SCAnDroid: Automated Side-Channel Analysis of Android APIs

Raphael Spreitzer
Graz University of Technology

Gerald Palfinger
Graz University of Technology

Stefan Mangard
Graz University of Technology

ABSTRACT

Although the Android system has been continuously hardened against side-channel attacks, there are still plenty of APIs available that can be exploited. However, most side-channel analyses in the literature consider specifically chosen APIs (or resources) in the Android framework, after a manual analysis of APIs for possible information leaks has been performed. Such a manual analysis is a tedious, time consuming, and error-prone task, meaning that information leaks tend to be overlooked.

To overcome this tedious task, we introduce SCAnDroid, a framework that automatically profiles the Java-based Android API for possible information leaks. Events of interest, such as website launches, Google Maps queries, or application starts, are triggered automatically, and while these events are being triggered, the Java-based Android API is analyzed for possible information leaks that allow inferring these events later on. To assess the Android API for information leaks, SCAnDroid relies on dynamic time warping.

By applying SCAnDroid on Android 8 (Android Oreo), we identified several Android APIs that allow inferring website launches, Google Maps queries, and application starts. The triggered events are by no means exhaustive but have been chosen to demonstrate the broad applicability of SCAnDroid. Among the automatically identified information leaks are, for example, the `java.io.File` API, the `android.os.storage.StorageManager` API, and several methods within the `android.net.TrafficStats` API. Thereby, we identify the first side-channel leaks in the Android API on Android 8 (Android Oreo).

KEYWORDS

Android API; automatic analysis; side-channel analysis; Java API

ACM Reference Format:
Raphael Spreitzer, Gerald Palfinger, and Stefan Mangard. 2018. SCAnDroid: Automated Side-Channel Analysis of Android APIs. In *WiSec '18: Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks, June 18–20, 2018, Stockholm, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3212480.3212506

1 INTRODUCTION

The exploitation of unintended information leaks on mobile devices, denoted as side-channel analysis, represents a vivid area of research. Mobile devices, in particular, Android-based devices, represent a popular target since these devices store and process sensitive information. Among the different types of side-channel attacks [29], software-based side-channel attacks constitute a powerful attack technique since these attacks can be conducted without the attacker being physically present with the targeted device, and attack applications can be spread easily via established app markets such as Google Play. In case of software-based side-channel attacks, sensitive information unintentionally leaks to untrusted third-party applications (apps) via side channels. Such software-based side-channel attacks on mobile devices range from microarchitectural attacks [14, 26, 30, 31, 36], via sensor-based attacks [6, 7, 11, 16, 20, 22, 23, 25], to attacks exploiting the Android API and procfs resources [8, 10, 12, 13, 17, 24, 27, 28].

Nevertheless, so far most side-channel analyses focused on the exploitation of manually chosen resources, such as sensors, or the exploitation of manually selected information leaks via specific resources within the procfs or the Android API. Only recently, a framework denoted as ProcHarvester [28]—that allows analyzing procfs resources for possible information leaks automatically—has been published. However, a similar approach for the Android framework (the Java-based Android API) is still missing but would be beneficial to complement the automatic assessment of information leaked through the Android API. Especially in light of the steadily increasing number of Android APIs introduced with every new API level, such automatic frameworks are of particular importance. In contrast to ProcHarvester [28], where the side-channel assessment is based on numerical information read from procfs resources, assessing information leaks in the Android API requires the construction of objects and the invocation of methods with semantically correct parameters. Hence, the questions that we are addressing in this paper are: *(1) how do we automatically identify information leaks in the Android API?*, and *(2) given specific events of interest, what information can be exploited to infer these events?* To answer these questions, we need to address the following challenges.

(1) List of APIs. A selection of APIs, which are to be profiled for a specific Android API level, is required.
Approach: We parse the official package index of the Android API to extract classes, methods, and corresponding parameters.

(2) Valid Objects and Parameters. Valid objects must be created by passing meaningful parameters to constructors, and method calls require semantically meaningful parameters.
Approach: We aim for a mostly automated approach to select parameters for meaningful method calls, including object creation. Additionally, specific parameters can also be easily preconfigured to ensure semantically correct parameter values.

(3) Profiling Framework. The profiling of dedicated events should be done automatically, *i.e.*, the triggering of events as well as the profiling of methods on the mobile devices.
Approach: Based on the ProcHarvester framework [28], which in turn relies on the Android Debug Bridge (ADB), we automatically trigger events and command the profiling phase.

(4) Analysis Framework. A framework to automatically analyze gathered information for possible side-channel leaks is required.

WiSec '18, June 18–20, 2018, Stockholm, Sweden

Approach: Based on the ProcHarvester framework [28], which relies on dynamic time warping, we analyze the gathered information to assess possible information leaks.

By solving these challenges and by extending ProcHarvester [28], we come up with a framework denoted SCAnDroid that allows assessing information leaks within the Java-based Android API automatically. Thereby, we overcome the tedious task of manually analyzing the Android framework for side-channel leaks. Overall, we positively answer the motivating question: *The automated profiling of the Android API framework for side-channel leaks is possible.* The application of SCAnDroid on Android 8 (Android Oreo) revealed several new side-channel leaks.[1]

Contributions. The contributions of this paper are as follows:

(1) We propose a framework denoted SCAnDroid to assess side-channel leaks in the Android framework automatically.
(2) We demonstrate the applicability of SCAnDroid by automatically profiling various events of interest, such as website launches, Google Maps searches, and application starts.
(3) Based on the above-outlined events of interest, we identify new side-channel leaks within the Android API.

Outline. In Section 2, we discuss background information and related work. In Section 3, we introduce the SCAnDroid framework. In Section 4, we showcase SCAnDroid by automatically assessing information leaks for various events. In Section 5, we discuss limitations and future work. In Section 6, we conclude this work.

2 BACKGROUND

Side-Channel Attacks. Software-based side-channel attacks on mobile devices exploit contention for shared resources, e.g., contention for CPU caches in case of microarchitectural attacks [14, 26, 30, 31, 36], specific features or resources of the device, e.g., sensors in case of sensor-based attacks [6, 7, 16, 18, 20, 22, 23, 25, 33], or the Android Framework [27, 34, 37] and the procfs [8, 10, 12, 17, 24, 27, 28]. For a comprehensive overview about side-channel attacks on mobile devices we refer to [29].

In this work, we are particularly interested in side-channel attacks on the Java-based Android framework level. Among these attacks are, for example, the work of Zhou et al. [37], who exploited per-process network traffic statistics to infer activities within Twitter, WebMD, and Yahoo! Finance. Zhang et al. [34] exploited per-process traffic statistics of a surveillance app that interacts with a surveillance video camera to infer when a user's home is empty. Spreitzer et al. [27] exploited per-process network statistics of browsers to infer visited websites. In response to these attacks, the `TrafficStats.getUid[R|T]xBytes(<uid>)` API has been hardened and cannot be queried anymore for network statistics of other processes since Android 7 (Android Nougat) [5]. Hence, these attacks do not work on more recent Android versions, such as Android 7 (Android Nougat) and Android 8 (Android Oreo), anymore. However, the question arises whether there exist other Java interfaces in the Android framework that enable similar attacks.

[1]We responsibly disclosed our findings to Google. The SCAnDroid framework is available at: https://github.com/IAIK/SCAnDroid.

Automated Analysis Frameworks. While the above discussed investigations consider the exploitation of manually chosen resources, approaches towards a more systematic side-channel evaluation are quite scarce. Only recently, the need for an automated analysis framework that allows identifying side-channel leaks in the procfs on Android-based mobile devices has been motivated [28]. Currently, ProcHarvester [28] is the only framework enabling an automated analysis of procfs information leaks on Android. While dedicated events of interest are being triggered on the device, ProcHarvester automatically reads numerical values from the procfs. The gathered values are then analyzed using machine-learning approaches to identify possible information leaks in these resources.

A similar framework to assess possible information leaks in the Android API itself does not exist. Compared to ProcHarvester, a significant challenge of such a framework is the invocation of constructors and methods with valid parameters.

Android Permission System. Specific features and resources on Android are protected by dedicated permissions. For resources outside of Android apps that are considered as being non-sensitive, so-called *normal permissions* [4] are required. Although these permissions need to be specified in the Android Manifest.xml, these permissions are granted to any application without requiring the user's consent. Hence, apps relying on normal permissions only are considered as zero-permission apps. Among these normal permissions is, for example, the `INTERNET` permission.

APIs and resources that are considered as sensitive require so-called *dangerous permissions* or even *system-level permissions* [4]. These permissions also need to be specified in the Android Manifest.xml, but unlike normal permissions, these permissions need to be explicitly granted by the user. In case of dangerous permissions, the user needs to grant these permissions through a pop-up dialog. System-level permissions need to be explicitly granted for specific apps via the settings menu and, thus, also require user consent.

Threat Model. The specific type of side-channel attacks that we are interested in are software-based side-channel attacks [29], and in particular attacks on the Java-based Android API. These attacks typically consist of a *training phase* and an *attack phase*. In the training phase, the attacker models events of interest, e.g., the attacker builds templates for information leaks related to secret events. In the attack phase, the attacker distributes a malicious app, which in general does not require any permission (only normal permissions that are automatically granted without the user's consent). After the targeted user installed this malicious app, the app observes the previously identified information leaks (that can be accessed without any permission) and infers the corresponding events. SCAnDroid automatically identifies information leaks that can be exploited by such software-based side-channel attacks.

3 SCANDROID

SCAnDroid allows profiling events of interest in order to identify information leaks within the Android framework. Similar to ProcHarvester [28], SCAnDroid builds on the concept of template attacks, a conventional and powerful attack technique for side-channel analyses. In the first phase, (possibly) leaking information is modeled as templates for events of interest. In the second phase, it

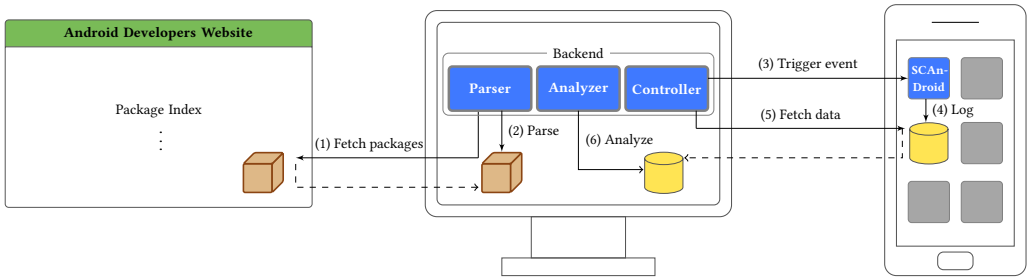

Figure 1: Basic design of SCANDROID.

is tested whether or not the established models (templates) allow to infer the corresponding events based on the observed information leaks. Such template attacks enable automatic analysis since no background knowledge about the leaking information is required.

Figure 1 illustrates the basic design of SCANDROID. In the first step, the *Parser* component fetches a list of available constructors, and methods from the Android Developers website [3], and parses this information in the second step to extract methods to be profiled. Furthermore, it extracts all permissions, including normal, dangerous, and system-level permissions. In the third step, the *Controller* component commands the smartphone to trigger events of interest, and the *Controller* also interacts with the SCANDROID logging app on the smartphone. The SCANDROID app on the smartphone relies on Java Reflections [15] to create objects according to the information retrieved from the *Parser* component and logs the corresponding methods in the fourth step. After the profiling has been finished, the *Controller* component fetches the gathered information in the fifth step. Finally, the *Analyzer* component analyzes the gathered information to identify information leaks in the sixth step.

The *Analyzer* and the *Controller* component are based on the ProcHarvester [28] framework. Subsequently, the working principle of each component is discussed in more detail.

3.1 Backend

Parser Component. Although Java Reflections allow to examine the Java API during runtime and, thus, to retrieve a list of available classes and methods at runtime, the semantics of parameter names cannot be retrieved dynamically on the device. That is, only the type of a parameter and a generic name are available, e.g., `int method(int arg0)`, but the parameter name (which often reflects the meaning and purpose of the parameter) is not available. For example, in contrast to an abstract parameter name $\texttt{arg}_0$, the meaning of `long time` or `int uid` can be inferred easily. Hence, we parse the Android Developer documentation [3] to establish a list of constructors and methods, including the corresponding parameters, *i.e.*, their type and name information. Based on this list, we extract methods of interest with a specific prefix. The method prefixes that are currently being considered as relevant are: `get`, `has`, `is`, and `query`. We focus on these prefixes since so-called accessor methods—used to access encapsulated member variables in object-oriented programming languages such as Java—are typically prefixed with these keywords.

Furthermore, the parser component extracts all permissions, which allow us to analyze information leaks according to the three permission groups: (1) zero-permission apps, (2) dangerous permission apps, and (3) system-level permission apps [4]. Of particular interest in the context of software-based side-channel attacks are information leaks that can be exploited by zero-permission apps, *i.e.*, apps that only require normal permissions.

Controller Component. The *Controller* component interacts with the SCANDROID app on the smartphone to command the profiling via the Android Debug Bridge (ADB) [1], e.g., to start and stop the logging process on the device. Furthermore, the controller programmatically triggers events of interest on the connected smartphone. Currently, SCANDROID supports triggering website launches, Google Maps search queries, and application starts, but SCANDROID can be easily extended to support further events. The list of events is by no means exhaustive. It is only intended to demonstrate the working principle of SCANDROID. After all events of interest have been triggered, the controller fetches the gathered information for further analysis.

Analyzer Component. The time series gathered for one method and one triggered event forms a so-called *trace*, and the *Analyzer* component analyzes these time series using machine learning. Therefore, it relies on dynamic time warping (DTW),[2] which aims to find a warping path with minimal distance between two time series, and the Python framework *scikit-learn* [21]. The appealing benefit of dynamic time warping is that it allows to compare time series of different lengths, and also to identify similarities in time series that vary in speed. Thus, DTW identifies similarities even if the information is misaligned, stretched, or compressed (cf. [19, 32]).

Before the actual processing, all traces are normalized by subtracting the mean. All traces are combined in the so-called training data $T = \{(e_i, t_i)\}$, which holds tuples of events (labels) e_i and their corresponding time series t_i. To evaluate whether a specific resource leaks information, we rely on k-fold cross validation. Hence, we split the training data $T = \{(e_i, t_i)\}$ into k folds, take each fold $T_f = \{(e_i, t_i)\}$ (one at a time) and compute the warping paths between all tuples in this fold and all remaining tuples in the training data $T \setminus T_f$. Let us denote the tuple with the minimal warping path between t_i and t_j as (e_j, t_j). If e_i equals e_j, then the event has been inferred correctly. The ratio of correctly inferred events and the total number of traces in the current fold T_f indicates the inference accuracy, which represents a metric for information leakage. The higher the inference accuracy, the more accurate events can be

[2] https://github.com/honeyext/cdtw

inferred and, thus the more information leaks. To obtain a robust inference accuracy, we average the resulting inference accuracies over all folds, which is the usual approach of k-fold cross validation. We consider an API to leak information if the inference accuracy is better than randomly guessing the corresponding event.

Although we also tried to automatically extract dedicated features from the time series using *tsfresh* [9], and to train (more sophisticated) supervised classifiers, the classification results were not that promising. In our experiments, DTW seems to yield the best results in terms of an automated information leakage analysis.

3.2 Smartphone Service

SCANDROID App. The profiling of methods of interest is done in an Android app (a background service) on the mobile device or an emulator, similar to the ProcHarvester [28] framework. Compared to ProcHarvester, the actual profiling, however, requires more engineering effort and significantly more pre-processing since SCANDROID needs to create objects and to invoke methods dynamically, instead of reading numerical information from procfs resources.

Listing 1 indicates the approach to inspect Java APIs during runtime. More specifically, Java Reflections allow inspecting available packages as well as classes. For each class, Java Reflections further allow retrieving constructors, to create objects, as well as to inspect and invoke methods on these objects. Inspecting the full Android API using Java Reflections, however, requires a significant amount of information (e.g., about classes, constructors, methods, and their corresponding parameters). Thus, inspecting the whole API at once is a memory intensive task, which requires more RAM than what might be available on today's mobile devices.

Listing 1: Inspecting Java APIs using Java Reflections.

```
// For all available packages and all available classes
Class c = Class.forName("android...");
Constructor[] ctors = c.getDeclaredConstructors();
// Get types of parameters
for (Constructor ctor : ctors) {
  Class<?>[] parameters = ctor.getParameterTypes();
  // Create parameter objects recursively
  ArrayList<Object[]> parameterObjects;
  for (Class parameter : parameters) {
      parameterObjects.add(...);
  }

  // Permute the created parameter objects
  // in case of multiple constructors for one parameter
  permuteParameters(parameterObjects);
  for (Object[] parameterObject : parameterObjects) {
    // Create object of each constructor
    classObjects.add(ctor.newInstance(parameterObject));
  }
}
// Get available methods
Method[] methods = c.getDeclaredMethods();
for (Method m : methods) {
  for (Object classObject : classObjects) {
    // Get available parameters for methods
    Class<?>[] parameters = m.getParameterTypes();
    // Create and permute parameter objects
    // for each method invocation
    ArrayList<Object[]> parameterObjects =
      createAndPermutateObjects(parameters);
    for (Object[] parameterObject : parameterObjects) {
      Object returnValue =
        m.invoke(classObject, parameterObject);
      if (!isPrimitiveType(returnValue)) {
        // explore returned object recursively
      }
    }
  }
}
```

Recursion Flattening. To cope with this memory intensive task, we split the total number of packages in the Android API into several parts. For example, in case of our Nexus 5X with 2 GB of RAM, we split the Android API into four parts, which allows us to perform this step without running out of memory. Now, each of these four parts is inspected and prepared as depicted in Listing 1.

To improve performance and to enable profiling at all, we flatten the recursive call hierarchy. More specifically, for all methods of interest and their corresponding objects (which are required to invoke the methods of interest), we perform the following step: Invoke `obj.method(...)` and process the return value as follows.

- If it returns primitive data types, such as boolean and numerical values—either as primitive data type, in an array, or in a collection of a subclass of `java.util.Collection` (e.g, `ArrayList`)—the method and the corresponding object are stored (in a flattened list) to be invoked during the profiling step later on.
- If it returns an `Object`, recursively invoke methods of interest on this `Object` and repeat processing the return value(s) until the pre-defined recursion depth is reached.

For the recursion flattening step, the configuration file allows specifying two parameters. First, the recursion depth is currently set to 2, which is a trade-off between performance (overall runtime and memory consumption), and exhaustiveness (coverage) of analyzed methods. Second, it allows specifying classes for which the corresponding objects (used to invoke a method of interest) should be re-generated before each method invocation. For example, we specified that `NetworkStats.Bucket` should be re-generated every time methods of interest in this class are invoked, *i.e.*, we invoke each method of interest on a new object.

Parameter Selection. Although the SCANDROID app generates random parameters for most method invocations, the corresponding parameters of constructors and methods can also be manually analyzed in terms of their semantics to provide meaningful values, e.g., in case of start and end times or storage `UUIDs`. In general, we follow a trial and error approach to choose parameters for constructor and method invocations. That is, in the first place, SCANDROID randomly chooses parameters. If object creation or method invocation fails, e.g., due to an exception, we manually analyze the resulting errors and thrown exceptions to pre-configure parameters where random parameter selection seems to be the problem. Besides static parameter values, such as specific start or end times, SCANDROID also supports dynamic parameter generation as well as array-type parameters. For example, in case of a time stamp, "`System.currentTimeMillis()+10`" could be specified as parameter value for specific methods, which are evaluated before the actual method invocation. Array-type parameters can be specified in case a particular method or constructor should be invoked multiple times with different parameters. In total, we defined 73 values, including array-type parameters, for 11 parameter names.

Pre-Profiling. After the objects and methods to be called as well as the corresponding parameters have been acquired, events are triggered continuously while the prepared methods are invoked simultaneously. If the returned value of a method changes at least once during two event triggering phases, we consider the corresponding method in the subsequent profiling phase. This allows

Table 1: Coverage of analyzed methods for API level 27 (Stock Android 8.1).

Methods	#	%
Available according to Java Reflections	53 913	-
Documented in the Android API	36 339	-
Public and considered relevant (`get`, `is`, `has`, `query`)	**12 012**	100.0%
In abstract classes or interfaces	2 860	23.8%
Corresponding object creation fails (e.g., due to exception or missing parameter)	3 664	30.5%
Corresponding class not found (`android.test`, `junit.runner`)	208	1.7%
Removed (e.g., due to segmentation faults in the `android.graphics` package)	511	4.3%
Crash (e.g., due to missing permissions or incorrect parameters)	692	5.8%
Theoretically to be profiled	4077	33.9%
Actually profiled	**5 056**	**42.1%**
Do not change for website launches	5 020	-
Change for website launches	36	-
Do not change for Google Maps queries	5 020	-
Change for Google Maps queries	36	-
Do not change for app starts	5 019	-
Change for app starts	37	-

us to significantly reduce the set of actual methods to be profiled, as the majority of methods do not react while events are being triggered. Overall, this pre-profiling step identifies methods that possibly leak information and, thus, should be profiled in the next step to assess possible information leaks.

Profiling. The actual profiling is the least intensive step in terms of computations and memory consumption. The corresponding methods are continuously invoked, while the events are being triggered simultaneously on the device via the backend. The returned values of these methods are gathered in dedicated files, which will be analyzed by the *Analyzer* component in the backend later on.

Performance. On our test device, it takes about 7–8 hours for the whole API (all four parts) to be processed. The most intensive part is the pre-profiling step, which prepares the methods to be profiled. The actual profiling depends on the number of events to be triggered and the profiling time for each event. For example, in case of triggering websites, we consider 20 websites, 8 samples per website, a profiling time of 10 seconds, and a cool-down period of 2 seconds between two events,[3] resulting in a profiling time of $20 \cdot 8 \cdot (10 + 2) = 1\,920\,s$, *i.e.*, about half an hour. The subsequent k-fold cross validation based on DTW takes about one minute.

3.3 Coverage Analysis

Table 2 depicts the configurations of our test device. Besides performing the side-channel assessment, we also use this device to investigate the coverage of profiled Android APIs by SCAnDroid, which is indicated in Table 1. Interestingly, according to Java Reflections, the number of methods available (53 913) is larger than the number of methods documented in the Android API (36 339). This is due to the fact that Java Reflections allow inspecting private and protected methods, which are, in general, not documented as they are only available within a specific class or package, respectively.

As mentioned above, we consider methods to be of interest, if they are prefixed with a specific keyword, such as `get`, `has`, `is`, or `query`. Out of the 12 012 public methods that we consider as being relevant, 2 860 methods are defined in abstract classes or interfaces and, thus, cannot be invoked directly. For 3 664 non-static methods, a corresponding object could not be created, e.g., due to exceptions while creating the object, missing constructors, or because parameter construction fails. For 208 methods, the corresponding class required to call the method could not be found, e.g., in the packages `android.test`, and `junit.runner`. 511 methods have been removed due to segmentation faults in the underlying native libraries (mostly inside the `android.graphics` package). 692 methods crash during the profiling, e.g., they throw an exception due to a missing permission or an incorrect parameter and are thus not considered in the side-channel assessment. In theory, SCAnDroid should thus be able to cover about 4 077 methods of interest, *i.e.*, 33.9% of the relevant methods. However, due to the recursive approach followed by SCAnDroid, we also use objects returned by recursive method invocations to further increase the coverage to 5 056 methods of interest (42.1%). For example, `Context.getSystemService(...)` returns `NetworkStats` objects, which cannot be created directly.

Table 2: Device configurations.

Device	Android
Nexus 5X (2 GB RAM)	LineageOS 15.1 (Android 8.1)
Nexus 5X (2 GB RAM)	Stock Android 8.1

We profiled these 5 056 methods while triggering events of interest, *i.e.*, website launches, Google Maps queries, and app starts. The list of websites, apps, and points of interests (POIs) used for the evaluations are given in Appendix A. Table 1 also shows that many methods do not react while events are being triggered. Hence, the information returned from these methods (probably) cannot be exploited. Although the information might not be exploitable directly, another side-channel leak might still exist. For example, a timing variation for different events or parameters might be observable. This particular case is, however, beyond the scope of this paper.

Our experiment also revealed that 1 096 public methods, which we consider relevant for side-channel analysis, e.g., prefixed with `get`, `has`, `is` or `query`, have been added in Android 8.0 (Android Oreo). This increasing number of APIs also stresses the importance of an automatic approach to assess possible information leaks.

[3] These parameters are chosen similar to other website fingerprint attacks [12, 27].

Table 3: Information leaks for website fingerprinting on Android 8.1. Accuracy evaluated for 20 websites. All APIs can be accessed by zero-permission apps.

API	Accuracy
`android.net.TrafficStats.getMobileTxBytes()`	89.4 %
`android.net.TrafficStats.getTotalTxBytes()`	88.8 %
`android.net.TrafficStats.getMobileTxPackets()`	86.2 %
`android.net.TrafficStats.getTotalRxPackets()`	85.6 %
`android.net.TrafficStats.getTotalTxPackets()`	85.0 %
`android.net.TrafficStats.getMobileRxPackets()`	83.1 %
`android.net.TrafficStats.getTotalRxBytes()`	79.4 %
`android.net.TrafficStats.getMobileRxBytes()`	76.2 %
`android.app.usage.StorageStatsManager.getFreeBytes(java.util.UUID)`	46.9 %
`java.io.File.getUsableSpace()`	39.4 %
`java.io.File.getFreeSpace()`	38.1 %
`android.os.storage.StorageManager.getAllocatableBytes(java.util.UUID)`	36.2 %
`android.os.Process.getElapsedCpuTime()`	21.9 %

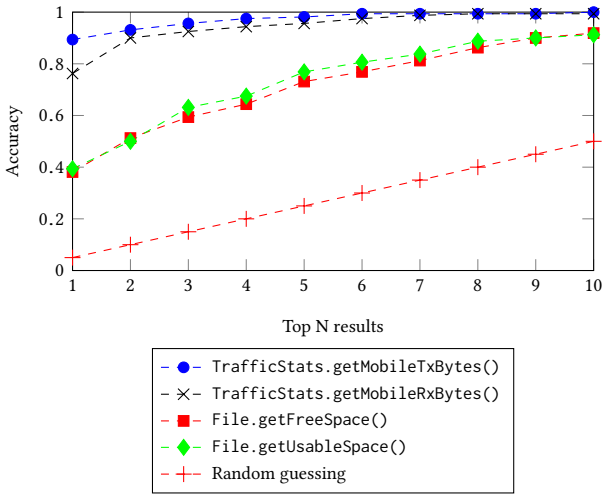

Figure 3: Inference accuracy for websites when considering the top N results returned from the classifier.

4 CASE STUDIES

In this section, we demonstrate the applicability of SCANDROID by automatically analyzing the Android API for possible information leaks in various attack scenarios.

4.1 Website Fingerprinting

The idea of website fingerprinting attacks is to observe side-channel information on mobile devices and to determine the websites a user navigates to. Such attacks represent a severe privacy threat as has been argued in [12, 27]. To assess possible information leaks in the Android API that allow conducting such website fingerprinting attacks, we automatically launch websites in the browser application and simultaneously profile the Android API in the background. Similar to [12, 27], we considered a profiling duration of 10 seconds for each website and gathered 8 samples (time series) per website.

Normal Permissions (Zero-Permission Apps). Table 3 depicts information leaks that enable website fingerprinting on Android 8.1 (Android Oreo) when launching websites in Google Chrome 64. The accuracy has been evaluated for the top 20 websites according to alexa.com, where duplicates have been removed. Although per-application statistics have been restricted in Android 7 (Android Nougat), SCANDROID revealed that global statistics released via the `TrafficStats` API still allow website fingerprinting.

The `java.util.UUID` required by the `StorageStatsManager` and `StorageManager` API represents a 128-bit ID identifying the storage volume. This parameter has been set in the configuration file in order to provide a meaningful parameter. Interestingly, these APIs have been added in Android 8 (Android Oreo) and, thus, represent new information leaks introduced in the latest Android version.

While the network statistics provided via the `TrafficStats` API might be considered as more obvious information leaks, the information leaks via the `File` API are more subtle. More specifically, the API `File.getUsableSpace()` reports the number of bytes available to non-root users, and `File.getFreeSpace()` reports the number of bytes available to root users [2]. To the best of our knowledge, there is no study yet discussing the possibility to infer visited websites based on these APIs. Even if an inference accuracy of about 40% for the API `java.io.File.getUsableSpace()` seems to be quite moderate, randomly guessing the correct website out of a set of 20 websites yields an accuracy of $\frac{1}{20} = 5\%$. Thus, SCANDROID clearly identified an information leak.

Figure 2 illustrates traces for three of the identified information leaks for amazon.com and reddit.com. We observe similarities between traces for the same website and differences between different websites. SCANDROID successfully detected the corresponding information leaks using DTW. Experiments on Android 7 (Android Nougat) revealed similar results and have been omitted.

Top N Metric. A more conservative metric to identify information leaks is to consider the accuracy when taking the top N results returned from the classifier into consideration, which is illustrated in Figure 3. If we consider the top 3 websites returned from the classifier based on DTW, we achieve an inference accuracy of more than 60% for these information leaks. Furthermore, we observe that all information leaks significantly outperform random guessing.

Dangerous and System-Level Permissions. Considering APIs that require dangerous permissions, SCANDROID also identified APIs that can be exploited to conduct side-channel attacks. Table 4 depicts the identified information leaks that allow website fingerprinting on Android 8.1 (Android Oreo) when launching websites in Google Chrome 64. We provide these results here for the sake of completeness only since these resources cannot be exploited by zero-permission apps.

In order to retrieve a `NetworkStats.Bucket` object, SCANDROID invoked `NetworkStatsManager.querySummaryForDevice(. . .)` with parameters `int networkType`, `String subscriberId`, `long startTime`, and `long endTime`. Querying this method requires the `PACKAGE_USAGE_STATS` (system-level) permission, and the parameters have been pre-configured in the configuration file. For instance, `networkType` has been set to `TYPE_MOBILE`, `subscriberId`

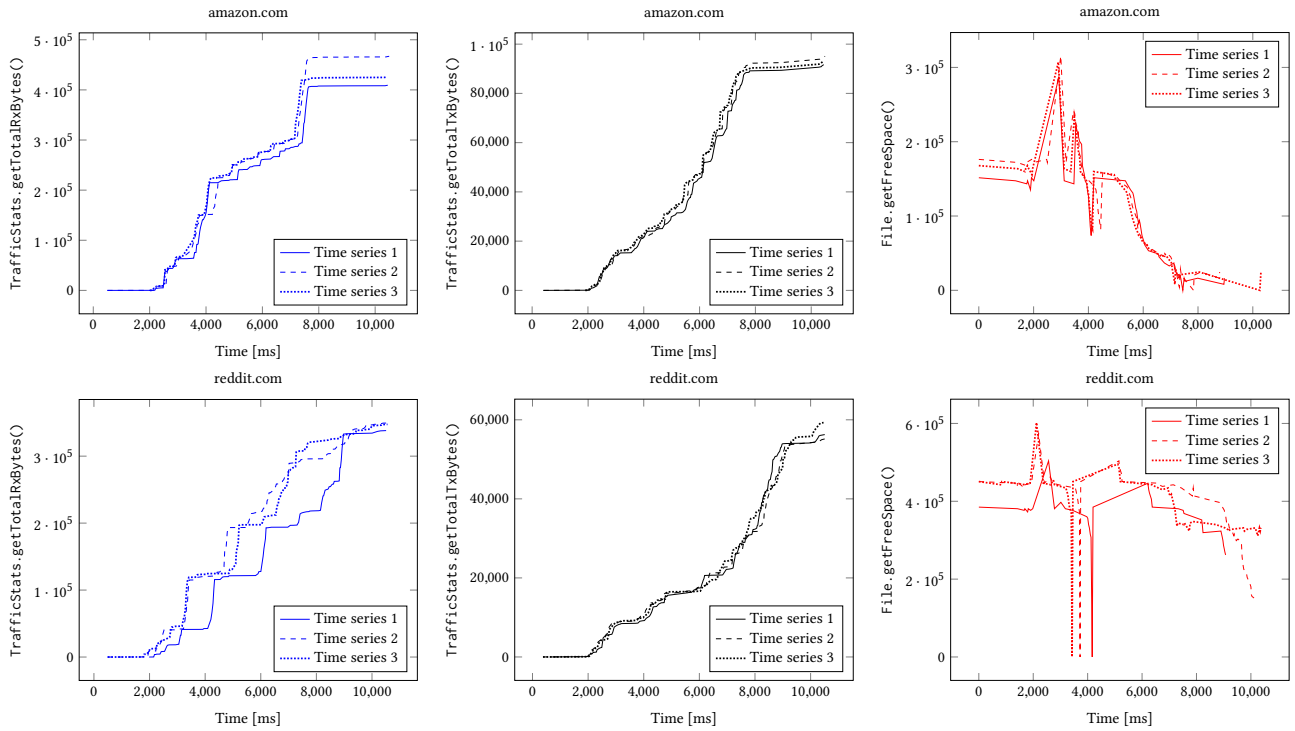

Figure 2: `TrafficStats.getTotalRxBytes()`, `TrafficStats.getTotalTxBytes()`, and `File.getFreeSpace()` when launching amazon.com and reddit.com, respectively.

Table 4: Information leaks for website fingerprinting on Android 8.1. Accuracy evaluated for 20 websites. All APIs require dangerous or system-level permissions.

API(s) requiring `PACKAGE_USAGE_STATS` (system-level permission)	Accuracy
`android.app.usage.NetworkStats.Bucket.getTxBytes()`	90.0 %
`android.app.usage.NetworkStats.Bucket.getTxPackets()`	85.6 %
`android.app.usage.NetworkStats.Bucket.getRxPackets()`	83.1 %
`android.app.usage.NetworkStats.Bucket.getRxBytes()`	76.9 %
API(s) requiring `READ_PHONE_STATE` (dangerous permission)	
`android.telephony.TelephonyManager.getDataActivity()`	20.0 %

has been set to "`TelephonyManager.getSubscriberId()`", which in turn requires the `READ_PHONE_STATE` (dangerous) permission, `startTime` has been set to `System.currentTimeMillis()`, and `endTime` has been set to "`System.currentTimeMillis()+100`", respectively. This example demonstrates that SCAnDroid is able to handle complex method invocations.

We verified the information leaks by manually launching the 20 websites in the browser and monitoring the identified information leaks in the background. As shown in Appendix B, the identified APIs also leak in case of manually triggering the event.

4.2 Google Maps Search Inference

Inspired by Zhang et al. [35], who suggested to infer search queries in Apple Maps by exploiting side-channel information, we consider such a scenario on Android. Such an attack allows zero-permission apps to infer what places a user might want to visit or what places the user is interested. Again, to automatically assess possible information leaks in the Android API, we automatically search for various POIs in Google Maps and simultaneously profile the Android API for 15 seconds in the background.

Normal Permissions (Zero-Permission Apps). Table 5 depicts the identified information leaks for inferring Google Maps search queries on Android 8.1 (Android Oreo). The accuracy has been evaluated for 20 POIs around the world, and 8 samples per POI. Again, the `TrafficStats` API reporting global network traffic statistics allows inferring these events with a high accuracy. Although `java.io.File.getUsableSpace()` achieves an accuracy of 10.6% only, it still represents an information leak since randomly guessing would result in an accuracy of $\frac{1}{20} = 5\%$.

Figure 4 illustrates traces for three of the identified information leaks when querying Google Maps for Eiffel Tower and The Great Wall, respectively. For `TrafficStats.getMobileRxBytes()` and `TrafficStats.getMobileTxBytes()` we observe similarities between traces for the same POI and differences between the two POIs. However, while the time series of `File.getUsableSpace()` for The Great Wall seem to be correlated, the time series for the Eiffel Tower only share a few peaks and patterns, which decreases the accuracy. Nevertheless, exploiting this API by matching gathered time series with DTW still outperforms random guessing attacks.

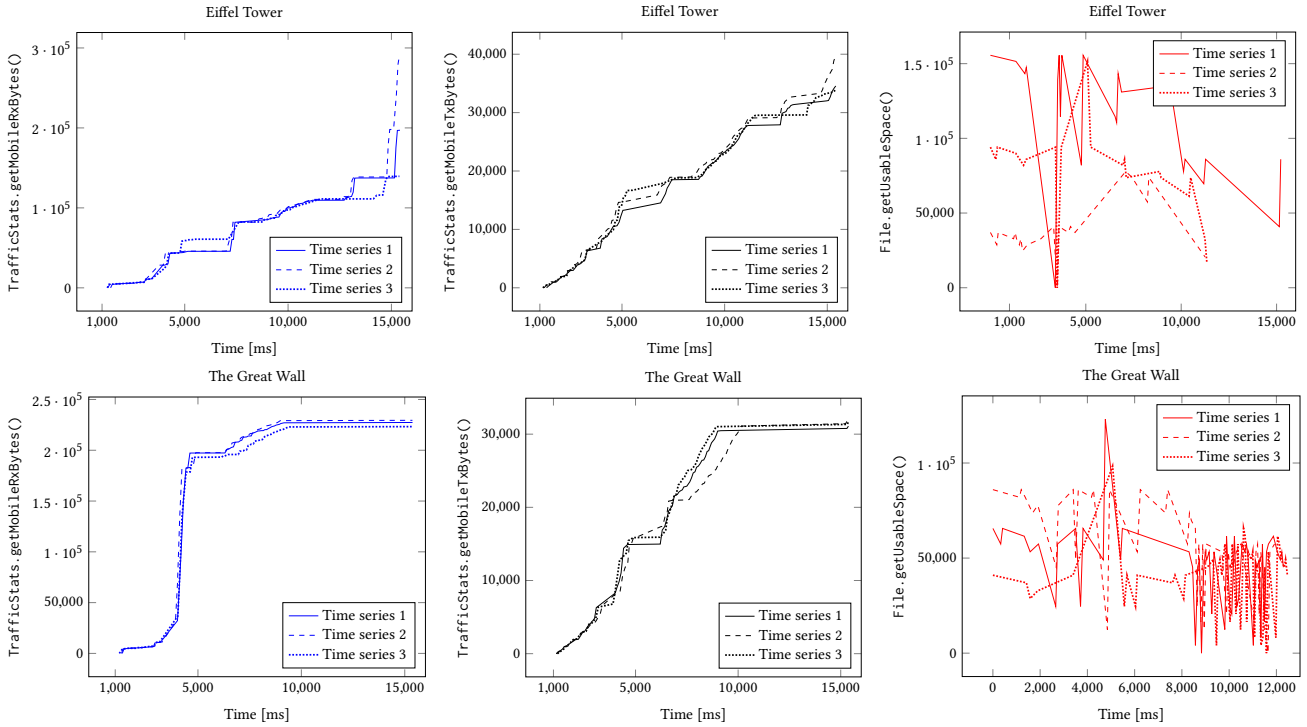

Figure 4: `TrafficStats.getMobileRxBytes()`, `TrafficStats.getMobileTxBytes()`, and `File.getUsableSpace()` when searching for Eiffel Tower and The Great Wall, respectively.

Table 5: Information leaks for Google Maps searches on Android 8.1. Accuracy evaluated for 20 POIs. All APIs can be accessed by zero-permission apps.

API	Accuracy
`android.net.TrafficStats.getTotalRxBytes()`	87.5 %
`android.net.TrafficStats.getMobileRxBytes()`	83.8 %
`android.net.TrafficStats.getMobileRxPackets()`	76.2 %
`android.net.TrafficStats.getTotalRxPackets()`	73.1 %
`android.net.TrafficStats.getTotalTxPackets()`	68.1 %
`android.net.TrafficStats.getMobileTxPackets()`	66.9 %
`android.net.TrafficStats.getTotalTxBytes()`	49.4 %
`android.net.TrafficStats.getMobileTxBytes()`	48.8 %
`android.app.usage.StorageStatsManager.getFreeBytes(java.util.UUID)`	16.2 %
`android.os.storage.StorageManager.getAllocatableBytes(java.util.UUID)`	13.1 %
`android.os.Process.getElapsedCpuTime()`	13.1 %
`java.io.File.getFreeSpace()`	11.9 %
`java.io.File.getUsableSpace()`	10.6 %

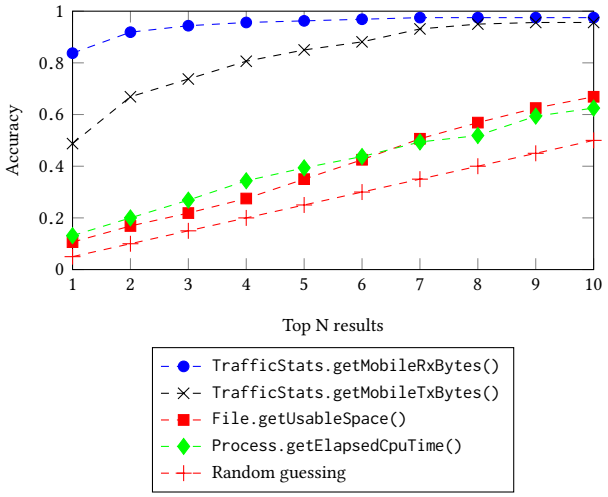

Figure 5: Inference accuracy for Google Maps queries when considering the top N results returned from the classifier.

Top N Metric. Figure 5 illustrates the inference accuracy when considering the top N results returned from the classifier. Again, all information leaks outperform random guessing attacks.

Dangerous and System-Level Permissions. Table 6 depicts the identified information leaks that allow inferring Google Maps searches on Android 8.1 (Android Oreo) when considering apps that request dangerous permissions. Thus, these resources cannot be exploited by zero-permission apps. The `NetworkStats.Bucket` object has been retrieved as described in Section 4.1.

Table 6: Information leaks for Google Maps searches on Android 8.1. Accuracy evaluated for 20 POIs. All APIs require system-level or dangerous permissions.

API(s) requiring PACKAGE_USAGE_STATS (system-level permission)	Accuracy
android.app.usage.NetworkStats.Bucket.getRxBytes()	85.0 %
android.app.usage.NetworkStats.Bucket.getRxPackets()	75.0 %
android.app.usage.NetworkStats.Bucket.getTxPackets()	64.4 %
android.app.usage.NetworkStats.Bucket.getTxBytes()	49.4 %
API(s) requiring READ_PHONE_STATE (dangerous permission)	
android.telephony.TelephonyManager.getDataActivity()	18.8 %

Table 7: Information leaks for app inference on Android 8.1. Accuracy evaluated for 20 apps. All APIs can be accessed by zero-permission apps.

API	Accuracy
android.net.TrafficStats.getTotalRxBytes()	85.6 %
android.net.TrafficStats.getMobileRxBytes()	84.4 %
android.net.TrafficStats.getTotalTxPackets()	83.1 %
android.net.TrafficStats.getMobileTxPackets()	82.5 %
android.net.TrafficStats.getTotalRxPackets()	81.9 %
android.net.TrafficStats.getMobileTxBytes()	80.0 %
android.net.TrafficStats.getMobileRxPackets()	80.0 %
android.net.TrafficStats.getTotalTxBytes()	76.9 %
android.os.storage.StorageManager. getAllocatableBytes(java.util.UUID)	61.2 %
java.io.File.getFreeSpace()	60.6 %
java.io.File.getUsableSpace()	57.5 %
android.app.usage.StorageStatsManager. getFreeBytes(java.util.UUID)	53.1 %
android.os.Process.getElapsedCpuTime()	35.6 %

4.3 Application Start Inference

The idea of app start inference attacks is to observe side-channel information in order to learn currently executed applications on a mobile device. Currently executed apps represent sensitive information as this information enables targeted attacks, such as stealing login credentials [8]. Therefore, since Android 5 (Android Lollipop) the list of running apps cannot be retrieved by third-party applications anymore. Still by exploiting side-channel information from the procfs [10, 28] it is possible to infer running applications. Again, in order to assess possible information leaks in the Android API, we automatically start applications and simultaneously profile the Android API for 8 seconds in the background.

Normal Permissions (Zero-Permission Apps). Table 7 depicts the identified information leaks for application cold starts on Android 8.1 (Android Oreo). The accuracy has been evaluated for 20 apps and 8 samples per app. Again, global network statistics allow inferring application cold starts with a high accuracy.

Top N Metric. Figure 6 indicates the inference accuracy when considering the top N results returned from the classifier. Again, this plot demonstrates that the identified information leaks outperform random guessing attacks.

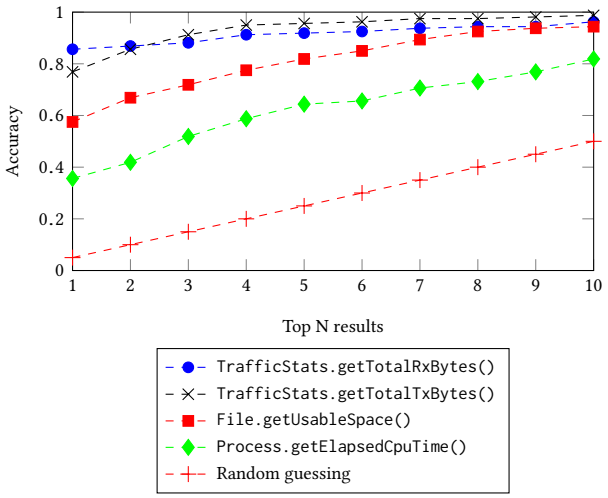

Figure 6: Inference accuracy for app starts when considering the top N results returned from the classifier.

Table 8: Information leaks for application inference on Android 8.1. Accuracy evaluated for 20 apps. All APIs require system-level permissions.

API(s) requiring PACKAGE_USAGE_STATS (system-level permission)	Accuracy
android.app.usage.NetworkStats.Bucket.getRxBytes()	85.0 %
android.app.usage.NetworkStats.Bucket.getRxPackets()	83.8 %
android.app.usage.NetworkStats.Bucket.getTxPackets()	78.8 %
android.app.usage.NetworkStats.Bucket.getTxBytes()	78.8 %

Dangerous and System-Level Permissions. Table 8 depicts the identified information leaks that allow inferring application starts on Android 8.1 (Android Oreo) when considering apps that request dangerous permissions. Thus, these resources cannot be exploited by zero-permission apps. The `NetworkStats.Bucket` object has been retrieved as described in Section 4.1.

5 DISCUSSION

Limitations. Similar to the limitations of ProcHarvester [28], SCAnDroid also suffers from false negatives. That is, even though SCAnDroid does not report an information leak for a specific event and a specific API level, this does not guarantee that there are indeed no information leaks present. However, automated analysis tools always aim to generalize the detection of information leaks and, thus, possibly suffer from false negatives.

Countermeasures. We believe that an automated tool such as SCAnDroid represents a valuable tool for OS developers to test whether or not a newly introduced API interface leaks sensitive information. Thereby, identified information leaks could be prevented in the first place, *i.e.*, before the corresponding API interfaces are released. Currently, it seems that side-channel leaks on the API level need to be addressed on the OS level. For example, by restricting access to these APIs or by releasing more coarse-grained information in case of some statistics.

More generic approaches such as App Guardian [34] that aim to detect ongoing side-channel attacks by monitoring side-channel information of currently executed applications represent an interesting approach to cope with software-based side-channel attacks. However, App Guardian relies on resources that have been restricted in newer Android versions and, thus, this approach needs to be constantly updated in order to deal with these restrictions.

Future Work. As the application of SCANDROID demonstrated, the Android API exposes many exploitable interfaces that enable side-channel attacks, even on the latest Android 8 (Android Oreo). However, considering the complexity of the Java-based Android API, there is still room for improvement for automatic frameworks such as SCANDROID. For instance, to further increase the coverage of the analyzed Android framework, SCANDROID could be extended to support Android-specific callback interfaces. More specifically, Android handles sensor events via so-called listeners. Thereby, an app registers sensor-event listeners that get notified in case of sensor updates. Currently, such listener APIs are not covered by SCANDROID.

In addition, as our coverage analysis in Section 3.3 revealed, there are still a few hundred methods left that have not been profiled so far. Especially Java interfaces that call native libraries, e.g., in case of the `android.graphics` package, crash due to segmentation faults. These faults require further analysis in order to determine why these interfaces crash, and whether or not these interfaces can be profiled by specifying pre-configured parameters in the configuration file (instead of randomly chosen parameters). For example, it might be possible that the parameters chosen by SCANDROID are outside of a defined range, which causes the method to crash. Nevertheless, with a coverage of about 42.1% of all public methods that are prefixed with `get`, `has`, `is`, and `query`, several new information leaks have been detected in the Android API.

As already mentioned in Section 3.3, a method might expose a different timing behavior due to different input parameters or due to different events being triggered. For example, on iOS it has been shown [35] that the `fileExistsAtPath` API yields a different timing depending on whether or not the passed file exists, even if the running app is not allowed to read the file. Future work should possibly consider the extension of automated frameworks to cover such information leaks as well.

6 CONCLUSION

In this paper we introduced SCANDROID, a framework to automatically assess the Java-based Android API for possible side-channel leaks. To enable such an automatic assessment of side-channel leaks in the Android framework, we had to tackle the challenge of dealing with the variety of Java APIs.

The need for a framework such as SCANDROID arises from the observation that manual analysis of side-channel information leaks is tedious and error-prone, *i.e.*, information leaks tend to be overlooked, especially in case of very subtle information leaks. Thus, we argue that automated frameworks for such tasks are inevitable. First, to investigate existing APIs and to fix identified information leaks and, second, to identify information leaks of APIs upfront.

We demonstrated the applicability of SCANDROID in practice by considering various side-channel attacks—already known from manual investigations in the literature—and we identified several information leaks automatically. These findings clearly underline the effectiveness of our chosen approach. Especially network traffic statistics—even though published only globally since Android 7 (Android Nougat)—represent sensitive information that allows inferring most of the triggered events rather accurately. Mostly because smartphones are connected to the Internet all the time, and heavily rely on Internet connectivity to provide the desired functionality. Hence, the publication of such statistics must be reconsidered.

Overall we believe that SCANDROID advances the API-based side-channel analysis on mobile devices and hopefully helps to prevent various information leaks in the first place.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644052 (HECTOR). This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

REFERENCES

[1] Android Developers. n. d.. Android Debug Bridge (ADB). https://developer.android.com/studio/command-line/adb.html. Accessed: February 2018.
[2] Android Developers. n. d.. File API. https://developer.android.com/reference/java/io/File.html. Accessed: February 2018.
[3] Android Developers. n. d.. Package Index. https://developer.android.com/reference/packages.html. Accessed: February 2018.
[4] Android Developers. n. d.. Permissions Overview. https://developer.android.com/guide/topics/permissions/overview.html. Accessed: February 2018.
[5] Android Developers. n. d.. TrafficStats API. https://developer.android.com/reference/android/net/TrafficStats.html. Accessed: February 2018.
[6] Adam J. Aviv, Benjamin Sapp, Matt Blaze, and Jonathan M. Smith. 2012. Practicality of Accelerometer Side Channels on Smartphones. In *Annual Computer Security Applications Conference – ACSAC 2012*. ACM, 41–50.
[7] Liang Cai and Hao Chen. 2011. TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion. In *USENIX Workshop on Hot Topics in Security – HotSec*. USENIX Association.
[8] Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. 2014. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *USENIX Security Symposium 2014*. USENIX Association, 1037–1052.
[9] Maximilian Christ, Andreas W. Kempa-Liehr, and Michael Feindt. 2016. Distributed and Parallel Time Series Feature Extraction for Industrial Big Data Applications. *arXiv ePrint Archive, Report 1610.07717* (2016).
[10] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. 2016. No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis. In *IEEE Symposium on Security and Privacy – S&P 2016*. IEEE Computer Society, 414–432.
[11] Muzammil Hussain, Ahmed Al-Haiqi, A. A. Zaidan, B. B. Zaidan, Miss Laiha Mat Kiah, Nor Badrul Anuar, and Mohamed Abdulnabi. 2016. The Rise of Keyloggers on Smartphones: A Survey and Insight Into Motion-Based Tap Inference Attacks. *Pervasive and Mobile Computing* 25 (2016), 1–25.
[12] Suman Jana and Vitaly Shmatikov. 2012. Memento: Learning Secrets from Process Footprints. In *IEEE Symposium on Security and Privacy – S&P 2012*. IEEE Computer Society, 143–157.
[13] Chia-Chi Lin, Hongyang Li, Xiao-yong Zhou, and XiaoFeng Wang. 2014. Screenmilker: How to Milk Your Android Screen for Secrets. In *Network and Distributed System Security Symposium – NDSS 2014*. The Internet Society.
[14] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium 2016*. USENIX Association, 549–564.
[15] Glen McCluskey. 1998. Using Java Reflections. http://www.oracle.com/technetwork/articles/java/javareflection-1536171.html. Accessed: February 2018.
[16] Maryam Mehrnezhad, Ehsan Toreini, Siamak Fayyaz Shahandashti, and Feng Hao. 2016. TouchSignatures: Identification of User Touch Actions and PINs based

on Mobile Sensor Data via JavaScript. *J. Inf. Sec. Appl.* 26 (2016), 23–38.
[17] Yan Michalevsky, Aaron Schulman, Gunaa Arumugam Veerapandian, Dan Boneh, and Gabi Nakibly. 2015. PowerSpy: Location Tracking Using Mobile Device Power Analysis. In *USENIX Security Symposium 2015*. USENIX Association, 785–800.
[18] Emiliano Miluzzo, Alexander Varshavsky, Suhrid Balakrishnan, and Romit Roy Choudhury. 2012. Tapprints: Your Finger Taps Have Fingerprints. In *Mobile Systems – MobiSys 2012*. ACM, 323–336.
[19] Meinard Müller. 2007. *Dynamic Time Warping*. Springer, 69–84. https://doi.org/10.1007/978-3-540-74048-3
[20] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. 2012. ACCessory: Password Inference Using Accelerometers on Smartphones. In *Mobile Computing Systems and Applications – HotMobile 2012*. ACM, 9.
[21] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[22] Dan Ping, Xin Sun, and Bing Mao. 2015. TextLogger: Inferring Longer Inputs on Touch Screen Using Motion Sensors. In *Security and Privacy in Wireless and Mobile Networks – WISEC 2015*. ACM, 24:1–24:12.
[23] Laurent Simon and Ross J. Anderson. 2013. PIN Skimmer: Inferring PINs Through the Camera and Microphone. In *Security and Privacy in Smartphones & Mobile Devices – SPSM@CCS*. ACM, 67–78.
[24] Laurent Simon, Wenduan Xu, and Ross J. Anderson. 2016. Don't Interrupt Me While I Type: Inferring Text Entered Through Gesture Typing on Android Keyboards. *PoPETs* 2016 (2016), 136–154.
[25] Raphael Spreitzer. 2014. PIN Skimming: Exploiting the Ambient-Light Sensor in Mobile Devices. In *Security and Privacy in Smartphones & Mobile Devices – SPSM@CCS*. ACM, 51–62.
[26] Raphael Spreitzer and Benoît Gérard. 2014. Towards More Practical Time-Driven Cache Attacks. In *Information Security Theory and Practice – WISTP 2014 (LNCS)*, Vol. 8501. Springer, 24–39.
[27] Raphael Spreitzer, Simone Griesmayr, Thomas Korak, and Stefan Mangard. 2016. Exploiting Data-Usage Statistics for Website Fingerprinting Attacks on Android. In *Security and Privacy in Wireless and Mobile Networks – WISEC 2016*. ACM, 49–60.
[28] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. 2018. ProcHarvester: Fully Automated Analysis of Procfs Side-Channel Leaks on Android. In *Asia Conference on Computer and Communications Security – AsiaCCS 2018*. ACM. In press.
[29] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. 2018. Systematic Classification of Side-Channel Attacks: A Case Study for Mobile Devices. *IEEE Communications Surveys and Tutorials* 20 (2018), 465–488.
[30] Raphael Spreitzer and Thomas Plos. 2013. Cache-Access Pattern Attack on Disaligned AES T-Tables. In *Constructive Side-Channel Analysis and Secure Design – COSADE 2013 (LNCS)*, Vol. 7864. Springer, 200–214.
[31] Raphael Spreitzer and Thomas Plos. 2013. On the Applicability of Time-Driven Cache Attacks on Mobile Devices. In *Network and System Security – NSS 2013 (LNCS)*, Vol. 7873. Springer, 656–662.
[32] Pepe Vila and Boris Köpf. 2017. Loophole: Timing Attacks on Shared Event Loops in Chrome. In *USENIX Security Symposium 2017*. USENIX Association, 849–864.
[33] Zhi Xu, Kun Bai, and Sencun Zhu. 2012. TapLogger: Inferring User Inputs on Smartphone Touchscreens Using On-Board Motion Sensors. In *Security and Privacy in Wireless and Mobile Networks – WISEC 2012*. ACM, 113–124.
[34] Nan Zhang, Kan Yuan, Muhammad Naveed, Xiao-yong Zhou, and XiaoFeng Wang. 2015. Leave Me Alone: App-Level Protection against Runtime Information Gathering on Android. In *IEEE Symposium on Security and Privacy – S&P 2015*. IEEE Computer Society, 915–930.
[35] Xiaokuan Zhang, Xueqiang Wang, Xiaolong Bai, Yinqian Zhang, and XiaoFeng Wang. 2018. OS-level Side Channels without Procfs: Exploring Cross-App Information Leakage on iOS. In *Network and Distributed System Security Symposium – NDSS 2018*. The Internet Society.
[36] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. 2016. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In *Conference on Computer and Communications Security – CCS 2016*. ACM, 858–870.
[37] Xiao-yong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A. Gunter, and Klara Nahrstedt. 2013. Identity, Location, Disease and More: Inferring Your Secrets From Android Public Resources. In *Conference on Computer and Communications Security – CCS 2013*. ACM, 1017–1028.

A CONSIDERED WEBSITES, APPS, AND POIS

Tables 9, 10, and 11 show the websites, the POIs, and the applications used for the evaluations in this paper.

Table 9: Websites used for website fingerprinting evaluation.

http://www.360.cn	http://www.netflix.com
http://www.amazon.com	http://www.qq.com
http://www.baidu.com	http://www.reddit.com
http://www.facebook.com	http://www.sina.com.cn
http://www.google.com	http://www.sohu.com
http://www.imgur.com	http://www.tmall.com
http://www.instagram.com	http://www.vk.com
http://www.jd.com	http://www.wikipedia.org
http://www.linkedin.com	http://www.yahoo.com
http://www.live.com	http://www.yandex.ru

Table 10: POIs used for Google Maps evaluation.

Acropolis of Athens	Petronas Towers
Big Ben	Pyeongchang
Burj Khalifa	Pyongyang
Cape Town	Pyramids of Giza
Colosseum Rome	Singapore
Eiffel Tower	Sydney Harbour
Empire State Building	Taipei 101
Mirabell Gardens	The Great Wall of China
Mt. Everest	Toronto Canada
Peking	Wencelas Square Prague

Table 11: Applications used for app start evaluation.

com.airbnb.android
com.duckduckgo.mobile.android
com.fsck.k9
com.instagram.android
com.isis_papyrus.raiffeisen_pay_eyewdg
com.moshbit.studo
com.paypal.android.p2pmobile
com.tripadvisor.tripadvisor
com.twitter.android
com.waze
com.whatsapp
de.mcdonalds.mcdonaldsinfoapp
de.pilot.newyorker.android
de.prosiebensat1digital.prosieben
de.spiegel.android.app.spon
de.zalando.mobile
org.chromium.chrome
org.indywidualni.fblite
org.mozilla.firefox
org.zwanoo.android.speedtest

B MANUALLY TRIGGERED EVENTS

Table 12 depicts the verified information leaks of manually launched websites. The evaluation has been performed with the same set of 20 websites as listed in Table 9. The identified information leaks are the same as the ones revealed during the automatic evaluation in Section 4.1.

Table 12: Information leaks for website fingerprinting of manually triggered events on Android 8.1.

API(s)	Accuracy
android.net.TrafficStats.getTotalTxPackets()	51.9 %
android.net.TrafficStats.getTotalTxBytes()	46.9 %
android.net.TrafficStats.getTotalRxBytes()	46.9 %
android.net.TrafficStats.getMobileTxBytes()	45.6 %
android.net.TrafficStats.getMobileRxPackets()	45.0 %
android.net.TrafficStats.getMobileRxBytes()	45.0 %
android.net.TrafficStats.getMobileTxPackets()	43.1 %
android.net.TrafficStats.getTotalRxPackets()	41.2 %
java.io.File.getUsableSpace()	18.8 %
java.io.File.getFreeSpace()	16.2 %
android.app.usage.StorageStatsManager. getFreeBytes(java.util.UUID)	16.2 %
android.os.storage.StorageManager. getAllocatableBytes(java.util.UUID)	13.8 %
android.os.Process.getElapsedCpuTime()	13.8 %
API(s) requiring PACKAGE_USAGE_STATS (system-level)	
android.app.usage.NetworkStats.Bucket.getTxPackets()	46.9 %
android.app.usage.NetworkStats.Bucket.getRxPackets()	46.9 %
android.app.usage.NetworkStats.Bucket.getTxBytes()	46.2 %
android.app.usage.NetworkStats.Bucket.getRxBytes()	41.2 %
API(s) requiring READ_PHONE_STATE (dangerous permission)	
android.telephony.TelephonyManager.getDataActivity()	7.5 %