names. Compared with these information types, Email and User Name are more difficult to identify due to the various ways to refer to them, but DAISY still achieved 71.7% precision for email and 78.3% precision for user name, indicating that using DAISY developers do not need to spend too much time on ruling out false positives. DAISY performs worst on IMEI, partly because methods returning IMEI are rarely called so

Table 4. Precision of DAISY on the Random Subset

App	Level 1	Level 2	Level 3	Overall
com.simplygood.ct	1/2	3/5	6/13	10/20 (50.0%)
com.skimble.fitnessflow.lite	0/1	1/5	2/6	3/12 (25.0%)
com.zenga.zengatv	1/1	5/12	5/22	11/35 (31.4%)
com.mlssoccer	3/5	10/14	15/24	28/43 (65.1%)
com.oki.letters	0/3	2/4	5/27	7/34 (20.6%)
com.mtvn.mtvPrimeAndroid	12/18	17/48	54/109	83/175 (47.4%)
com.klab.lods.en	2/2	2/9	8/25	12/36 (33.3%)
com.williamsinteractive.goldfish	2/3	8/14	8/29	18/46 (39.1%)
com.kidoz	3/3	6/10	8/11	17/24 (70.8%)
com.nevosoft.mysteryville	2/4	5/8	3/15	10/27 (37%)
Overall	26/42 (61.9%)	59/129 (45.7%)	114/281 (40.6%)	199/452(44.0%)

the dataset becomes very unbalanced. In the training set, less than 0.1% of in-context methods are labeled with IMEI, indicating that a random selection will lead to a precision of less than 0.1% and DAISY is 500 times more discriminative than that.

From Table 4, we can see that in the random subset DAISY is able to achieve an average precision values from 20.6% to 70.8% in ten apps, and an overall precision of 44.0%. From calling-context lengths of one, two, and three, the precision value is 61.9%, 45.7%, and 40.6%, respectively. It should be noted that sensitive in-context methods are sparse so the data imbalance is severe. In our training set, less than 1% of all in-context methods are labeled as sensitive, indicating that a random selection will lead to an overall precision of less than 1%, and DAISY performs tens of times better than that.

5.5 Supplementing Existing Approaches

To answer RQ2, we check whether the sensitive in-context methods detected by DAISY can be detected by either static taint analysis or dynamic analysis.

Static Taint Analysis. Based on an existing list of sensitive Android API methods, it is possible to use static taint analysis to expand the list and discover more sensitive methods. Listing 6 shows an example where the *sensitive method* `findDeviceId()` (detected by DAISY) obtained the IMEI information from the Android API method `android.telephony.TelephonyManager.getDeviceId()`. Because there is a data flow from the known Android API source to `findDeviceId()`, it is considered to be detectable by static taint analysis. To check whether our reported in-context methods are also detectable by static taint analysis, we applied FlowDroid [9] using sensitive Android API method list of SUSI [54] as sources, and checks whether SUSI sources may flow to the confirmed true positive in-context methods in Table 3 and Table 4. Specifically, we first collected a list of known sources from SUSI [54] that was labeled with the six information types included in this work (Advertising ID, Android ID, Email account, IMEI, Serial Number, and User name). Then we applied FlowDroid to statically analyze the test set apps for leaks using the collected SUSI sources as sources and the DAISY-reported *sensitive methods* as sinks. If FlowDroid identified a path from a SUSI source to a *sensitive method*, we considered that *sensitive method* to be detectable by static taint analysis. The analysis results show that only four of the 464 discovered in-context methods (by DAISY) were detected with the static taint analysis.

```java
// findDeviceId() is the source identified by DAISY

// android.telephony.TelephonyManager.getDeviceId() is an Android API source identified by
    SUSI

public String findDeviceId(Context context){
   TelephonyManager tm = (TelephonyManager) context.getSystemService("phone");
   return tm.getDeviceId();
}
```

Listing 6. An example of a DAISY source obtaining IMEI from a SUSI source.

Dynamic Analysis. It is also possible to use dynamic analysis to detect sensitive methods. Actually, our automatic labeling in the training process can be deemed as an approach to discover sensitive methods. However, fully automatic dynamic analysis is limited by test coverage, while manual testing to improve test coverage requires additional human effort. In our comparison, we used Monkey to perform the testing for the reasons stated in Section 4.2.2. For each app, we ran Monkey for one hour without any human interaction and used the automatic labeling described in Section 4.2.3 to identify sensitive in-context methods. The comparison shows that among the 464 true positives from Table 3 and Table 4, only 23 of them were detected by the dynamic analysis.

To sum up, the comparison shows that almost all sensitive methods detected by DAISY are new and cannot be easily detected by static taint analysis or dynamic analysis.

5.6 Conditional Sensitive Methods

To answer RQ3, we performed a statistical analysis on the result to see how many extra sensitive in-context methods were identified when we increased the length of calling context to be considered. Note that when the considered length is equal to one, our model identifies only unconditional source methods. When the length increase, our model identifies more conditional source methods, together with their calling contexts.

Table 5 shows the additional sensitive in-context methods reported when increasing the level of calling contexts (numbers in Columns 3 and 4). From the table, we can see that increasing calling context level to two helps DAISY to report 4,195 more potential sensitive in-context methods, and increasing the calling context level to three further helps DAISY to report 9,324 more potential sensitive in-context methods. Note that an in-context method $a \longrightarrow b$ is considered additional if and only if $a \longrightarrow b$ is reported as sensitive at Level 2, but a is not reported as sensitive at Level 1. Table 6 shows the total additional sensitive in-context methods that are confirmed as true positives in our evaluation subsets. With calling-context length increased to two, DAISY identifies 46 additional sensitive in-context methods that cannot be identified at level 1, increasing the calling-context length to three further helps to identify 23 sensitive in-context methods. Since in-context methods for different levels are sampled separately, in this table, an in-context method $a \longleftarrow b$ is considered additional if and only if $a \longleftarrow b$ is reported and manually confirmed as sensitive at Level 2, but a is not reported as sensitive at Level 1 (note that a does not need to be in the sampled sets). From the results in two tables, we can see that (1) increasing calling context length to two and three helps DAISY to detect many more sensitive in-context methods which are not detectable by classifying methods without context and (2) the additionally detected in-context methods cover all information types, indicating that conditional sensitive methods are common in all information types.

5.7 Origin of DAISY sources

We further investigated the sensitive in-context methods DAISY identified to find out where the methods were declared. Most of the sensitive methods found by DAISY are from third-party libraries. For example, one Advertising ID method `com.facebook.internal. Attribution.Identifiers.getAndroidAdvertiserId()` comes from a Facebook library. For the sensitive methods that return data from a user's input, such as email and username, most of them are located within the app itself (i.e., app-specific methods). For example, an Email

Table 5. Additionally Reported In-context Methods by Increasing Calling-Context Levels

InfoType	Level 1	Δ Level 2	Δ Level 3
Advertising ID	1128	1602	2368
Android ID	484	874	1132
Email	386	910	3986
IMEI	0	183	308
Serial	108	47	97
User name	131	569	1433
Total	**2237**	**4195**	**9324**

Table 6. Additionally Confirmed In-context Methods by Increasing Calling-Context Levels

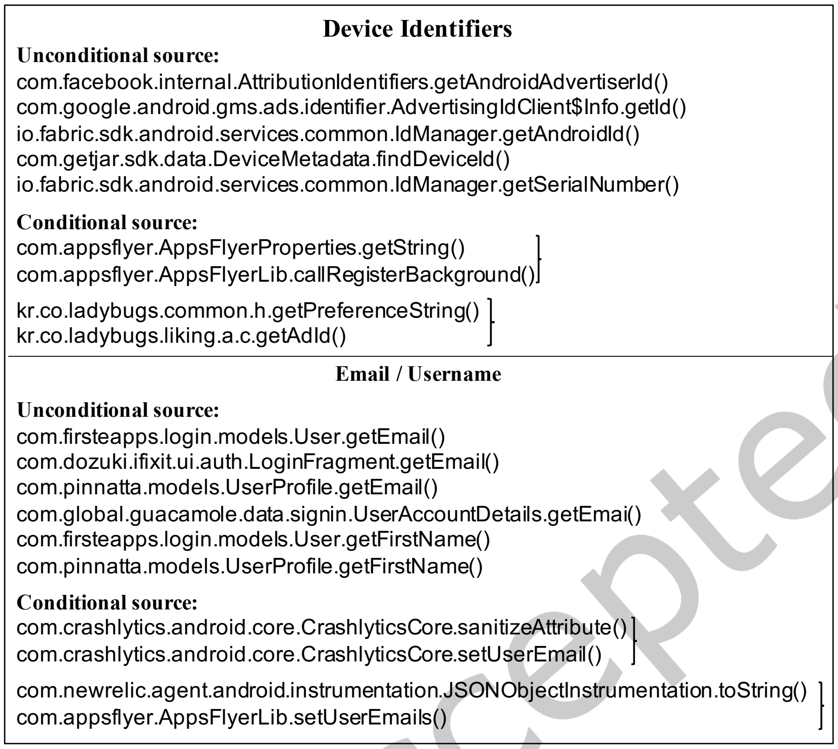
InfoType	Level 1	Δ Level 2	Δ Level 3
Advertising ID	32	7	0
Android ID	21	4	2
Email	20	12	13
IMEI	0	6	7
Serial	16	5	1
User name	14	12	0
Total	**103**	**46**	**23**

method `com.dozuki.ifixit.ui.auth.LoginFragment.getEmail()` is an app-specific method from the app `com.dozuki.ifixit`. From the method signature, we can tell that this method collects email accounts while a user logs in.

As discussed in Section 4.2.3, some methods may return different values in different calling contexts, and only return sensitive information in a specific context. Some examples of conditional (methods that return sensitive data only in certain contexts) and unconditional (methods that return sensitive data in any context) methods in different information types can be found in Figure 8. From the figure we can see that, DAISY can not only identify unconditional source methods, such as `getAndroidId()`, but also identify conditional source methods (e.g., `getString()`) which return general values and return sensitive information under certain calling context.

5.8 Leaks from DAISY sources

When using taint analysis to detect privacy leaks, a false negative occurs when there is a data leak in the application but the analysis tool is unable to detect it. One of the reasons for false negatives is caused by incomplete sources. If a source is missing, the analysis tool will not track any data flow from it. The goal of DAISY is to reduce such false negatives by automatically identifying sensitive methods in apps and third-party libraries, which can be used as sources in taint analysis. We refer to the sensitive in-context methods detected by DAISY as *DAISY sources.* To evaluate DAISY on data leak detection and answer RQ4, we applied static taint analysis to the DAISY sources to see if they help detect any additional leaks compared to sources from SUSI [54] and ConDySTA[79]. To do so, we ran FlowDroid with the confirmed 464 true positive DAISY sources described in TABLE 3 and TABLE 4 as sources and the sink list from SUSI as sinks. It should be noted that, rather than a single method being used as the source in taint analysis, the DAISY source we used in taint analysis is a method with its calling context. So we need to make sure that the calling context is encoded in the taint analysis. To do so, we adopted the approach from ConDySTA[79] to encode the calling context as an data flow prefix in the IFDS [55] (Inter-procedural Finite



Device Identifiers

Unconditional source:
com.facebook.internal.AttributionIdentifiers.getAndroidAdvertiserId()
com.google.android.gms.ads.identifier.AdvertisingIdClient$Info.getId()
io.fabric.sdk.android.services.common.IdManager.getAndroidId()
com.getjar.sdk.data.DeviceMetadata.findDeviceId()
io.fabric.sdk.android.services.common.IdManager.getSerialNumber()

Conditional source:
com.appsflyer.AppsFlyerProperties.getString()
com.appsflyer.AppsFlyerLib.callRegisterBackground()

kr.co.ladybugs.common.h.getPreferenceString()
kr.co.ladybugs.liking.a.c.getAdId()

Email / Username

Unconditional source:
com.firsteapps.login.models.User.getEmail()
com.dozuki.ifixit.ui.auth.LoginFragment.getEmail()
com.pinnatta.models.UserProfile.getEmail()
com.global.guacamole.data.signin.UserAccountDetails.getEmai()
com.firsteapps.login.models.User.getFirstName()
com.pinnatta.models.UserProfile.getFirstName()

Conditional source:
com.crashlytics.android.core.CrashlyticsCore.sanitizeAttribute()
com.crashlytics.android.core.CrashlyticsCore.setUserEmail()

com.newrelic.agent.android.instrumentation.JSONObjectInstrumentation.toString()
com.appsflyer.AppsFlyerLib.setUserEmails()

Fig. 8. DAISY Sources

Distributive Subsets) static analysis framework, so that the calling context will be automatically matched in the data flow analysis in FlowDroid.

The analysis results show that FlowDroid reported 30 leaks coming from DAISY sources. The information types of the reported leaked DAISY sources include Android ID, Email, Serial number and User name. Table 7 column 1 shows the names of the apps where leaks were detected, and columbe 2 shows the number of leaks from DAISY sources. We performed further analysis to check whether those 30 leaks could be detected by CondySTA. As Table 7 colume 3 shows, 11 leaks from one app were detected by ConDySTA. In addition, to compare with SUSI sources, we ran FlowDroid with SUSI sources and sinks, and only one DAISY leak was detected. From the result, we can conclude that DAISY can effectively reduce false negatives by detecting leaks that are not detectable by existing approaches, a significant supplement for existing techniques.

5.9 Evaluation on Benchmarks

We evaluated DAISY on DroidBench [9], an open test suite for evaluating the effectiveness and accuracy of taint-analysis tools for Android apps. In particular, we applied call graph analysis on all apps in DroidBench and extracted all possible static calling contexts (within three levels) of String-type methods. Then, we manually labeled a dataset of 367 different static in-context methods. We split the dataset into five equal parts and performed five-fold cross validation on it (60% training, 20% validation, and 20% testing). Table 8 shows the evaluation results on the three different levels. It should be noted that IMEI is the only data type being used in DroidBench.

Table 7. Leaks from DAISYsources

App	DAISY	ConDySTA	SUSI
com.dozuki.ifixit	1	0	0
com.thirtysixyougames.google.boyfriendmaker	4	0	0
com.gamegarden.fk	11	11	0
com.pinnatta.android	4	0	0
com.miniclip.dinopets	4	0	1
com.yellowpages.androidtablet.ypmobile	1	0	0
com.nevosoft.mysteryville	4	0	0
com.zenga.zengatv	1	0	0
Total	**30**	**11**	**1**

Table 8. DAISY Performance on DroidBench

		Performance				Number of Traces			
InfoTypes	Level	Accuracy	Precision	Recall	F1	TP	FP	TN	FN
IMEI	Level 1	71.7%	60.0%	21.4%	31.6%	24	16	239	88
	Level 2	82.3%	74.7%	63.4%	68.6%	71	24	231	41
	Level 3	83.9%	77.3%	67.0%	71.8%	75	22	233	37
	Overall	79.3%	73.3%	50.6%	59.9%				

From the results, we can see that with a well-labeled training set, machine learning-based prediction of sensitive methods can achieve an overall accuracy of 79.3%, precision of 73.3%, recall of 50.6%, and F1 of 59.9%.

5.10 Subword embedding feature

To answer RQ5, evaluating the effectiveness of the subword embeddings, we retrained our models using the same training set without subword embeddings. The same test set was then applied for prediction. Table 9 and Table 10 show the precision of DAISY without subword embeddings on the high-confidence subset and the random subset respectively. In Table 9, column 1 presents the information type, while in Table 10, column 1 presents the app name. Columns 2-4 of both Table 9 and Table 10 present the precision for each information type / app with each length of calling context. In each cell, the number before the "/" symbol is the number of manually confirmed true positives, while the number after the "/" symbol is the number of reported in-context methods. Column 5 shows the overall precision, which is computed by combining data from Columns 2-4. We put the overall precision of DAISY into the last column in each table for comparison. In the high-confidence subset, four of the six information types of DAISY without subword embeddings have slightly lower precision than DAISY, while the other two information types have higher precision than DAISY. The overall precision of DAISY without subword embeddings is 82.9%, while DAISY is 77.9%. The two information types "imei" and "email" made the overall precision without subword embeddings higher than DAISY. In the random subset, DAISY without subword embeddings has the same overall precision compared with DAISY.

It should be noted that theoretically the subword embeddings allow more flexible matching of words when detecting sensitive methods. Since such additional word matches may be either proper or improper (the training process will give the proper ones more weight but improper noises may still exist), the subword embeddings generally will not enhance precision, especially for the high-confidence instances because they can already be confirmed with whole words. However, the subword embeddings can help detect sensitive methods that are not

Table 9. DAISY without subword embeddings: Precision on the High-Confidence Subset

	DAISY without subword embeddings				DAISY
InfoType	Level 1	Level 2	Level 3	Overall	Overall
adsId	18/20	20/20	19/20	95.0%	97.1%
androidId	15/20	12/20	20/20	78.3%	80.0%
email	19/20	17/20	17/20	88.3%	71.7%
imei	9/16	16/20	20/20	80.4%	52.5%
serial	13/20	17/20	17/20	78.3%	85.0%
username	12/20	14/20	20/20	76.7%	78.3%
overall	86/116	96/120	113/120	82.9%	77.9%

Table 10. DAISY without subword embeddings: Precision on the Random Subset

	DAISY without subword embeddings				DAISY
App	Level 1	Level 2	Level 3	Overall	Overall
com.simplygood.ct	1/2	2/4	3/7	46.2%	50.0%
com.skimble.fitnessflow.lite	0/0	0/1	1/6	14.3%	25.0%
com.zenga.zengatv	1/1	6/10	4/13	45.8%	31.4%
com.mlssoccer	3/4	6/9	13/17	73.3%	65.1%
com.oki.letters	2/4	0/3	4/12	31.6%	20.6%
com.mtvn.mtvPrimeAndroid	9/16	13/30	36/92	42.0%	47.4%
com.klab.lods.en	3/4	1/7	5/13	37.5%	33.3%
com.williamsinteractive.goldfish	1/3	5/8	10/26	43.2%	39.1%
com.kidoz	2/4	6/7	6/10	66.7%	70.8%
com.nevosoft.mysteryville	1/3	2/10	1/8	19.0%	37.0%
Overall	23/41	41/89	83/204	44.0%	44.0%

detectable by matching whole words only. As shown in Table 11, DAISY reported more sensitive methods at most levels for most information types. In total, DAISY without subword embeddings reported 17,323 sensitive methods while DAISY predicted 26,927. Since Table 10 shows that DAISY achieves a similar precision as DAISY without subword embeddings, we can see that the subword embeddings allowed our model to identify more true sensitive methods.

5.11 Threats to Validity

The main threat to the internal validity of our evaluation is in the selection and labeling of reported in-context methods. To reduce the bias and errors in the labelling process, we have two people to perform manual labelling independently, and a third people to resolve conflicts. The main threat to the external validity of our evaluation is that the results may apply to only the test set and the labeled dataset. To provide a more comprehensive evaluation of DAISY, we used two different selection mechanisms, corresponding to two usage scenarios of DAISY. The evaluation results on two selected subsets are reasonably consistent as they both show high discrimination power of the classification models, but the results on the random subset is lower than those on the high-confidence subset, which is as expected.

Table 11. Number of sensitive methods reported by DAISY wihtout subword embeddings and DAISY

		DAISY without subword embeddings	DAISY
level1	androidId	374	484
	email	354	386
	username	59	131
	imei	33	0
	adsId	960	1128
	serial	67	108
	Total	**1847**	**2237**
level2	androidId	1154	1534
	email	839	1550
	username	144	685
	imei	203	183
	adsId	2088	3059
	serial	144	203
	Total	**4572**	**7214**
level3	androidId	2120	3042
	email	1565	5252
	username	2581	2241
	imei	194	510
	adsId	4250	6091
	serial	194	340
	Total	**10904**	**17476**
Total		**17323**	**26927**

6 DISCUSSION

6.1 Extension of Our Approach

6.1.1 Domain-Specific Information Types. In our research, we consider top Android apps ranked by PlayDrone and common sensitive personal information such as email address. However, our approach can also be applied to domain-specific information types such as transaction information in financial apps or grading information in education apps. To apply our approach to those information types, we can collect app sets in such domains. Furthermore, to address the potential sparsity problem for domain-specific information types, we plan to leverage transfer learning [75] which incorporates a model trained from a large general data set (e.g., general information type data set), and adjusts the weights and parameters of the model based on a smaller adaptation data set from the specific domain.

6.1.2 Applications Beyond Privacy Leak Detection. Besides privacy leak detection, our approach may have other applications in security and software engineering. For example, after the sensitive methods are detected, we can add monitors to monitor their behaviors at runtime or during fuzzing to check whether proper security mechanisms (e.g., encryption, anonymization, and permission checking) have been applied. More generally, if we expand the information types beyond sensitive information, our approach may help developers to search for the methods that handle certain data of interest and better understand the code base.

6.2 Evaluation Strategies

As it is time-intensive to manually verify and label all reported sensitive in-context methods, we manually validate a subset of the methods. We chose two approaches to select subsets. In the first selection strategy, we chose the top 20 most likely (i.e., highest probability) predictions made by DAISY for each of the six information types under each of the three context lengths. This resulted in 340 predictions for manual labeling and validation and represents the effectiveness of DAISY on the in-context methods that it is most confident are sensitive. This highest-confidence subset is useful to show DAISY's effectiveness for processing batches of apps, which is of interest to third-party audits and quality and security assurance. For the second selection strategy, we selected 10 apps and randomly sampled 20% of DAISY predictions from those apps for manual labeling and validation, resulting in 452 in-context methods.

There were two other selection strategies we considered. The first was a confidence threshold strategy where separate evaluations were reported at different confidences (i.e., different prediction probability thresholds) to show the precision of DAISY at those confidences. Unfortunately, due to the size of the number of sensitive in-context predictions made it was infeasible to perform a manual labeling and validation of even the top confidence thresholds (i.e., 100% or 99% confidence). Instead, we chose the highest-confidence strategy in order to show DAISY's effectiveness at its maximum confidence. The second was a random sampling of the predictions. Again, due to the number of sensitive in-context predictions made it would be infeasible to attempt a manual labeling and validation of more than 1% of the total predictions. This sampling would be too small to guarantee a good representation of all ranks of apps throughout the dataset. We chose to perform the ranked selection strategy instead in order to have a more accurate representation of the dataset.

6.3 Limitations

6.3.1 Obfuscated Code. DAISY is designed for non-obfuscated code and the performance of DAISY could be affected when predicting an obfuscated in-context method, depending on the level of obfuscation. DAISY can handle lightly obfuscated method signatures with meaningful words. For instance, DAISY correctly predicts `"com.appsflyer.f.getAdvertisingId"` as a sensitive method that returns the user's advertising ID. However, DAISY cannot identify a heavily obfuscated sensitive method. For example, method `"com.google.a.b.c.d()"` will be predicted as non-sensitive. We consider this acceptable as our approach is primarily designed for developers and privacy liaisons to make their code better comply with privacy policies. Two usage scenarios were considered in this work. In the first scenario, DAISY is used by the app developer / app producing organization who wants to avoid privacy policy violations by examining all suspicious methods. Since developers have the source code, obfuscation is not an issue. In the second scenario, DAISY is used for app scanning at app stores / security analysis services. Analyzers, such as privacy auditors from the Google Play Store, might require the app developers to upload non-obfuscated version of the code for security analysis. In summary, although heavily obfuscated sensitive methods may not be identified by DAISY, they could be reasonably avoided in real-world usage scenarios.

6.3.2 Encryption. Due to the nature of value-based dynamic taint analysis, encrypted values will be missed when creating a training data set. In other words, if a method returns encrypted sensitive data, it is not considered a source. The goal of identifying sources is to use them in taint analysis to see if they flow to sinks, and taint flow of encrypted values is usually of less concern.

6.3.3 Special Code. As part of the static analysis, we use ANDROGUARD to create a call graph for each app and extract calling context of each method. ANDROGUARD is able to extract native methods, but since is a static analysis based tool, it lacks handling of reflection and dynamic code loading. Cloaking methods are methods that can hide their behaviors during testing, such as a method that randomly returns sensitive information once in

1,000 invocations. Such methods may pollute our training data collected through Monkey testing and return value checking. However, due to their rarity and the huge size of our training data, their influence could be limited as evident by our promising results. They will not affect our prediction phase because the phase relies on the static calling context instead of return values.

7 RELATED WORKS

In this section, we introduce the related techniques for analyzing mobile apps, including different types of sources in taint analysis, using call stack and code elements for machine learning tasks.

7.1 Android API sources

Privacy violations caused by sensitive data leakage in Android applications are well known in the community. To protect the user's privacy, various approaches for tracking tainted data have been proposed, both statically [9, 12, 25, 30, 38, 43, 45, 46, 48, 70, 73, 74] and dynamically [20, 60, 63, 76]. Those approaches, however, rely on the manual configuration of lists of sources of sensitive data as well as sinks that may leak data to untrusted observers.

The Android platform provides a large-scale application programming interface (API) to support application development. This API enables the developer to access the system's features and resources such as user data, settings, and hardware. To access the user's data, once the user grants permissions, the app could access the device's corresponding resources through the Android resources API. For example, if the LOCATION permission has been granted, the developer could call `Android.Location.LocationManager.getLastLocation ()` to retrieve the device location. If the READ_PHONE_STATE permission has been granted, developers could call `Android.telephony.TelephonyManager.getDeviceId()` to retrieve the device's IMEI. The Android API methods that are used to access user's sensitive data have been used as sources in many data flow analysis tools to assess how apps use private user data. These analysis tools either construct the list of sources manually [30, 48] or semi-automatically based on Android permissions [45, 73]. However, it is difficult to classify all sources manually or semi-automatically given the large, continuously growing number of public methods in the Android API. [54] thus propose SUSI, a machine-learning-guided approach for identifying sources and sinks from existing Android APIs. The identified sources and sinks have enabled many studies on the detection of privacy leaks in Android apps [10, 12, 24, 29, 43, 44, 59]. To supplement SUSI, recent work [61] collected fields and methods related to sensors as sources for taint analysis and extended FlowDroid to support field sources to detect sensor-based data leaks in Android apps. However, Android API sources can only cover the sources of device data. Sensitive data can come from a variety of sources, including user input, servers, and other systems. In order to detect as many as possible privacy leaks, we expect to identify as many sources as possible for data flow analysis. Unlike previous works, the goal of DAISY is to identify sources beyond the Android API: sensitive methods defined by apps or third-party libraries.

7.2 Non-Android API sources

In order to reduce false negatives in data flow analysis caused by missing sources, ConDySTA [79] uses dynamic analysis results to identify non-Android API sources and leaks caused by them. ConDySTA identifies sources by observing the String type return value of a method during GUI testing. Since ConDySTA's dynamic analysis is based on GUI testing, it is limited by test coverage and will miss sources that are not triggered during testing. Furthermore, it requires human effort and thus cannot be applied to a large number of apps. Different in both scope and approach from ConDySTA, DAISY utilizes machine learning techniques to fully automatically identify sources based on method calling context. It detects sources among all the source code statically without running the app or any human effort.

PAMDroid [78] detects PII (personally identifiable information) leaks to third-party services using dynamic analysis. They observe the data flows to the sink methods defined by third-party services. [35, 50, 68] considered user input data as sources, which cannot be identified using the Android API methods alone and requires tracing potential sensitive data through GUI API method executions (e.g., `android.widget.EditText.getText()`). This requires classifying the input data's information types by first analyzing the GUI hierarchy [56] and then classifying labels associated with the method invocations. DAISY instead aims to identify all sensitive methods by statically analyzing the app's program.

7.3 Call stack analysis for machine learning tasks

To our knowledge, DAISY is the first approach that uses automatically labeled call stacks to train a classification model to classify sensitive information types of methods. We are aware of the following works which use machine learning and other techniques for the detection of specific runtime behavior based on call stacks.

Hou et al. use a combination of deep learning and dynamic analysis to detect malware in Android apps. Their system, Deep4MalDroid, uses a dynamic analysis approach called *Component Traversal* to maximize the code executed for a given app [34]. A deep learning framework is then applied over the resulting graph to identify malware based on Android system calls. While deep learning and program analysis are used by Deep4MalDroid to detect security-related constructs, the system entails a complete, or nearly complete, directed graph of the app in question, rather than an individual, arbitrary method. Furthermore, detection and classification of private data types is not within the scope of their work.

Xie et al. use dynamic analysis to detect anomalous runtime behavior in high-performance computing systems [72]. The approach builds call stack tree representations (*CSTrees*) as feature vectors from the stacks and applies a One-Class Support Vector Machine to detect anomalies. The resulting visual structures can then be reviewed by a human to validate candidate anomalies. Unlike our approach, the detection of such anomalous behavior requires a broader context and human intervention (which is necessary for the problem space). As our approach seeks to detect and label sensitive data usage as a result of a single series of calls, only the call stack in question is necessary.

The analysis of call stacks has been used before to detect runtime anomalies. For example, Brodie et al. use machine learning to identify similar runtime problems as they reoccur [13]. The primary application of this technique is in situations where humans are employed to detect problems with software (e.g., help-desks). The technique treats call stacks from which the problem arises as a symptom or signature which can then be identified in later executions. The system can then notify the human of the issue and suggest resolutions. Similarly, Feng et al. analyze call stacks to detect security exploits [21]. In their approach, the call stack and program counter are used to generate abstract execution paths which can be compared to well-known behavior in previous executions of the program. These works are similar to ours in that they use call stacks to identify and classify program behavior, however our method can be more easily applied to arbitrary call stacks due to our machine learning approach. Furthermore, DAISY is designed specifically for the detection of data sources rather than problematic behavior or exploitation.

7.4 Machine learning on code units

Previous studies [26, 33] have indicated that a similarity exists between code and natural language, and that code is even less surprising (i.e., it follows expected patterns more) than normal natural language text. This suggests that NLP technologies can also be applied to solve problems in analyzing code. Many machine learning based approaches have been proposed for the classification of elementary code units such as methods and variables.

SUSI [54] uses machine learning to identify Android API methods that retrieves sensitive information. [53] proposed a machine learning approach to detect sources, sinks, validators, and authentication methods for Java programs.‘ Code2Vec [6] trains code embeddings to infer high level semantics of code, and their evaluation is performed via method name prediction, where they generate method names from the method code. Vasilescu et al. [64] proposed a machine learning based approach to infer variable names from obfuscated code. Russell et al. [57] performs a vulnerability detection task on C/C++ code. Two datasets are created using a combination of static analysis and manual inspection to label code functions as “vulnerable” or “not vulnerable”. Source code is parsed into a vocabulary of code elements and reduced to 156 representative tokens. Word embeddings are then constructed for these representative tokens which are used in a convolution neural network (CNN) classifier to learn and predict if a given function is vulnerable or not vulnerable. Heaps et al. [32] propose to construct word embeddings for code functions by utilizing the function definition as input to a recurrent neural network (RNN). The word embeddings produced by this approach would more accurately represent the semantics of a function rather than learning the semantics based on the usage of the function. These word embeddings can then be used for vulnerability detection. Compared with these approaches, our research focuses on the classification of methods, and we use dynamic analysis to automatically build our data set.

8 CONCLUSION

In this paper, we present an automated approach for discovering additional sensitive information sources based on a training set of labeled methods with their calling contexts. We further show that this training set can be automatically constructed and labeled using dynamic analysis. We train and evaluate our approach on a data set collected from 300 Android apps. The evaluation results show that our approach achieved an overall precision of 77.9% in 340 sources reported with the highest confidence, and an overall precision of 44.0% in 452 sources randomly sampled from all the reported results of 10 apps in the test set. We manually labeled these 792 sources and confirmed that 464 are real sources. Our further analysis shows that very few of the sources DAISY reports can be detected by existing static and dynamic approaches, and increasing calling-context does help to find more conditional sources. In addition, we applied the 464 confirmed DAISY sources in taint analysis to detect data leaks and identified 30 leaks, 18 of which are undetectable by existing technologies.

As future work, we plan to extend our research in the following directions. First, we plan to evaluate our approach on a larger data set. Second, we plan to enhance our classification accuracy by adding more features from the code, such as the smali code of the methods and other fields / methods declared in the same class. Third, we plan to work on domain-specific information types by constructing domain-specific data sets and performing transfer learning to adapt our general model to specific domains.

9 ACKNOWLEDGMENTS

This work was supported by the NSF Award 1948244, 1736209, 2007718, 1846467, and 2221843.

REFERENCES

[1] 2018. *GDPR definition of personal data.* Retrieved October, 2018 from https://gdpr-info.eu/art-4-gdpr/

[2] 2018. *GDPR online identifiers for profiling and identification.* Retrieved October, 2018 from https://gdpr-info.eu/recitals/no-30/

[3] Consumer Data Protection Act. 2021. The Virginia Consumer Data Protection Act (CDPA). https://lis.virginia.gov/cgi-bin/legp604.exe?211+sum+SB1392

[4] The California Consumer Privacy Act. 2018. The California Consumer Privacy Act of 2018 (CCPA). https://oag.ca.gov/privacy/ccpa

[5] Prerna Agrawal and Bhushan Trivedi. 2020. Unstructured data collection from APK files for malware detection. *International Journal of Computer Applications* 975 (2020), 8887.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[7] Benjamin Andow, Akhil Acharya, Dengfeng Li, William Enck, Kapil Singh, and Tao Xie. 2017. Uiref: analysis of sensitive user inputs in android applications. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 23–34.

[8] 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. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 259–269.

[9] 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 (2014), 259–269.

[10] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining apps for abnormal usage of sensitive data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 426–436.

[11] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics* 5 (2017), 135–146.

[12] Amiangshu Bosu, Fang Liu, Danfeng Yao, and Gang Wang. 2017. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 71–85.

[13] Mark Brodie, Sheng Ma, Leonid Rachevsky, and Jon Champlin. 2005. Automated problem determination using call-stack matching. *Journal of Network and Systems Management* 13, 2 (2005), 219–237.

[14] Hsin-Yu Chuang and Sheng-De Wang. 2015. Machine learning based hybrid behavior models for Android malware analysis. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 201–206.

[15] Federal Trade Commission. 1998. Children's Online Privacy Protection Act of 1998 (COPPA). https://www.ftc.gov/enforcement/rules/rulemaking-regulatory-reform-proceedings/childrens-online-privacy-protection-rule

[16] Anthony Desnos. 2015. Androguard. https://github.com/androguard/androguard

[17] Android Developers. 2012. Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey.html

[18] 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* (2018).

[19] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 73–84.

[20] 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 (2014), 1–29.

[21] Henry Hanping Feng, Oleg M Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. 2003. Anomaly detection using call stack information. In *2003 Symposium on Security and Privacy, 2003.* IEEE, 62–75.

[22] Ruitao Feng, Sen Chen, Xiaofei Xie, Lei Ma, Guozhu Meng, Yang Liu, and Shang-Wei Lin. 2019. Mobidroid: A performance-sensitive malware detection system on mobile platform. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 61–70.

[23] Centers for Medicare & Medicaid Services. 1996. The Health Insurance Portability and Accountability Act of 1996 (HIPAA). https://www.hhs.gov/hipaa/for-professionals/privacy/laws-regulations/combined-regulation-text/index.html

[24] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2016. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 377–396.

[25] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. 2009. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland, http://www.cs.umd.edu/avik/projects/scandroidascaa* 2, 3.

[26] Mark Gabel and Zhendong Su. 2010. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. 147–156.

[27] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. 2020. Borrowing your enemy's arrows: the case of code reuse in android via direct inter-app code invocation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 939–951.

[28] Xiuting Ge, Ya Pan, Yong Fan, and Chunrong Fang. 2019. AMDroid: android malware detection using function call graphs. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 71–77.

[29] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. 2012. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *International Conference on Trust and Trustworthy Computing*. Springer, 291–307.

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

[31] Yongzhong He, Xuejun Yang, Binghui Hu, and Wei Wang. 2019. Dynamic privacy leakage analysis of Android third-party libraries. *Journal of Information Security and Applications* 46 (2019), 259–270.

[32] John Heaps, Xiaoyin Wang, Travis Breaux, and Jianwei Niu. 2019. Toward Detection of Access Control Models from Source Code via Word Embedding. In *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies*. 103–112.

[33] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[34] Shifu Hou, Aaron Saas, Lifei Chen, and Yanfang Ye. 2016. Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs. In *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*. IEEE, 104–111.

[35] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. 2015. {SUPOR}: Precise and Scalable Sensitive User Input Detection for Android Apps. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 977–992.

[36] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2016. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759* (2016).

[37] Md Yasser Karim, Huzefa Kagdi, and Massimiliano Di Penta. 2016. Mining android apps to recommend permissions. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 427–437.

[38] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. 1–6.

[39] JD Koli. 2018. RanDroid: Android malware detection using random machine learning classifiers. In *2018 Technologies for Smart-City Energy Security and Power (ICSESP)*. IEEE, 1–6.

[40] Pingfan Kong, Li Li, Jun Gao, Timothée Riom, Yanjie Zhao, Tegawendé F Bissyandé, and Jacques Klein. 2021. ANCHOR: locating android framework-specific crashing faults. *Automated Software Engineering* 28, 2 (2021), 1–31.

[41] Daniel E Krutz, Mehdi Mirakhorli, Samuel A Malachowsky, Andres Ruiz, Jacob Peterson, Andrew Filipski, and Jared Smith. 2015. A dataset of open-source android applications. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 522–525.

[42] Christophe Leung, Jingjing Ren, David Choffnes, and Christo Wilson. 2016. Should you use the app for that? comparing the privacy implications of app-and web-based online services. In *Proceedings of the 2016 Internet Measurement Conference*. 365–372.

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

[44] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. 2014. Automatically exploiting potential component leaks in android applications. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 388–397.

[45] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 229–240.

[46] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. 229–240.

[47] Samin Yaseer Mahmud, Akhil Acharya, Benjamin Andow, William Enck, and Bradley Reaves. 2020. Cardpliance:{PCI}{DSS} Compliance of Android Applications. In *29th USENIX Security Symposium (USENIX Security 20)*. 1517–1533.

[48] Christopher Mann and Artem Starostin. 2012. A framework for static detection of privacy leaks in android applications. In *Proceedings of the 27th annual ACM symposium on applied computing*. 1457–1462.

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

[50] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. 2015. Uipicker: User-input privacy identification in mobile applications. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 993–1008.

[51] Haofei Niu, Tianchang Yang, and Shaozhang Niu. 2016. Clone analysis and detection in android applications. In *2016 3rd International Conference on Systems and Informatics (ICSAI)*. IEEE, 520–525.

[52] European Parliament and The Council of the European Union. 2016. General Data Protection Regulation (GDPR). https://gdpr-info.eu/

[53] 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.

[54] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A machine-learning approach for classifying and categorizing android sources and sinks.. In *NDSS*, Vol. 14. Citeseer, 1125.

[55] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 49–61.

[56] A Rountev, D Yan, S Yang, H Wu, Y Wang, and H Zhang. 2017. GATOR: Program analysis toolkit for Android. http://web.cse.ohio-state.edu/presto/software/gator/

[57] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 757–762.

[58] Justin Sahs and Latifur Khan. 2012. A machine learning approach to android malware detection. In *2012 European Intelligence and Security Informatics Conference*. IEEE, 141–147.

[59] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D Breaux, and Jianwei Niu. 2016. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International Conference on Software Engineering*. 25–36.

[60] Mingshen Sun, Tao Wei, and John Lui. 2016. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 331–342.

[61] Xiaoyu Sun, Xiao Chen, Kui Liu, Sheng Wen, Li Li, and John Grundy. 2021. Characterizing Sensor Leaks in Android Apps. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 498–509.

[62] Xiaoyu Sun, Li Li, Tegawendé F Bissyandé, Jacques Klein, Damien Octeau, and John Grundy. 2021. Taming reflection: An essential step toward whole-program analysis of android apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–36.

[63] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *22nd Annual Network and Distributed System Security Symposium*.

[64] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. 2017. Recovering clear, natural identifiers from obfuscated JS names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 683–693.

[65] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A measurement study of google play. In *The 2014 ACM international conference on Measurement and modeling of computer systems*. 221–233.

[66] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of android test generation tools in industrial cases. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 738–748.

[67] Wei Wang, Ruoxi Sun, Minhui Xue, and Damith C Ranasinghe. 2020. An automated assessment of Android clipboards. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1249–1251.

[68] Xiaoyin Wang, Xue Qin, Mitra Bokaei Hosseini, Rocky Slavin, Travis D Breaux, and Jianwei Niu. 2018. Guileak: Tracing privacy policy claims on user input data for android applications. In *Proceedings of the 40th International Conference on Software Engineering*. 37–47.

[69] Zhaoguo Wang, Chenglong Li, Yi Guan, and Yibo Xue. 2015. Droidchain: A novel malware detection method for android based on behavior chain. In *2015 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 727–728.

[70] Fengguo Wei, Sankardas Roy, and Xinming Ou. 2018. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS)* 21, 3 (2018), 1–32.

[71] R Winsniewski. 2012. Android–apktool: A tool for reverse engineering android apk files.

[72] Cong Xie, Wei Xu, and Klaus Mueller. 2018. A visual analytics framework for the detection of anomalous call stack trees in high performance computing applications. *IEEE transactions on visualization and computer graphics* 25, 1 (2018), 215–224.

[73] Zhemin Yang and Min Yang. 2012. Leakminer: Detect information leakage on android with static taint analysis. In *2012 Third World Congress on Software Engineering*. IEEE, 101–104.

[74] Zhemin Yang and Min Yang. 2012. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In *Proceedings of the 2012 Third World Congress on Software Engineering*. 101–104.

[75] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. How transferable are features in deep neural networks?. In *Advances in neural information processing systems*. 3320–3328.

[76] Wei You, Bin Liang, Wenchang Shi, Peng Wang, and Xiangyu Zhang. 2017. TaintMan: an ART-compatible dynamic taint analysis framework on unmodified and non-rooted Android devices. *IEEE Transactions on Dependable and Secure Computing* (2017).

[77] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. 2014. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. 25–36.

[78] Xueling Zhang, Xiaoyin Wang, Rocky Slavin, Travis Breaux, and Jianwei Niu. 2020. How does misconfiguration of analytic services compromise mobile privacy?. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1572–1583.

[79] Xueling Zhang, Xiaoyin Wang, Rocky Slavin, and Jianwei Niu. 2021. ConDySTA: Context-Aware Dynamic Supplement to Static Taint Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*.