Technical Report

Nr. TUD-CS-2013-0114
May 8th, 2013

SuSi: A Tool for the Fully Automated Classification and Categorization of Android Sources and Sinks

TECHNISCHE UNIVERSITÄT DARMSTADT

EC SPRIDE
EUROPEAN CENTER FOR SECURITY AND PRIVACY BY DESIGN

Authors
Steven Arzt (EC SPRIDE)
Siegfried Rasthofer (EC SPRIDE)
Eric Bodden (EC SPRIDE)

SuSi: A Tool for the Fully Automated Classification and Categorization of Android Sources and Sinks

Steven Arzt, Siegfried Rasthofer and Eric Bodden
Secure Software Engineering Group
European Center for Security and Privacy by Design (EC SPRIDE)
Technische Universität Darmstadt & Fraunhofer SIT
Darmstadt, Germany
{firstname.lastname}@ec-spride.de

ABSTRACT

Today's smartphone users face a security dilemma: many apps they install operate on privacy-sensitive data, although they might originate from developers whose trustworthiness is hard to judge. Researchers have proposed more and more sophisticated static and dynamic analysis tools as an aid to assess the behavior of such applications. Those tools, however, are only as good as the privacy policies they are configured with. Policies typically refer to a list of *sources* of sensitive data as well as *sinks* which might leak data to untrusted observers. Sources and sinks are a moving target: new versions of the mobile operating system regularly introduce new methods, and security tools need to be re-configured to take them into account.

In this work we show that, at least for the case of Android, the API comprises hundreds of sources and sinks. We propose SuSi, a novel and fully automated machine-learning approach for identifying sources and sinks directly from the Android source code. On our training set, SuSi achieves a recall and precision of more than 92%. To provide more fine-grained information, SuSi further categorizes the sources (e.g., unique identifier, location information, etc.) and sinks (e.g., network, file, etc.), with an average precision and recall of about 89%. We also show that many current program analysis tools can be circumvented because they use hand-picked lists of source and sinks which are largely incomplete, hence allowing many potential data leaks to go unnoticed.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Information flow controls

General Terms

Security, Experimentation

Keywords

Android, Sources and Sinks, Dataflow, Information Flow, Static Analysis, Dynamic Analysis

1. INTRODUCTION

Current smartphone operating systems, such as Android or iOS, allow users to run a multitude of applications developed by many independent developers available through various app markets. While this flexibility is very convenient for the user, as one will find a suitable application for almost every need, it also makes it hard to determine the trustworthiness of these applications.

Smartphones are widely used to store and process highly sensitive information such as text messages, private and business contacts, calendar data, and more. Furthermore, while a large variety of sensors like GPS allow a context-sensitive user experience, they also create additional privacy concerns if used for tracking or monitoring.

To address this problem, researchers have proposed various analysis tools to detect and react to data leaks, both statically [1, 2, 8, 10, 12, 15, 16, 18, 20, 23, 24, 33, 35] and dynamically [6,11,19,30,32]. Virtually all of these tools are configured with a privacy policy, usually defined in terms of lists of *sources* of sensitive data (e.g., the user's current location) and *sinks* of potential channels through which such data could leak to an adversary (e.g., a network connection). As an important consequence, the tools are only as good as the policies they are configured with. If a source is missing, a malicious app can retrieve its information without the analysis tool noticing. A similar problem exists for information written into unrecognized sinks.

This work focuses on Android. As we show, existing analysis tools, both static and dynamic, focus on a handful of hand-picked sources and sinks, and can thus be circumvented with ease. Moreover, considering that Android version 4.2, for instance, comprises about 110,000 public methods, a manual classification of sources and sinks is clearly infeasible. Furthermore, new methods are added in every new framework version, outdating classification results in regular intervals.

We therefore propose SuSi, a fully-automated machine-learning approach for identifying sources and sinks directly from the Android source code, without requiring any manual effort. We have identified both semantic and syntactic features to train a model for sources and sinks on a small subset of hand-classified Android methods. SuSi uses this model to classify arbitrarily large numbers of previously unknown Android API methods. We evaluate SuSi using a ten-fold cross validation and manually inspect some of the well-known sources and sinks. Our evaluation shows that our approach is highly precise with a recall and precision of more than 92%. Interestingly, SuSi finds *several hundred sources and* *sinks*, only a small fraction of which was previously known from the scientific literature or included in configurations of available analysis tools.

There exist some code analysis approaches, for instance LeakMiner [33], that only consider methods as sources and sinks that require a permission to execute. These methods can be identified using a permission map which can be created either statically [5, 7] or dynamically [13]. Instead, our approach does not simply rely on the methods in the permission map because this would consider only a subset of the whole Android API. SuSi identified methods that do not require any permissions, but nevertheless provide access to personal information. For instance the *getNetworkOperatorName()* method in the *TelephonyManager* class returns the name of the network operator or carrier, but does not require a permission. Applications can use this method to show targeted advertisements in an attempt to convince the user to change her provider or to offer special services in the current provider's network. Potential data leaks of such kind of private information would be missed in approaches that rely on the permission model alone.

Furthermore, only speaking of sources and sinks is too coarse-grained for practical use. If a leak is found, the user often desires information on what information has leaked where, e.g., location information to the internet. Our approach thus further classifies the identified sources and sinks into 14 source categories and 17 sink categories. The categorization shows that there is often more than one way to retrieve a certain piece of data, and that there are multiple ways to send it out to an attacker since all categories contain more than a single method.

This paper presents the following original contributions:

- a practical and precise definition of data sources and sinks in Android applications,
- a fully automated machine-learning approach for identifying data source and sink methods in the Android framework,
- a fully automated classifier for data source and sink methods into semantic categories like network, files, contact data, etc., and
- a categorized list of sources and sinks for Android version 4.2 that can be directly used by the different static and dynamic analysis approaches.

Our complete implementation and all our classification results are available as an open-source project at:

`http://sseblog.ec-spride.de/susi/`

The remainder of this paper is structured as follows. Section 2 presents a motivating example, while Section 3 gives a precise definition of the notions of sources and sinks. Section 4 presents the fully automated classifier, which we evaluate in Section 5. Other sources of sensitive information not directly related to method calls are discussed in Section 6. In Section 7, we give an overview of related work, and we conclude in Section 8.

2. MOTIVATING EXAMPLE

Lists of sources and sinks known from the scientific literature [11, 12, 15] only contain some few well-known methods

```
 1 void onCreate() {
 2 TelephonyManager tm; GsmCellLocation loc;
 3 // Get the location
 4 tm = (TelephonyManager) getContext().
 5     getSystemService
           (Context.TELEPHONY_SERVICE);
 6 loc = (GsmCellLocation)
       tm.getCellLocation();
 7
 8 //source: cell-ID
 9 int cellID = loc.getCid();
10 //source: location area code
11 int lac = location.getLac();
12 boolean berlin = (lac == 20228 && cellID
       == 62253);
13
14 String taint = "Berlin: " + berlin + " ("
       + cellID + " | " + lac + ")";
15 String f = this.getFilesDir() +
       "/mytaintedFile.txt";
16 //sink
17 FileUtils.stringToFile(f, taint);
18 //make file readable to everyone
19 Runtime.getRuntime().exec("chmod 666 "+f);
20 }
```

Listing 1: Android Location Leak Example

for obtaining and sending out potentially sensitive information in Android. However, there are often multiple ways to achieve the same effect. Developers of malicious applications can thus choose less well known sources and sinks to circumvent analysis tools. Let us assume an attacker is interested in obtaining the user's location information and writing it to a publicly accessible file on the internal storage without being noticed by existing program-analysis approaches. Listing 1 shows an example that tries to hide the data leak by using less common methods for both the source and the sink.

In our scenario, we have two source methods. Firstly, line 9 calls *getCid()*, returning the cell ID. Line 11 then calls *getLac()*, returning the location area code. Both pieces of data in combination can be used to uniquely identify the broadcast tower servicing the current GSM cell. While this is not an exact location, it nevertheless provides the approximate whereabouts of the user. In line 12 the code checks for a well-known cell-tower ID in Berlin. An actual malicious app would perform a lookup in a more comprehensive list.

Finally, the code needs to make the data available to the attacker. The example creates a publicly accessible file on the phone's internal storage, which can be accessed by arbitrary other applications without requiring any permissions. Instead of employing Java's normal file writing functions, the code uses a little-known Android system function (line 17) which SuSi identifies as a "FILE" sink but which is normally hidden from the SDK. The *FileUtils.stringToFile* function can only be used if the application is compiled against a complete platform JAR file obtained from a real phone, as the *android.jar* file supplied with the Android SDK does not contain this method. Nevertheless, the example application runs on an unmodified stock Android phone.

We have tested this example with publicly-available static [12, 15] and dynamic [11] taint analysis tools and confirmed that none of them detected the leak. This shows how important it is to generate a comprehensive list of sources and sinks for detecting malicious behavior in deceptive applications.

SuSi finds and classifies appropriately all sources and sinks used in the example.

3. DEFINITION OF SOURCES AND SINKS

Before one can infer sources and sinks, one requires a precise definition of the terms "source" and "sink". Several publications in the area of taint and information-flow analysis discuss sources and sinks, but all leave open the precise definitions of these terms. For instance, Enck et al. [11] define sinks informally as "data that leaves the system" which is, however, too imprecise for automatic reasoning.

Taint and information-flow analysis approaches track through the program the flow of *data*. Sources are where such data flows enter the program and sinks are where they leave the program again. This requires us to first define *data* in the context of data flows in Android applications.

Definition 1 (Data). *A piece of data is a value or a reference to a value.*

For instance, the IMEI in mobile applications is a piece of data, as would be the numerical value 42. We also treat as data, for instance, a database cursor pointing to a table of contact records, since it directly points to a value and is thus equivalent in terms of access control.

In taint tracking, one monitors the flow of data between resources such as the file system or network. Conversely, due to Android's app isolation, data that is simply stored in the app's address space is not of interest. Before one can define sources and sinks, one must therefore define the notion of a resource method. Mobile operating systems like Android enable applications to access resources using predefined methods. While one could also imagine fields being used for resource access, we found this not to be the case with Android.

Definition 2 (Resource Method). *A resource method reads data from or writes data to a shared resource.*

For instance, the operating system method for reading the IMEI (*getDeviceId()* in class *TelephonyManager*) is a resource method. In this case, the phone's hardware itself is the resource as the IMEI is branded into the silicon. The *sendTextMessage()* method in class *SmsManager* is a resource method for sending text messages to a specific phone number. The resource is the GSM network.

Note that a writing resource method does not necessarily need a reading counterpart. In our definition, there is no restriction on how the data is shared. A writing resource method might, for instance, send out data over the network (which is a resource). Though another application cannot directly obtain this data through a simple method call, the data can easily be sniffed from the network and is thus shared. Data leaving the phone is thus always considered shared.

After defining *data* and *resource methods* we can now define sources and sinks in the context of Android applications:

Definition 3 (Android Source). *Sources are calls into resource methods returning non-constant values into the application code.*

The *getDeviceId()* resource method is an Android source. It returns a value (the IMEI) into the application code. The IMEI is considered non-constant as the method returns a different value on every phone. Looking at the source code alone does not reveal this value. In contrast, a function that just reads a fixed constant from a database is a resource method but, by our definition, is not an Android source.

Definition 4 (Android Sinks). *Sinks are calls into resource methods accepting at least one non-constant data value from the application code as parameter, if and only if a new value is written or an existing one is overwritten on the resource.*

The *sendTextMessage()* resource method is an Android sink as both the message text and the phone number it receives are possibly non-constant. On the other hand, the *reboot* method in the *PowerManager* class, for instance, just receives a kernel code for entering special boot modes which must be part of a predefined set of supported flags. This method is thus only a resource method (the data is written into the kernel log), but not an Android Sink. We require this restriction on constant values for methods which do not introduce any new information into the calling application in the case of sources, or do not directly leak any data across the application boundary in the case of sinks. The values at calls to such methods are of a purely technical kind (e.g., system constants, network pings etc.) and not of interest to typical analysis tools. Note that our definition also excludes some implicit information flows. This is a design choice. For instance, in our approach the vibration state of the phone is not considered a single-bit resource, even though it could theoretically be observed and would then be "shared".

A malicious app can try to access private information not only through calls to the official Android framework API but also through calls to code of pre-installed apps. For instance, the default email application provides a readily-available wrapper around the *getDeviceId()* function. This app is pre-installed on every stock Android phone, which gives a malicious app easy access to the wrapper: the app just instructs the Android class loader to load the respective system APK file and then instantiates the desired class. To cover such cases, our approach does not only analyze the framework API but the pre-installed apps as well. (We use a Samsung Galaxy Nexus with Android 4.2.)

4. FULLY AUTOMATED CLASSIFICATION

In this section, we explain the details of SuSi, our machine-learning approach to fully-automatically identify sources and sinks corresponding to the definitions given in Section 3. We address two classification problems. For a given unclassified Android method, SuSi first decides whether it is a *source*, a *sink*, or *neither*. The second classification problem refines the classification of sources and sinks identified in the first step. All methods previously classified as *neither* are ignored. For an uncategorized source or sink, SuSi determines the most likely semantic category it belongs to. In our design, every method is assigned to exactly one category.

Section 4.1 gives a short introduction to machine learning. Section 4.2 then presents the general architecture of SuSi, while Section 4.3 discusses the features SuSi uses to solve its classification problems. Section 4.4 gives more details on one particularly important family of features which deals with data flows inside the methods to be classified. In Section 4.5 we show how the semantics of the Java programming language can be exploited to artificially generate further annotated

ID	Experience	Alcohol	Phone No	Accident
T1	5 yrs	0.6	1234	yes
T2	11 yrs	0.4	45646	yes
T3	7 yrs	0.2	76546	yes
T4	4 yrs	0.0	54645	no
T5	10 yrs	0.2	78354	no
C1	6 yrs	0.1	6585	?
C2	12 yrs	0.55	67856	?

Table 1: Classification Example on Drunk Driving

training data. Section 4.6 discusses some prefiltering that SuSi applies.

4.1 Machine Learning Primer

SuSi uses *supervised learning* to train a classifier on a relatively small subset of manually-annotated training examples. This classifier is afterwards used to predict the class of an arbitrary number of previously unseen test examples. Classification is performed using a set of features. A feature is a function that associates a training or test example with a value, i.e., evaluates a certain single domain-specific criterion for the example. The approach assumes that for every class there is a significant correlation between the examples in the class and the values taken by the feature functions.

As a simple example, consider the problem of estimating the risk of a driving accident for an insurance company. We may identify three features: years of experience, blood alcohol level and the driver's phone number. Assume the learning algorithm deduces that a higher level of experience is negatively correlated with the accident rate, while the alcohol level is positively correlated and the phone number is completely unrelated. The impact of a single feature on the overall estimate is deduced from its value distribution over the annotated training set. If there are many examples with high-alcohol accidents, then this feature will be given a greater weighting than the years of experience. However, if there are more accidents of inexperienced drivers in the training set than alcohol-related issues, the classifier will rank the experience feature higher.

The classifier works on a matrix, organized with one column per feature and one row per instance. Table 1 shows some sample data. An additional column indicates the class and is only filled in for the training data. In our example, this column would indicate whether or not an accident took place. The first five rows are training data, the last two rows are test records to be classified.

In this example, a simple rule-based classifier would deduce that all reports with alcohol levels larger than 0.2 also contained accidents, so C2 would be classified as *accident: yes*. However, since the converse does not hold, further reasoning is required for C1. Taking the experience level into account, there are two records of inexperienced drivers with levels of 0.2 or below in our test set: one with an accident and one without. In this case, the classifier would actually pick randomly, since both *accident:yes* and *accident:no* are equally likely. A probabilistic classifier could also choose *accident:yes* because accidents are more likely for inexperienced drivers (two out of three with five years of experience or less in this test data set) in general. This demonstrates that results can differ depending on the choice of the classifier.

As a concrete classifier, we use *support vector machines*

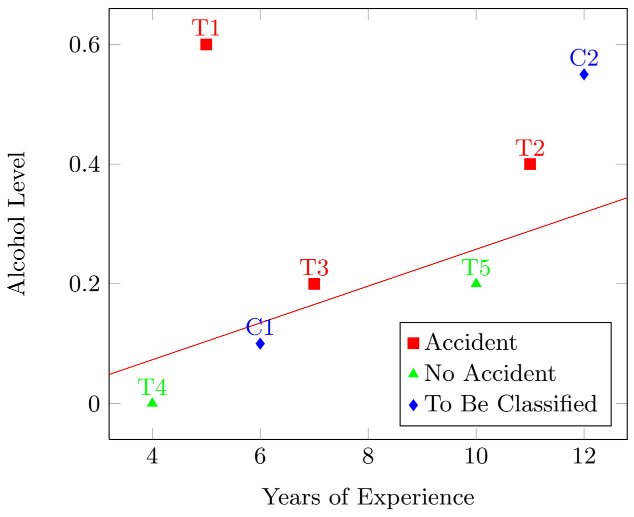

Figure 1: SMO Classification Example

(SVM), a margin classifier, more precisely the *SMO* [26] implementation in Weka [17] with a linear kernel. We optimize for minimal error. The basic principle of an SVM is to represent training examples of two classes (e.g., "sink" and "not a sink") using vectors in a vector space. The algorithm then tries to find a hyper-plane separating the examples. For a new, previously unseen test example, to determine its estimated class, it checks on which side of the hyper-plane it belongs. In general, problems can be transformed into higher-dimensional spaces if the data is not linearly separable, but this did not prove necessary for any one of our classification problems.

Figure 1 shows an SMO diagram for Table 1. We have not included the phone number feature since it is unrelated to the probability of an accident. The red line shows a projection of the hyper-plane. In this example, the SMO detects that all points above the line are positive examples (i.e., records of accidents), and all points below are negative ones (i.e., no accident). C2 would thus be classified as an accident, just as with the simple rule-based classifier above, but C1 would now definitely be classified as non-accident because it lies below the line.

SMO is only capable of separating two classes. However, we have three classes in the first problem and a lot more in the second one (the categorization). We solve the problem with a one-against-all classification, a standard technique in which every possible class is tested against all other classes packed together to find out whether the instance corresponds to the current single class or whether the classification must proceed recursively to decide between the remaining classes.

We also evaluated other classification algorithms based on different principles, for instance Weka's J48 rule learner, which implements a pruned C4.5 decision tree [27]. The main problem with a rule set is its lack of flexibility. While many source-method names, for instance, start with *get*, this is not the case for *all* source methods. On the other hand, not all methods that start with *get* are actually sources. Since this rule of thumb is correct most of the time, however, a rule tree would usually include a rule mapping all *get* methods to sources and only perform further checks if the method name has a different prefix. With an SVM, such aspects that are usually correct, but not always, can be expressed more

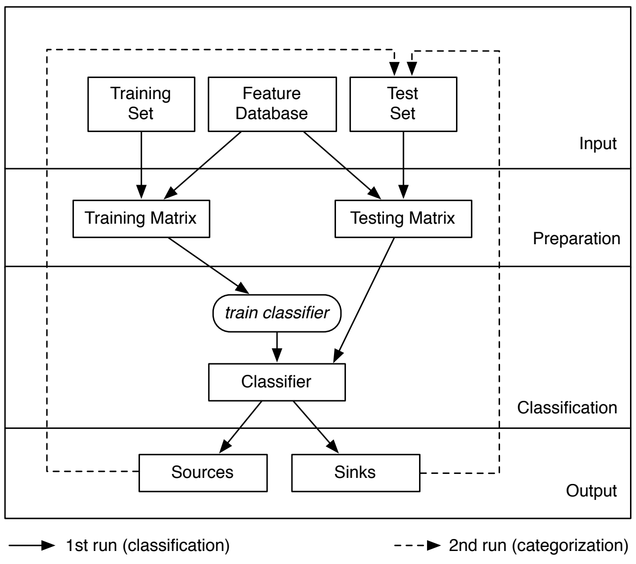

Figure 2: Machine Learning Approach

appropriately by shifting the hyper-plane used for separation.

Probabilistic learning algorithms like Naive Bayes [34] produced very imprecise results. This happens because our classification problem is *almost* rule-based, i.e., has an almost fixed semantics. The variance is simply not large enough to justify the imprecision introduced by probabilistic approaches which are rather susceptible to outliers.

4.2 Design of the Approach

Figure 2 shows SuSi's overall architecture. It includes four different layers: *input*, *preparation*, *classification*, and *output*. The square elements denote objects, while the round elements represent actions. We run two rounds: One for classifying methods as *sources*, *sinks*, or *neither*, and one for categorizing them. Solid lines denote the data flow within SuSi. The two dashed lines denote the initialization of the second round. The general process is the same for both rounds. For the categorization, SuSi just takes the outputs of the classification as test data inputs. More precisely, SuSi categorizes separately those methods it has previously identified as sources or sinks and disregards those it classified as neither.

SuSi starts with the input data for the first classification problem, i.e., for identifying sources and sinks. This data consists of the Android API methods to analyze. These methods can be separated into a set of training data (hand-annotated training examples) and a set of test data for which we do not know whether a method is a source, sink or neither. The set of training data is much smaller than the set of unknown test data, in our case only roughly 0.7% for the classification and about 0,4% for the categorization. Beside the API methods we need a database of features, both for the classification and categorization. The features are different for classification and categorization. See Section 4.3 for details.

As described in in Section 4.1, a supervised learning approach requires two matrices. The first one is built by evaluating the features on the set of hand-annotated training data, the second one by applying the same feature set as well to the test data yet to be classified (*preparation* step). SuSi then uses the first matrix to train the classifier (*classification* step), which afterwards decides on the records in the test matrix (*output* step).

While there are a few methods in the Android library that are both sources and sinks (such as some of the *transceive* methods of the NFC implementation), their scarcity stops us from establishing a fourth category "*both*", even though in theory such a category might sound sensible. Respectively, we treat such methods as either sources or sinks. This decision affects both the training data and the classifier's results.

In a second step, SuSi *categorizes* the sources and sinks set. In this step, SuSi separately considers the sources and sinks determined in the first step as new test sets (dashed arrows). Note that methods classified as *neither* are ignored at this point. SuSi also requires new training data for the second classification problem. To provide such data, we hand-annotated a subset of the Android sources and sinks with semantic categories related to the mobile domain. We furthermore chose different kinds of features for the feature database as explained in Section 4.3. We chose 14 different kinds of source-categories that we identified as being sufficiently meaningful for the different Android API methods: *account*, *bluetooth*, *browser*, *calendar*, *contact*, *database*, *file*, *image*, *location*, *network*, *nfc*, *settings*, *sync*, and *unique-identifier*. For the sinks, we defined 17 different kinds of categories: *account*, *audio*, *browser*, *calendar*, *contact*, *email*, *file*, *location*, *log*, *network*, *nfc*, *phone-connection*, *phone-state*, *sms/mms*, *sync*, *system*, and *voip*. For the purpose of compiling our training data, if a method is not relevant or does not fit in any of the identified categories, it is annotated as belonging to the special *no-category* class. If one wants to add a new category, one simply has to create new features for the feature database and randomly annotate the corresponding API methods. Our approach automatically uses the new feature for the generation of the categorized sources and/or sinks. The subsequent steps as shown in Figure 2 are equal to the one for the classification. The final output consists of two files, one for the categorized sources and one for the categorized sinks.

Note that some of these categories refer to data being managed by applications, not the operating system itself. One example are contacts: The system provides a data interface to make sure that there is a uniform way of obtaining contacts for all applications that require them, e.g., travel planners, or calendars sending invitations. Additionally, Android contains system applications providing default implementations of these interfaces, so there are methods which are available on every Android phone and which can be called in order to obtain private data. Therefore, we include categories for such methods, despite them not being part of the operating system as such.

Since we have different categories for sources and sinks, their categorization comprises two distinct classification problems: one for sources and one for sinks. Though they share the same feature set (see Section 4.3), both are solved independently of each other. Thus, quite naturally, the resulting correlations might differ significantly, as some features might be more relevant to distinguish different kinds of sources than different kinds of sinks, and vice versa.

4.3 Feature Database

We used a set of 144 syntactic and semantic features for classifying the Android methods. A single feature alone does not usually give enough information to decide whether a given Android method is a source, a sink or neither. However, all features in combination can be used to train a highly precise classifier. The same holds for the second classification problem in which we need to find categories for our sources and sinks.

One main reason for why these features work is that many developers of the Android framework do in fact follow a certain regular coding style, or duplicate parts of one method's implementation when implementing another. These social aspects of software development lead to a certain degree of regularity and redundancy in the code base, which a machine-learning approach such as ours can discover and take advantage of.

Though we have a large number of distinct features, most of them are instances of the same parameterized class. For example, the "method name starts with" feature class has instances "method name starts with *get*", "method name starts with *put*", and so on. For identifying sources and sinks, SuSi uses the following classes of features:

Method Name The method name contains or starts with a specific string, e.g., "get", which can be an indicator for a source.

Method has Parameters The method has at least one parameter. Sinks usually have parameters, while sources might not.

Return Value Type The method's return value is of a specific type. A returned cursor, for instance, hints at a source, while a method with a void return value is rarely ever a source.

Parameter Type The method receives a parameter of a specific type. This can either be a concrete type or all types from a specific package. For instance, a parameter of type *java.io.** hints at a source or a sink.

Parameter Is An Interface The method receives a parameter of an interface type. This is often the case with methods that register callbacks. Note that such methods are neither sources nor sinks according to our definition, since they do not perform any actual operation on the data itself.

Method Modifiers The method is static/native/etc. Static methods are usually neither sources nor sinks, with some exceptions. Additionally, sources and sinks are usually public.

Class Modifiers The method is declared in a protected/abstract etc. class. Methods in protected classes are usually neither sources nor sinks.

Class Name The method is declared in a class whose name contains a specific string, e.g., *Manager*.

Dataflow to Return The method invokes another method starting with a specific string (e.g. *read* in the case of a source). The result of this call flows into the original method's return value. This hints at a source.

Dataflow to Sink One of the method's parameter flows into a call to some other method starting with a specific string, e.g., *update*, which would suggest a sink.

Data Flow to Abstract Sink One of the method's parameter flows into a call to an abstract method. This is a hint for sink as many command interfaces on the hardware abstraction layers are built on top of abstract classes.

Required Permission Invoking the method requires a specific permission. There is one such feature for every permission declared in the Android API. We were only able to use this feature on the approximately 12,600 methods for which we had permission annotations from the PScout [5] list.

All our features can assume one of three values: "True" means that the feature applies, i.e., a method does indeed start with a specific string. "False" means that the feature does not apply, i.e., the method name does not have the respective prefix. "Not Supported" means that the feature cannot be decided for this specific method. The latter can happen if, for example, the feature needs to inspect the method body, but no implementation is available in the current Android version's platform JAR file.

The details of our dataflow features are explained in Section 4.4. SuSi's features for *categorizing* sources and sinks can be grouped as follows:

Class Name The method is declared in a class whose name contains a specific string, e.g., *Contacts*.

Method Invocation The method directly invokes another method whose fully-qualified name starts with a specific string, e.g., *com.android.internal.telephony* for Android's internal phone classes. This feature does not consider the transitive closure of calls starting at the current method.

Body Contents The method body contains a reference to an object of a specific type, e.g. *android.telephony .SmsManager* for the *SMS_MMS* category).

Parameter Type The method receives a parameter of a specific type (similar feature as for the classification problem with different instances).

Return Value Type The method's return value is of a specific type, e.g., *android.location.Country* for regional data.

Note that we do not use permission-based features for the categorization, since many methods require permissions for internal functionality not directly related to their respective category. For instance, a backup method requests many permissions, but does not necessarily give out all of the data it accesses using these permissions if it only creates an internal savepoint that can be restored later. The permission list alone thus does not directly relate to the method's category.

It becomes apparent that semantic features are much more suitable for identifying sources and sinks than for categorizing them. On the source-code level, Android's sources and sinks share common patterns which can be exploited by our dataflow feature. For finding categories, however, there seems no such technical distinction and SuSi must rather rely on syntactical features such as class and method names.

4.4 Dataflow Features

As we found through empirical evaluation, considering a method's signature and the syntax of its method body alone is insufficient to reliably detect sources and sinks. With such features alone we were unable to obtain a precision or recall higher than about 60%. It greatly helps to take the data flows inside the method into consideration as well. Recall from our definitions in Section 3 that sources must read from and sinks must write to resources.

To analyze data flows, we originally experimented with a highly precise (context-, flow- and object-sensitive) data-flow analysis based on Soot [21], but found out that this did not easily scale to the approximately 110,000 methods of the Android SDK. Computing precise call graphs and alias information simply took too long to be practical. We thus changed to a much more coarse-grained intra-procedural approximation (also based on Soot) which runs much faster whilst remaining sufficiently precise for the requirements of our classification. Keep in mind that the result of the data-flow analysis is only used as one feature out of many. Thus, it suffices if the analysis is somewhat precise, i.e., produces correct results with just a high likelihood.

Our data-flow features are all based on taint tracking inside the Android API method to be classified. Depending on the concrete feature, we support the following analysis modes:

- Treat all parameters of the method as sources and calls to methods starting with a specific string as sinks.
- Treat all parameters of the method as sources and calls to abstract methods as sinks.
- Treat calls to specific methods as sources and the return value as the only sink. Optionally, parameter objects can also be treated as sinks.

Based on this initialization, we then run a fixed-point iteration with the following rules:

- If the right-hand side of an assignment is tainted, the left-hand side is also tainted.
- If at least one parameter of a well-known *transformer* method is tainted, its result value is tainted as well.
- If at least one parameter of a well-known *writer* method is tainted, the object on which it is invoked is tainted as well.
- If a method is invoked on a tainted object, its return value is tainted as well.
- If a tainted value is written into a field, the whole base object becomes tainted. For arrays, the whole array becomes tainted respectively.

When the first source-to-sink connection is found, the fixed point iteration is aborted and the dataflow feature returns "True" for the respective method to which it was applied. If the dataflow analysis completes without finding any source-to-sink connections, the feature returns "False".

While such an analysis would be too imprecise for a general-purpose taint analysis, it is very fast and usually reaches its fixed point in less than three iterations over the method body. Since the analysis is intra-procedural, its runtime is roughly bounded by the number of statements in the respective method.

4.5 Implicit Annotations for Virtual Dispatch

SuSi's implementation is based on Weka, a generic machine learning tool, which has no knowledge about the language semantics of Java. However, we found that when annotating methods to obtain training data it would be beneficial to propagate method annotations up and down the class hierarchy in cases in which methods are inherited. Such a propagation models the semantics of virtual dispatch in Java. We thus extended SuSi such that if encountering an annotated method *A.foo*, the annotation is implicitly carried over also to *B.foo* in case *B* is a subclass of *A* that does not override *foo* itself, thus inheriting the definition in *A*. Similarly, if *B.foo* were annotated, but not *A.foo*, we would copy the annotation in the other direction.

For our subset of 12,600 methods with permission annotations taken from the PScout list [5], SuSi was able to automatically create implicit annotations for 305 methods. After loading the remaining methods of the Android API to get our full list of 110,000 methods, SuSi was able to automatically annotate another 14 methods.

4.6 Prefiltering

Our definition of sources and sinks is based on how a method *behaves*. Abstract methods have no own behavior, which is why we remove such methods from consideration by SuSi. The same holds for methods whose definition for technical reasons is simply not available in the Android platform version at hand.

We also prune all private methods and all methods in private classes. This design choice is justified by the fact that apps can access such methods only through reflection. Given that we created SuSi primarily to aid static Android analysis tools, and given that none of these tools can currently handle Java reflection, including such private sources and sinks could not possibly aid those tools but might instead pollute our machine-learning approach with unimportant data. With this technique, our subset of 12,600 methods with permission information is pruned down to 6,700.

5. EVALUATION

Our evaluation considers the following research questions:

RQ1 Can SuSi effectively find sources and sinks with high accuracy?

RQ2 Can SuSi categorize the found sources and sinks with high accuracy?

RQ3 How complete are the lists of sources of sinks distributed with existing Android analysis tools and how do they relate to SuSi's outputs?

The following sections address these questions in order.

5.1 RQ1: Sources and Sinks

To evaluate SuSi, we use ten-fold cross validation: divide all training data into 10 equally-sized buckets, train the classifier on nine of them, and then classify the remaining bucket, repeating the process 10 times, omitting another bucket from training each time. In the end, SuSi reports the average precision and recall. For each class *c*, precision is the fraction of correctly classified elements in *c* within all elements that *were* assigned to *c*. If precision is low it means that *c* was assigned many incorrect elements. Recall is defined as fraction of correctly classified elements in *c* within all elements that *should have been* assigned to *c*. If recall is low it means that *c* misses many elements.

Table 2 shows the results of this ten-fold cross validation over our training set of 779 methods randomly picked from the PScout subset [5] of about 12,600 methods. We started with this subset as it provided mappings between methods and required permissions and thus enabled us to also use Android permissions as features for our classifier. The average we report in the table is weighted with the number of examples in the respective class.

Category	**Recall** [%]	**Precision** [%]
Sources	92.3	89.7
Sinks	82.2	87.2
Neither	94.8	93.7
Weighted Average	91.9	91.9

Table 2: Source/Sink Cross Validation PScout

Our final results for the source/sink classification had to be computed without any permission features, though, since we do not have permission associations for the complete Android API[1]. For assessing the impact of the permission feature, we ran the PScout subset again with the permission feature disabled, yielding the results shown in Table 3. Interestingly, the average precision and recall are almost the same with the permission feature and without. The impact of the permission feature is apparently low enough for not having to worry about the lack of permission information when analyzing the complete Android API.

Category	**Recall** [%]	**Precision** [%]
Sources	90.5	91.3
Sinks	86.0	88.8
Neither	95.2	94.4
Weighted Average	92.8	92.8

Table 3: Source/Sink Cross Validation PScout Without Permission Feature

Table 4 shows the ten-fold cross-validation results of applying our approach to the complete Android SDK of about 110,000 public methods. Note that we did not extend our set of manually annotated training records for this test. However, since SuSi automatically propagates classifications along the class hierarchy as explained in Section 4.5, one obtains slightly more *implicitly annotated* data and thus different results. SuSi again shows an average recall and precision of more than 92%.

Category	**Recall** [%]	**Precision** [%]
Sources	89.6	88.0
Sinks	84.7	90.8
Neither	95.2	93.6
Weighted Average	92.3	92.3

Table 4: Source/Sink Cross Validation Complete List

The classifier takes about 26 minutes to classify the complete Android API on a MacBook Pro computer running MacOS X version 10.7.4 on a 2.5 GHz Intel Core i5 processor and 8 GB of memory.

As explained in Section 4.1, we experimented with various classification algorithms, and found that SMO performed best. In Table 5, we compare the weighted average precision for SMO, J48, and Naive Bayes, the most well-known representatives of their respective families of classifiers (margin, rule-based and stochastic classifier, respectively). The results were computed on the complete 110,000 public methods of the Android 4.2 API without the permission feature.

[1]The available permission lists including PScout are incomplete since they exclude permissions enforced in native code.

Category	**Recall** [%]	**Precision** [%]
ACCOUNT	100.0	100.0
BLUETOOTH	83.3	100.0
BROWSER	80.0	100.0
CALENDAR	100.0	100.0
CONTACT	95.0	100.0
DATABASE	50.0	100.0
FILE	75.0	100.0
IMAGE	75.0	100.0
LOCATION	100.0	100.0
NETWORK	83.3	83.3
NFC	100.0	100.0
SETTINGS	75.0	85.7
SYNC	100.0	100.0
UNIQUE_IDENTIFIER	88.9	100.0
NO_CATEGORY	91.7	59.5
Weighted Average	88.8	89.7

Table 6: Source Category Cross Validation

Category	**Recall** [%]	**Precision** [%]
ACCOUNT	85.7	100.0
AUDIO	100.0	100.0
BROWSER	50.0	100.0
CALENDAR	100.0	100.0
CONTACT	91.7	100.0
EMAIL	100.0	100.0
FILE	60.0	100.0
LOCATION	100.0	100.0
LOG	100.0	71.4
NETWORK	72.7	88.9
NFC	100.0	100.0
PHONE_CONNECTION	75.0	85.7
PHONE_STATE	100.0	100.0
SMS_MMS	96.3	100.0
SYNC	80.0	100.0
SYSTEM	80.6	89.3
VOIP	66.7	100.0
NO_CATEGORY	97.1	70.2
Weighted Average	88.4	90.4

Table 7: Sink Category Cross Validation

5.2 RQ2: Categories for Sources and Sinks

We also use ten-fold cross validation on our training data to assess the quality of our categorization. For this task, we do not use the permission feature and thus only report results for the complete list of approximately 110,000 public methods in the Android API. Table 6 shows the cross-validation results for categorizing the sources, while Table 7 contains those for the sinks.

While SuSi achieves a very high precision and recall for most of the categories, the results for a few categories (e.g. Bluetooth) are considerably worse. These categories are rather small, i.e. randomly picking training methods from the overall set of 110,000 Android API methods yields only

Classifier	Avg. Recall			Avg. Precision		
	Class. [%]	Source Cat. [%]	Sink Cat. [%]	Class. [%]	Source Cat. [%]	Sink Cat. [%]
Margin (SMO)	92.3	88.8	88.4	92.3	89.7	90.4
Rule-Based (J48)	89.5	81.0	80.2	89.4	81.6	77.4
Probabilistic (Naive Bayes)	86.9	61.5	46.6	87.1	61.7	36.1

Table 5: Source/Sink Classifier Comparison

few entries belonging to such categories. Respectively, there is not much material to train the classifier on. Annotating more data (recall that we only have category annotations for 0.4% of all methods) would certainly improve the situation.

Categories can be ambiguous in some cases. A method to set the MSIDN (the phone number to be sent out when placing a call) could for instance be seen as a system setting (category SETTINGS), but could also be considered a UNIQUE_ID. In such cases, we checked the classifier's result and updated our training data if a misclassification was to due semantic ambiguity, i.e., the result would be right in both categories. Categories that ended up empty or almost empty due to such shifts were removed.

Categorizing the sources took about 6 minutes on our test computer. The sinks were classified in about 3 minutes.

5.3 RQ3: Existing Lists of Sources & Sinks

In this section we assess to what extent current static [1, 2, 12, 15, 16, 20, 23, 24, 33] and dynamic code analysis [11, 32] approaches could benefit from our categorized sources/sinks list. As our results show, SuSi finds all the sources and sinks these previous approaches mention, plus many others which the community was previously unaware of.

Unfortunately, most of the code-analysis tools were not publicly available to us, precluding us from directly comparing their source and sink lists to ours [16, 20, 23, 24, 32, 33]. As a best effort, we thus estimated the lists from the respective research papers.

Mann et al. [24] mention a few concrete source and sink methods. This hand-picked list is only a fraction of the one produced by SuSi. The taint-tracking tool CHEX [23] uses a list of 180 semi-automatically collected sources and sinks. Unfortunately, this list is not publicly available and the paper does not explain how the semi-automatic approach works. The authors do mention that their list is based on the Android permission map by Porter Felt et al. [14] but also argue that this list is insufficient. LeakMiner [33] uses the Android permission map to identify sources and sinks. From this map it filters out all methods an application is not allowed to use. However, this leaves open how the tool actually identifies the *relevant* sources and sinks in the remaining method set. Furthermore, if all methods not requiring a permission are filtered, some sensitive data might be overlooked as we have shown. Scandal [20] and AndroidLeaks [16] do not provide concrete lists of source and sink methods. The publications only provide categories (e.g., location information, phone identifier, internet, etc.), which are also covered by our automatic categorization. SCanDroid [15] is available as an open-source tool [3]. We extracted the source and sink specifications from the source code (version of April 2013). The resulting list appears hand-picked and only contains a small fraction of SuSi's. Enck et al. [12] implemented a tool that decompiles the Android bytecode into Java bytecode. This bytecode is then used as input for the commercial Fortify SCA [1] static code-analysis suite. Fortify can be configured with rules for defining sources and sinks. Enck et al. created such rules and made them publicly available [4]. The list contains about 100 Android sources and 35 Android sinks, all of which are also included in our results.

Aurasium [32] shifts the problem of identifying sources and sinks by intercepting calls at the system level, i.e., between the native Android libraries and the standard Linux system libraries. While this reduces the number of methods to consider, it makes it harder to reconstruct higher-level semantics, and is failure-prone in case of Android version upgrades. Due to this design, the sources and sinks considered by Aurasium are incomparable to SuSi's results. TaintDroid [11] is a well-known dynamic taint-analysis tool for Android applications. We tried to extract the list of sources and sinks from TaintDroid's source code which is, however, non-trivial. TaintDroid does not specify the high-level API calls as sources or sinks, and instead uses the smaller set of lower-level internal system methods called by those, an approach somewhat comparable to Aurasium. However, this again raises the problem of reconstructing the higher-level context from lower-level calls. The type of data leaked can thus be imprecise. Furthermore, we also found that TaintDroid over-approximates the list of sources and sinks, for instance by tainting the result value of all methods in the *TelephonyManager* class, including the result of *toString()*, which is just the Java object ID (default implementation inherited from *java.lang.Object*). We thus argue that automatically inferring higher-level API methods as provided by our approach would improve tools like TaintDroid as this would allow one to more easily categorize and differentiate various types of sources and sinks.

We also examined well-known commercial tools for static code analysis such as Fortify SCA [1] by HP and IBM AppScan Source [2]. As we found, by default these tools provide lists that are rather incomplete[2]. However, both provide an easy way to integrate new sources and sinks to be considered by the analysis. This shows that these tools shift the problem of defining sources and sinks to the analyst, who still needs to obtain such a list from somewhere. SuSi can help to provide more comprehensive defaults.

6. SOURCES NOT CONSIDERED BY SUSI

SuSi works well when it comes to classifying sources and sinks based on their structural similarity to other sources, respectively sinks. In practice, this seems to work well for sources that return data from method calls and sinks that obtain data through parameters. Android offers other less prevalent sources and sinks, however, which cannot be easily classified through machine learning which we will show in this section.

[2]AppScan Source does not allow to display the default source and sink lists. This has been verified with IBM Customer Support, so we empirically evaluated the list.

```
NmeaListener mylistener = new
    NmeaListener() {
  public void onNmeaReceived(long arg0,
      String nmea) {
    if (nmea.startsWith("$GPGLL")) {
      String[] data = nmea.split(",");
      Log.d("Loc", "Longitude: "
        + data[3]} + data[4]
        + ", Latitude: " + data[1] +
            data[2]);
    }
  }
};
LocationManager lm = (LocationManager)
    this.getSystemService(LOCATION_SERVICE);
lm.addNmeaListener(mylistener);
// Just to start GPS, no data from this
    callback is ever used
lm.requestLocationUpdates
    (LocationManager.GPS_PROVIDER, 0, 0,
    new LocationListener() { ... });
```

Listing 2: Android Location via NMEA Data

Applications can implement callback methods and receive data from the operating system through the parameters of these methods. This is commonly used to e.g., obtain the location in an Android application. In an attempt to avoid detection, the app could however register the callback with *onNmeaReceived* instead of the well-known *onLocationChanged* method and then parse the raw GPS data (the NMEA records) as shown in Listing 2. Both kinds of callbacks receive sensitive data as parameters. This shows, that a complete list of callback methods is required for finding all data leaks. SuSi cannot currently find such callbacks as, by our definition of sources, such callbacks are out of scope. Luckily, the number of callback interfaces in the Android operating system is sufficiently small for manual inspection, allowing those methods to indeed be added manually. Static analysis could also aid their detection.

Android defines layout controls through XML files. In the source code, they can be accessed by passing the respective identifier to the system's *findViewById* function. Depending on the ID that is passed, this function can return, for instance, a reference to a password field or to a button with a constant label. Thus, depending on the ID, the method can or can not be a source. Since calls to this function are present in almost every Android app, a precise analysis must model the Android resource system.

7. RELATED WORK

Our work was originally inspired by Livshit's orthogonal research on automatic placement of sanitizers [22]. Livshits' approach assumes sources, sinks and sanitizers to be known in advance, and then attempts to place them both effectively and efficiently with the goal of a low runtime overhead. Obtaining these lists triggered our interest in building SuSi.

Privacy violations through leaks of sensitive data in Android applications are well known in the community. To protect the user's privacy, different kinds of taint-tracking approaches have been proposed, both static [1, 2, 8, 10, 12, 15, 16, 18, 20, 23, 24, 33, 35] and dynamic [11, 19, 30, 32]. As already described in Section 1, such approaches are only as good as the source and sink lists they are configured with. In Section 5.3 we have shown that all approaches we have evaluated only consider a few sensitive methods for sources and sinks. With the support of our categorized list of sources and sinks, we argue that all of them could be improved to detect more data leaks that are a security problem for the mobile device user.

More generic policy enforcement approaches, for instance Kynoid [30] or AppGuard [6], also require comprehensive lists of sensitive information sources. AppGuard, for instance, provides the user with the ability to revoke permissions after app-installation time. The implementation inserts additional permission checks into the application (not the framework). This requires the identification of relevant methods at the API level for which such checks are required. Our list of sources and sinks includes many methods that require permissions and access sensitive information (e.g., phone identifier, location information, etc.) but are not considered by AppGuard (evaluated version 1.0.3).

Applying machine learning for security has already been done for automatic spam detection [29] or anomaly detection in network traffic [31]). Sarma et al. [28] and Peng et al. [25] successfully used various machine-learning approaches to detect malicious Android applications. MAST [9] is a machine-learning approach based on Multiple Correspondence Analysis (MCA) for automatically identifying malicious applications from various Android markets. The tool aims at ranking apps for inspection by a human security analyst, thereby giving priority to those applications that look suspicious. For classifying sources and sinks, we use SMO instead of MCA since MCA requires a logical ordering of records which is not applicable to our scenario. SuSi instead works on discrete and independent classes.

8. CONCLUSIONS

In this paper, we have shown that privacy-enhancing technologies for Android are threatened by the fact that they come with largely incomplete lists of sources and sinks of private information, thereby allowing attackers to circumvent their measures with ease. We have presented SuSi, a novel and fully automated machine-learning approach for identifying sources and sinks in the Android framework and pre-installed apps. The approach is capable of automatically categorizing findings according to the type of data being processed, for instance to distinguish between sources providing unique identifiers and sources providing file data.

A ten-fold cross validation showed our approach to have an average precision and recall of more than 92%. On Android 4.2, SuSi finds hundreds of sources and sinks. A manual comparison with existing hand-written (categorized) lists shows that, while SuSi finds all sources and sinks of the existing lists it also finds many more that were previously unknown, thus greatly reducing the risk for analysis tools to miss privacy violations. We further showed that current approaches based on permission checks alone are inadequate as permission checks are, contrary to popular belief, not a good indicator for a method's relevance.

As future work, we aim to apply our approach to interfaces for automatically finding and classifying sensitive callbacks. We also want to further investigate how our approach can be applied to other environments than Android, e.g., J2EE. We are confident that the same concepts can also be applied to identify sources and sinks in other procedural programming languages such as C#, C++ or PHP.

9. ACKNOWLEDGMENTS

We would like to thank Christian Wirth of the Knowledge Engineering Group at TU Darmstadt for his support in the field of machine learning and for his support with WEKA. Thanks to Damien Octeau for helping us to identify sources and sinks used by TaintDroid. This work was supported by a Google Faculty Research Award, by the BMBF within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, and by the DFG within the project RUNSECURE.

10. REFERENCES

[1] Fortify 360 source code analyzer (sca), April 2013. `http://www8.hp.com/us/en/software-solutions/software.html?compURI=1214365#.UW6CVKuAtfQ`.

[2] Ibm rational appscan, April 2013. `http://www-01.ibm.com/software/de/rational/appscan/`.

[3] Scandroid, apr 2013. `https://github.com/SCanDroid`.

[4] A study of android application security - fortify rules, April 2013. `http://www.enck.org/tools/fsca_rules-final.xml`.

[5] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.

[6] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. Appguard: enforcing user requirements on android apps. In *Proceedings of the 19th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, pages 543–548, Berlin, Heidelberg, 2013. Springer-Verlag.

[7] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Automatically securing permission-based software by reducing the attack surface: an application to android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 274–277, New York, NY, USA, 2012. ACM.

[8] Leonid Batyuk, Markus Herpich, Seyit Ahmet Camtepe, Karsten Raddatz, Aubrey-Derrick Schmidt, and Sahin Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software*, MALWARE '11, pages 66–72, Washington, DC, USA, 2011. IEEE Computer Society.

[9] Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. Mast: triage for market-scale mobile malware analysis. In *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*, WiSec '13, pages 13–24, New York, NY, USA, 2013. ACM.

[10] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[11] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 393–407, 2010.

[12] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[13] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[14] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

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

[16] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *Proceedings of the 5th international conference on Trust and Trustworthy Computing*, TRUST'12, pages 291–307, Berlin, Heidelberg, 2012. Springer-Verlag.

[17] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[18] Johannes Hoffmann, Martin Ussath, Michael Spreitzenbarth, and Thorsten Holz. Slicing Droids: Program Slicing for Smali Code. In ACM, editor, *Proceedings of the 28th Symposium On Applied Computing*, pages 0–0, 2013.

[19] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '12, pages 3–14, New York, NY, USA, 2012. ACM.

[20] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. ScanDal: Static analyzer for detecting privacy leaks in android applications. In Hao Chen, Larry Koved, and Dan S. Wallach, editors, *MoST 2012: Mobile Security Technologies 2012*, Los Alamitos, CA, USA, May 2012. IEEE.

[21] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[22] Benjamin Livshits and Stephen Chong. Towards fully automatic placement of security sanitizers and declassifiers. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 385–398, New York, NY, USA, 2013. ACM.

[23] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

[24] Christopher Mann and Artem Starostin. A framework for static detection of privacy leaks in android applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1457–1462, New York, NY, USA, 2012. ACM.

[25] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 241–252, New York, NY, USA, 2012. ACM.

[26] John C. Platt. Advances in kernel methods. chapter Fast training of support vector machines using sequential minimal optimization, pages 185–208. MIT Press, Cambridge, MA, USA, 1999.

[27] J. Ross Quinlan. *C4.5: programs for machine learning.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[28] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Android permissions: a perspective combining risks and benefits. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, SACMAT '12, pages 13–22, New York, NY, USA, 2012. ACM.

[29] Karl-Michael Schneider. A comparison of event models for naive bayes anti-spam e-mail filtering. In *Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics - Volume 1*, EACL '03, pages 307–314, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.

[30] Daniel Schreckling, Joachim Posegga, Johannes Köstler, and Matthias Schaff. Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android. In *Proceedings of the 6th IFIP WG 11.2 international conference on Information Security Theory and Practice: security, privacy and trust in computing systems and ambient intelligent ecosystems*, WISTP'12, pages 208–223, Berlin, Heidelberg, 2012. Springer-Verlag.

[31] Abdallah Abbey Sebyala, Temitope Olukemi, Lionel Sacks, and Dr. Lionel Sacks. Active platform security through intrusion detection using naive bayesian network for anomaly detection. In *In: Proceedings of London communications symposium*, 2002.

[32] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 27–27, Berkeley, CA, USA, 2012. USENIX Association.

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

[34] Harry Zhang. The Optimality of Naive Bayes. In Valerie Barr and Zdravko Markov, editors, *FLAIRS Conference*. AAAI Press, 2004.

[35] Zhibo Zhao and Fernando C. Colón Osorio. "trustdroid;": Preventing the use of smartphones for information leaking in corporate networks through the used of static analysis taint tracking. In *MALWARE*, pages 135–143, 2012.