



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

Journal:	<i>Transactions on Software Engineering and Methodology</i>
Manuscript ID	TOSEM-2022-0100
Manuscript Type:	Journal-First Paper
Date Submitted by the Author:	17-Apr-2022
Complete List of Authors:	Zhang, Xueling; Rochester Institute of Technology, Software Engineering Heaps, John; The University of Texas at San Antonio Slavin, Rocky; The University of Texas at San Antonio Niu, Jianwei; The University of Texas at San Antonio Breaux, Travis; Carnegie Mellon University, Institute for Software Research Wang, Xiaoyin; University of Texas at San Antonio, Department of Computer Science
Computing Classification Systems:	Software and its engineering, Security and privacy

April 17th, 2022

Dear Dr. Pezzè,

We wish to submit an original research article entitled "DAISY: Dynamic-Analysis-Induced Source Discovery for Sensitive Data" for consideration by the ACM Transactions on Software Engineering and Methodology.

In this paper, we propose a novel approach to identify methods for accessing sensitive data defined by apps and third-party libraries. Our goal is to reduce false negatives in taint analysis (a sensitive data flow or leak exists, but is undetectable) caused by incomplete sources, which describe the starting point in the flow and can be represented as a method signature. Such source methods are difficult to collect because they are unique to the apps, and there are many third-party libraries available on the market that evolve over time. To address this challenge, we propose a novel approach that leverages static and dynamic program analysis, as well as machine learning. The evaluation shows that our approach can effectively identify sources with decent precision and detect privacy leaks from reported sources that are not detectable by existing techniques, significantly reducing false negatives in taint analysis for privacy leaks.

Many previous researchers have collected source methods from the Android platform by identifying Android API methods that allow access to sensitive data. Different with those works, the goal of this work is to identify sources beyond Android APIs, sensitive methods defined by apps, or third-party libraries. Our prior work identifies app and third-party sources by observing the String type return value of a method using dynamic analysis. Because its dynamic analysis is based on GUI testing, it is limited by test coverage and will miss sources that are not triggered during testing. Besides, it requires human effort and thus cannot be applied to a large number of apps. This work differs from our previous work in both scope and approach since it 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.

We believe that this manuscript is appropriate for publication by ACM Transactions on Software Engineering and Methodology because the journal values contributions that identify significant challenges for software engineering research and creative approaches to addressing those challenges.

We confirm that this work is original and has not been previously published elsewhere, or is currently under being consideration for publication elsewhere.

Please send all correspondence concerning this manuscript to me at xueling.zhang@rit.edu

Thank you for your consideration of this manuscript.

Sincerely,

Dr. Xueling Zhang

Assistant Professor

Department of Software Engineering

Rochester Institute of Technology

xueling.zhang@rit.edu

Co-author

Dr. John Heaps

Assistant Professor of Instruction

Department of Computer Science

The University of Texas at San Antonio

john.heaps@utsa.edu

Rocky Slavin

Assistant Professor

Department of Computer Science

The University of Texas at San Antonio

rocky.slavin@utsa.edu

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60

Jianwei Niu  
Professor  
Department of Computer Science  
The University of Texas at San Antonio  
jianwei.niu@utsa.edu

Travis D. Breaux  
Associate Professor  
Computer Science Department  
Carnegie Mellon University  
breaux@cs.cmu.edu

Xiaoyin Wang  
Associate Professor  
Department of Computer Science  
University of Texas at San Antonio  
xiaoyin.wang@utsa.edu

For Peer Review

## 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. BREAU, 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, 12, 17, 37] 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 platform, third-party libraries, and custom app code. To comply with

---

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.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2022 Association for Computing Machinery.

Manuscript submitted to ACM

Manuscript submitted to ACM

2 Xueling Zhang, John Heaps, Rocky Slavin, Jianwei Niu, Travis D. Breaux, and Xiaoyin Wang

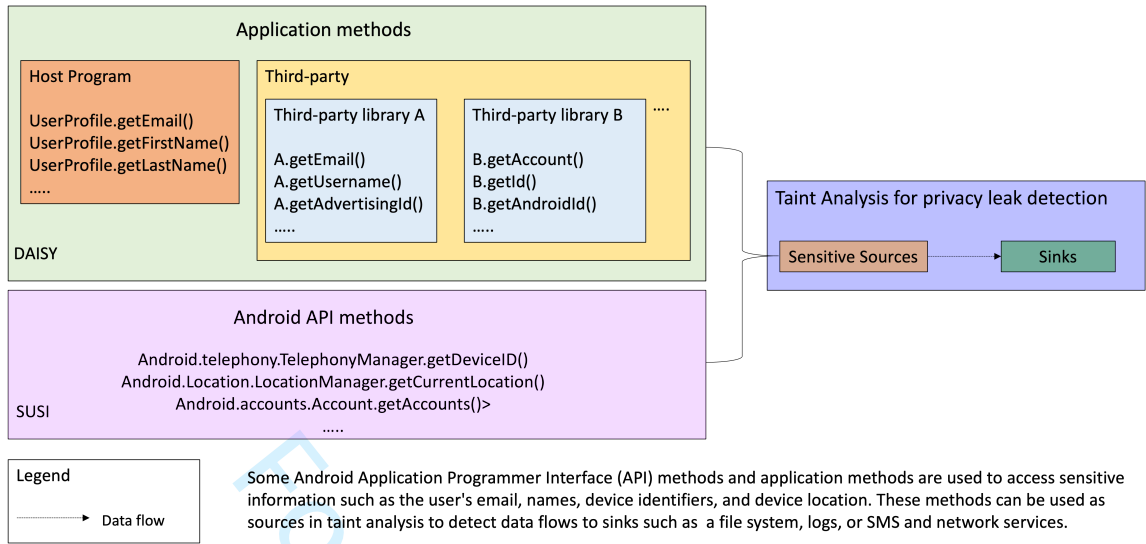


Fig. 1. Candidate sources for Android apps

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 [6, 15] 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 [34] or semi-automatically [32, 52] from the Android API methods. Given the large number of public methods (more than 100,000) in the APIs, researchers have been unable to classify all sources manually or semi-automatically. To supplement these original source lists, Rasthofer et al. [38] proposed SUSI, an automated machine learning-based approach for classifying sources and sinks from the Android platform API, 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 [20, 42], and third-party libraries [22].

While SUSI identifies sensitive sources in the Android platform API, there is a second category of sensitive sources that exists when data input flows through graphical user interfaces (GUI), which use Android platform API methods that are not labeled as sensitive in SUSI. Recent work [26, 36, 48] demonstrates how user input data can be traced through GUI API method executions (e.g., `android.widget.EditText.getText()`). 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 [40] and next to classify labels associated with the method invocations.

Aside from the Android platform API, including the GUI API, sensitive user data can be accessed by third-party services 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 platform API is 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 platform API, 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 platform 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 and leaks, our prior work, ConDySTA [57], 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

4 Xueling Zhang, John Heaps, Rocky Slavin, Jianwei Niu, Travis D. Breaux, and Xiaoyin Wang

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(...)`, `UserAuthHelper.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 sub-word embedding feature of the FASTTEXT [9, 27] framework. Sub-word embedding considers sub-strings of words when constructing word embeddings so that an unseen word with sub-strings seen in training can be properly embedded.

```
1 //Return Value:
2 xxxxxxxxxxxx@gmail.com
3 //Call Stack:
4 com.tubitv.helpers.PreferenceHelper.getString(PreferenceHelper.java:2)
5 com.tubitv.helpers.PreferenceHelper.getString(PreferenceHelper.java:3)
6 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 [46]. 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 (high-confidence set) consists of 340 in-context methods which are predicted by DAISY with the highest confidence for different considered context lengths and information types (up to top 20 for each combination). This 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 (random) set consists of 452 in-context methods which are randomly sampled (with 20% rate) from in-context methods of 10 apps (apps ranked 200, 210, 220, ... 290) in our testing app set. This 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 set and an average precision of 44.0% for the random set. 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 sub-word embedding.



- 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, we introduce DAISY and our machine learning model, followed by our evaluation setup and results in Section 4. Finally, we discuss important issues in Section 5 and related works in Section 6 before concluding in Section 7.

## 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 using program. 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 [7, 10, 18, 21, 28, 30, 32–34, 49, 52, 53], or dynamic [15, 43, 44, 55] 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 through the Android API method `getDeviceId()` (on Line 4), which is the source, and the the device ID is then sent out through the Android API method `sendTextMessage` (on Line 7), which is the sink.

6 Xueling Zhang, John Heaps, Rocky Slavin, Jianwei Niu, Travis D. Breaux, and Xiaoyin Wang

```
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 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. Listing 3 is an example of a call stack. In DAISY, we automatically collect call stacks as training set for our learning models.

```
1 ----- beginning of crash
2 FATAL EXCEPTION: main
3 Process: com.zenga.zengatv, PID: 13218
4 android.view.WindowManager$BadTokenException: Unable to add window -- token android.os.
   BinderProxy@189f3cf is not valid; is your activity running?
5   at android.view.ViewRootImpl.setView(ViewRootImpl.java:679)
6   at android.view.WindowManagerGlobal.addView(WindowManagerGlobal.java:342)
7   at android.view.WindowManagerImpl.addView(WindowManagerImpl.java:93)
8   at android.widget.Toast$TN.handleShow(Toast.java:459)
9   at android.widget.Toast$TN$2.handleMessage(Toast.java:342)
10  at android.os.Handler.dispatchMessage(Handler.java:102)
11  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.

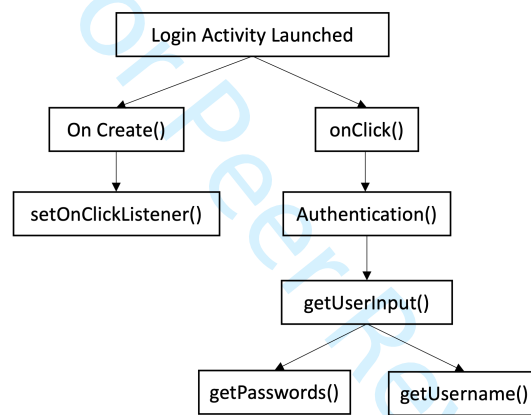


Fig. 2. A sample call graph in login activity

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 intends to predict whether getUsername() or getPassword() are sensitive or not based on their calling context.

## 3 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 3 shows the overview of our approach

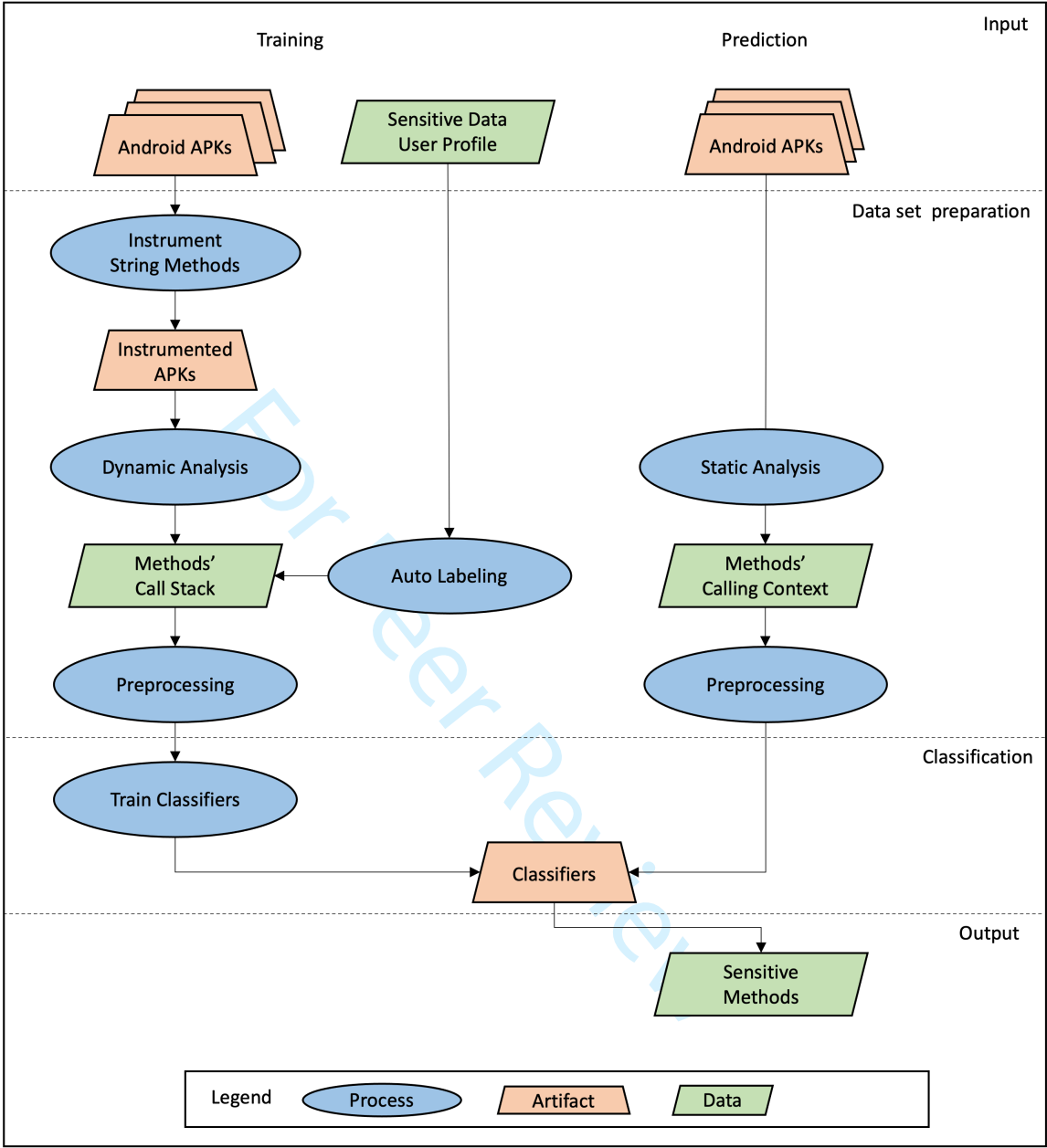


Fig. 3. Approach Overview

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 3.1 describes the input data for our approach, which includes the apps used for training and prediction, as well

Manuscript submitted to ACM

Table 1. Sensitive Data User Profile

Info Type	Sample Value
Advertising Id	"fc1303d8-7fbb-44d8-8a68-a79ffac06fea"
Android Id	"a54ecb914c21863"
Email	"*****@gmail.com"
IMEI	"355458061189396"
Serial	"ZX1G22KHQK"
User Name	"*****"

as the user profile we created for labeling. Section 3.2 describes how we use dynamic analysis to generate training data and automatically label them. Section 3.3 describes the process of training our classifiers. Section 3.4 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.

### 3.1 Input

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 3, serves as an input for the auto-labeling process. The details of how we use it for labeling will be discussed in Section 3.2. 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 an 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 defined by the EU General Data Protection Regulation (GDPR) [1, 2]. In addition, these six information types are well studied, prior works [29, 56, 57] consider them as important sensitive data in privacy protection. The input of the testing phase is a set of apps to be analyzed for identifying sensitive methods in them.

### 3.2 Training Set Preparation

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 noises as encryption and obfuscation are typically performed only when data is sent out. The automatic labeling process consists of four steps describe below.

10 Xueling Zhang, John Heaps, Rocky Slavin, Jianwei Niu, Travis D. Breaux, and Xiaoyin Wang

3.2.1 *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 3). 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 [50] to decompile APK files into smali code<sup>1</sup>. As seen in Figure 3, the resulting **Instrumented APKs** are then used as input to the next step to generate call stacks.

3.2.2 *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 3). We use the Android Debug Bridge (adb) to automatically install the rebuilt apps onto our test device and run MONKEY[14] to perform the testing on the apps. 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 created 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 methods that was triggered, its return value and call stacks will be 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.AttributionIdentifiers.getAndroidAdvertiserId()), 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 3), 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

3.2.3 *Automatic Labeling.* We label a call stack based on its return value (**Auto Labeling** in Figure 3). 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.getAndroidAdvertiserId(). line 1

<sup>1</sup>Assembler for the dex format used by Dalvik

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.sanitizeAttribute()` returns the user's email address when it was called by method `com.crashlytics.android.core.CrashlyticsCore.setUserEmail()`, but it returns a non-sensitive information when it was called by another methods. To address the challenge of same method returning different value 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 4, 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 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.

---

```

1 //Return sensitive data
2 Return Value: *****@gmail.com
3 Call Stack:
4 com.crashlytics.android.core.CrashlyticsCore.sanitizeAttribute(CrashlyticsCore.java:845)
5 com.crashlytics.android.core.CrashlyticsCore.setUserEmail(CrashlyticsCore.java:528)
6 com.fitradio.ui.login.task.BaseLoginJob.handleLoginResponse(BaseLoginJob.java:146)
7 com.fitradio.ui.login.task.EmailLoginJob.getUserLoginEvent(EmailLoginJob.java:68)
8 com.fitradio.ui.login.task.BaseLoginJob.onRunRun(BaseLoginJob.java:67)
9
10 //Return non-sensitive data
11 Return Value: expiration_date
12 Call Stack:
13 com.crashlytics.android.core.CrashlyticsCore.sanitizeAttribute(CrashlyticsCore.java:845)
14 com.crashlytics.android.core.CrashlyticsCore.setString(CrashlyticsCore.java:560)
15 com.fitradio.ui.login.task.BaseLoginJob.handleLoginResponse(BaseLoginJob.java:153)
16 com.fitradio.ui.login.task.EmailLoginJob.getUserLoginEvent(EmailLoginJob.java:68)
17 com.fitradio.ui.login.task.BaseLoginJob.onRunRun(BaseLoginJob.java:67)

```

---

Listing 5. Method Returns Different Data Under Different Context

**3.2.4 Preprocessing.** After creating call stacks and their corresponding labels, we preprocess call stacks into format that can be processed by a natural language processing mode. (**Preprocessing** in Figure 3). 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)



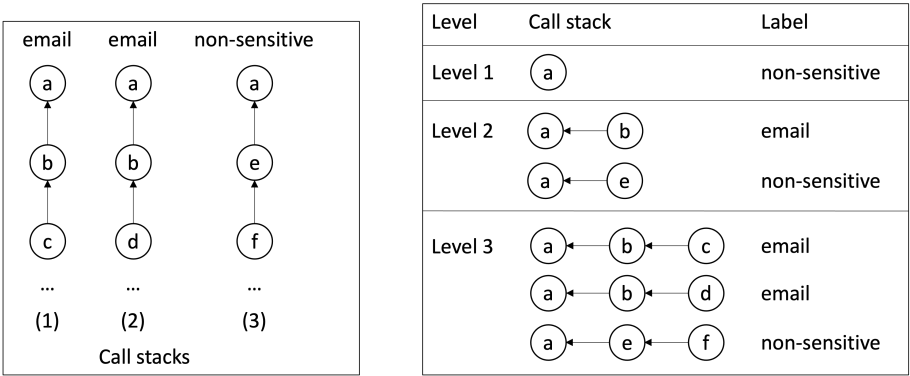


Fig. 4. Auto Labeling Strategy

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 word, which will then be fed into a natural language model for text classification learning.

### 3.3 Training of Classification Models

In this layer, the preprocessed call stacks will be used to train classifiers to predict if a sensitive information type is being accessed (**Train Classifier** in Figure 3). We use FASTTEXT to represent the word lists and train a supervised classifier for each information type.

**3.3.1 Method Signature Word Embeddings.** A calling context consists of a list of fully qualified method names denoting the order the 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 very severe problem in texts extracted from calling contexts and call stacks due to the informal language usage in source code, where 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 [27] to produce vector representations for the word sequence of a calling context. FASTTEXT is a text classification technique based on the skip-gram Word2Vec[35] model for learning vector representations for words and text classification. The FASTTEXT framework helps to overcome the challenge of unknown words by breaking all words down 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 similar to a whole word. 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 can be approximated using its constituent subwords. For example, if “biometric” was an unseen



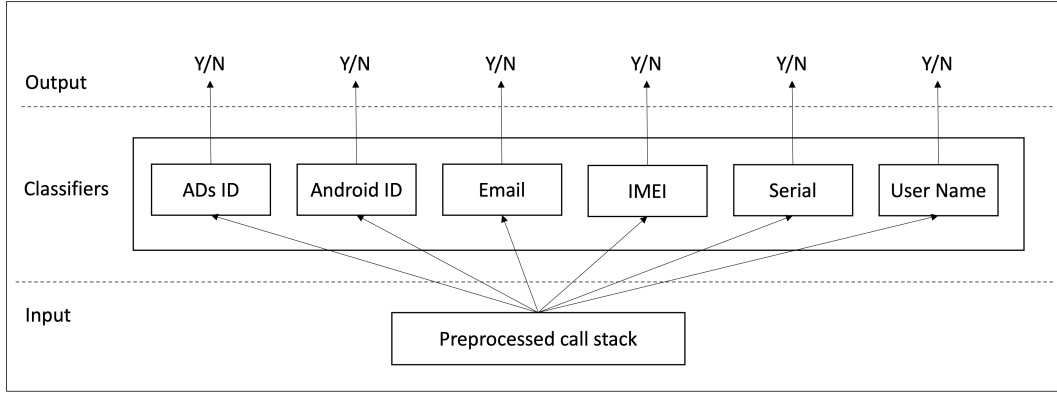


Fig. 5. Model Overview

word then it would have no existing vector representation, so 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 multiple 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”. Notably, in FastText, whole words are preprocessed by enclosing the words in the “<” and “>” characters to differentiate n-grams from n-length words, e.g., the tri-gram (i.e., subword) “met” can be differentiated from the word “<met>”.

It should be noted that although subword features are being considered, our data preprocessing (see 3.2.4) must still split full method signatures into their component word sequences so that subword extraction will be performed only for actual words and not process the entire signature as a single word.

**3.3.2 Sensitive Method Classifier.** The learning task is to classify a given method on whether its top method’s return value is of a sensitive information type. Normally, one would construct a multi-class learning model that predicts the best information type class, among a few information type classes, plus one non-sensitive class. This approach has a limitation of multi-labeling. The multi-label problem arises because a single method can return multiple sensitive information types, yet multi-class models typically predict one best class for each input. To mitigate these limitations, 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 5 shows the classifiers included in DAISY.

After FASTTEXT has completed its subword tokenization and embedding averaging (described in Section 3.3.1), the input vector representation is provided to six different binary logistic regression classifiers. The output of each classifier for a single method follows the equation below:

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

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

14 Xueling Zhang, John Heaps, Rocky Slavin, Jianwei Niu, Travis D. Breaux, and Xiaoyin Wang

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

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

where  $y_{ic} \in \{0, 1\}$  is an indicator variable for the  $i^{th}$  method's membership in class  $c$ . Minimizing the sum of equation 2 over each call stack is solved asynchronously using stochastic gradient descent [27]. Multiple threads are used to optimize performance and the number of threads is a hyperparameter of our model.

During inference, the output of equation 3.3.2 is calculated for each class. Our model predicts class membership using a threshold of 0.5 and, therefore, multiple classes can be predicted for a method. If no class probability is above 0.5 then the call trace is predicted to be "non-sensitive".

### 3.4 Data Preparation for the Prediction Phase

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 3) to extract methods with their static calling contexts. The static analysis consists of the following steps:

First, DAISY utilizes the Androguard framework[13] to disassemble an app and extract its method call graph. 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 3.2.1). 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 3**). For example, in Figure 6, (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 3.2.4, 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 6, 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.

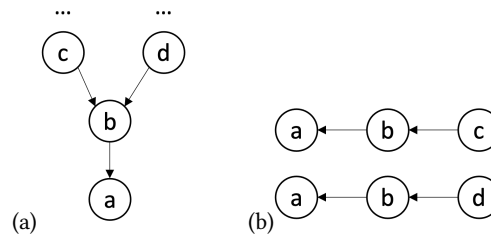


Fig. 6. Sample Call Graph and Calling Context

## 4 EVALUATION

### 4.1 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 analysis and 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?

### 4.2 Data Set

We collected our subject apps from Play-Drone[46], a collection of Android app meta data on the Google Play store. We used the 300 top-ranked apps whose APK can be de-compiled and instrumented successfully. Among the 300 apps, the top 150 were used as the training set, middle 50 apps (ranked 151-200) were used as the validation set, and the remaining 100 apps were used as our testing set<sup>2</sup>.

### 4.3 Experiment Process

We used the approach described in Section 3.4 to prepare our test set. We generate 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, leading to huge cost in the prediction process. Please note that in this paper our goal is to validate the usefulness of calling contexts, and we leave the determination of an optimal calling-context length (which may differ for different 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 set.

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 level 1, 2, and 3, respectively.

<sup>2</sup>See our dataset and code at <https://sites.google.com/view/dynacta/>

4.4 Ground Truth Labelling

The 100 apps in our testing set contain over 2.8 million in-context methods. So, it is impossible to label all of them and calculate the recall of DAISY. Furthermore, the recall metrics of DAISY do not prevent it from being practically useful. Since no technique can exhaustively detect all sensitive methods, any approach that can discover many more sensitive methods that are undetected by existing approaches or lists will bring significant additional value. Since users may need to either manually review the discovered sensitive methods, or manually review the detected information leaks from the discovered sensitive methods, the precision of DAISY can be very important, as it indirectly measures the additional effort DAISY users may need to spend when using it.

To calculate the precision of DAISY, we need to manually label the reported sensitive in-context methods to confirm true positives and 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 set) by choosing the 20 most 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). 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 ranking (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 set), 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 code are 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.

4.5 RQ1: Precision of Sensitive Method Discovery

To answer RQ1, Table 2 and Table 3 show the precision of DAISY on the high-confidence subset and the random subset respectively. In Table 2, column 1 presents the information type, while in Table 3, column 1 presents the app name. Columns 2-4 of both Table 2 and Table 3 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 2, 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 length one, two, and three, respectively. The overall precision of 77.9% indicates that

Table 2. 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 (91.7%)
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%)
Serial	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%)

Table 3. 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 majority of in-context methods identified by DAISY are 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 they are more technical with standard names, so methods returning these information types tend to have more standard name. In contrast, 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 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 3, 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 different calling-context lengths, the precision value is 61.9%, 45.7%, and 40.6%, respectively. It should be noted that sensitive in-context methods are very sparse so the data unbalance 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.

4.6 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 *sensitive method* findDeviceId() (detected by DAISY) obtained the IMEI information from Android API method android.telephony.TelephonyManager.getDeviceId(), so findDeviceId() 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 [7] using sensitive Android API method list of SUSI [38] as sources, and checks whether SUSI sources may flow to the confirmed true positives in-context methods in Table 2 and Table 3. The analysis results show that only four of the 464 discovered in-context methods are detected with static taint analysis.

```
1 // findDeviceId() is the source identified by DAISY
2
3 // android.telephony.TelephonyManager.getDeviceId() is an Android API source identified
  by SUSI
4
5 public String findDeviceId(Context context){
6     TelephonyManager tm = (TelephonyManager) context.getSystemService("phone");
7     return tm.getDeviceId();
8 }
```

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 use MONKEY to perform the testing because it is fully automatic and robust enough to be applied to all apps. Furthermore, existing studies [47] show that it achieves comparable coverage with state-of-the-art tools. For each app, we ran MONKEY for one hour without any human interaction and used the automatic labeling described in Section 3.2.3 to identify sensitive in-context methods. The comparison shows that among the 464 true positives from Table 2 and Table 3, only 23 of them are detected in 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.

4.7 Conditional Sensitive Methods

To answer RQ3, we performed a statistical analysis on the result to see how many extra sensitive in-context methods are identified when we increase 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 4 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  $a \rightarrow b$  is considered additional if and only if  $a \rightarrow b$  is reported as sensitive in Level 2, but  $a$  is not reported as sensitive in Level 1. Table 5

Table 4. Additionally Reported In-context Methods by Increasing Calling-Context Levels

InfoType	Level 1	$\Delta$ Level 2	$\Delta$ 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	<b>2237</b>	<b>4195</b>	<b>9324</b>

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

InfoType	Level 1	$\Delta$ Level 2	$\Delta$ 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	<b>103</b>	<b>46</b>	<b>23</b>

shows the total additional sensitive in-context methods that are confirmed as true positive in our evaluation subsets. With calling-context length increased to two, DAISY identified 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  $a \leftarrow b$  is considered additional if and only if  $a \leftarrow b$  is reported and manually confirmed as sensitive in Level 2, but  $a$  is not reported as sensitive in 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.

#### 4.8 Origin of DAISY sources

We further investigated the sensitive in-context methods DAISY identified to find out where the methods are 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). Such as an Email 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 login.



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.getEmail() 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. 7. DAISY Sources

4.9 Examples of Identified Sources

As discussed in Section 3.2.3, 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 7. 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 but will return sensitive information under certain calling context.

4.10 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 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, we applied static taint analysis to the DAISY sources to see if they help detect any additional leaks compared to sources from SUSI [38] and ConDySTA[57]. To do so, we ran FlowDroid with the confirmed 464 true positive DAISY sources described in TABLE 2 and TABLE 3 as sources and the sink list from SUSI as sinks. It should be



Table 6. 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	<b>30</b>	<b>11</b>	<b>1</b>

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[57] to encode the calling context as an data flow prefix in the IFDS [39] (Inter-procedural Finite 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 come from DAISY sources. The information types of the reported leaked DAISY sources include Android ID, Email, Serial number and User name. Table 6 column 1 shows the names of the apps where leaks were detected, and column 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 6 column 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, which is a significant supplement over existing techniques.

#### 4.11 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 testing 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.

## 5 DISCUSSION

### 5.1 Domain-Specific Information Types

In our research, we consider top Android apps and common sensitive 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 work on domain-specific information types, we can construct domain-specific app sets for sensitive application domains, such as education, finance, and health/medical care. Furthermore, to address

the potential sparsity problem for domain-specific information types, we plan to leverage transfer learning[54] 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.

5.2 Evaluation Strategies

As it is time-intensive to manual 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 20 top 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 very highest 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 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.

6.1 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 [7, 10, 18, 21, 28, 30, 32–34, 49, 52, 53] and dynamically [15, 43, 44, 55]. 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.

Android Platform provides large-scale application programming interfaces (APIs) to support application development. These APIs enable 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 APIs. 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 APIs that 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[21, 34] or semi-automatically based on the Android permission[32, 52]. However, it is impossible to classify all sources manually or semi-automatically given the large number of public methods in Android. [38] thus propose SUSI, a machine-learning guided approach for identifying sources and sinks from the existing Android APIs. The identified sources and sinks have enabled many studies on the detection of privacy leaks in Android apps[8, 10, 20, 30, 31, 42? ]. 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 as possible sources for data flow analysis. Different with previous works, the goal of DAISY is to identify sources beyond Android APIs, sensitive methods defined by apps or third-party libraries.

## 6.2 Non-Android API sources

In order to reduce false negatives in data flow analysis caused by missing sources, ConDySTA [57] 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. Besides, 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 [56] 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. [26, 36, 48] considered user input data as sources, which cannot be identified using the Android API methods alone and require 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 [40] and then classifying labels associated with the method invocations. DAISY instead aims to identify all sensitive methods by statically analyzing the app's program.

## 6.3 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 [25]. 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 [51]. 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 [11]. 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 [16]. 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.

6.4 Machine learning on code units

Previous studies [19, 24] have indicated 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 [38] uses machine learning to identify Android platform API methods that retrieves sensitive information. CODE2VEC [5] 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. [45] proposed a machine learning based approach to infer variable names from obfuscated code. Russell et al. [41] 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. [23] 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.

7 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 testing 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 enlarge our data set and evaluate our approach on it. Second, we plan to enhance our classification accuracy by adding more features from the code, such as 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.

## 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] 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.
- [6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Ochteau, 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.
- [7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Ochteau, 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.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] 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>
- [13] Anthony Desnos. 2015. Androguard.
- [14] Android Developers. 2012. Ui/application exerciser monkey. <https://developer.android.com/studio/test/monkey.html>
- [15] 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.
- [16] 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.
- [17] 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>
- [18] 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/scandroidasca2>, 3.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] John Heaps, Xiaoyin Wang, Travis Breau, 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.



26 Xueling Zhang, John Heaps, Rocky Slavin, Jianwei Niu, Travis D. Breaux, and Xiaoyin Wang

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

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

[38] 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.

[39] 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.

[40] 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/>

[41] 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.

[42] 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.

[43] 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.

[44] 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*.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

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

- [51] 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.
- [52] Zheming 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.
- [53] Zheming 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.
- [54] 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.
- [55] 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).
- [56] 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.
- [57] 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*.