Accurate Specification for Robust Detection of Malicious Behavior in Mobile Environments

Sufatrio(✉), Tong-Wei Chua, Darell J.J. Tan, and Vrizlynn L.L. Thing

Institute for Infocomm Research, 1 Fusionopolis Way, #21-01, Connexis, Singapore
{sufatrio,twchua,jjdtan,vriz}@i2r.a-star.edu.sg

Abstract. The need to accurately specify and detect malicious behavior is widely known. This paper presents a novel and convenient way of accurately specifying malicious behavior in mobile environments by taking Android as a representative platform of analysis and implementation. Our specification takes a sequence-based approach in declaratively formulating a malicious action, whereby any two consecutive security-sensitive operations are connected by either a control or taint flow. It also captures the invocation context of an operation within an app's component type and lifecycle/callback method. Additionally, exclusion of operations that are invoked from UI-related callback methods can be specified to indicate an action's stealthy execution portions. We show how the specification is sufficiently expressive to describe malicious patterns that are commonly exhibited by mobile malware. To show the usefulness of the specification, and to demonstrate that it can derive stable and distinctive patterns of existing Android malware, we develop a static analyzer that can automatically check an app for numerous security-sensitive actions written using the specification. Given a target app's uncovered behavior, the analyzer associates it with a collection of known malware families. Experiments show that our obfuscation-resistant analyzer can associate malware samples with their correct family with an accuracy of 97.2 %, while retaining the ability to differentiate benign apps from the profiled malware families with an accuracy of 97.6 %. These results positively show how the specification can lend to robust mobile malware detection.

Keywords: Behavior specification · Mobile security · Malware detection

1 Introduction

Recent years have seen smart mobile devices becoming increasingly pervasive in our world. The threat posed by malicious mobile applications (apps), however, seriously undermines the security and privacy of mobile users [16], who are usually not even aware of any incidents occurring on their own devices. To deal with this, a mechanism that can accurately specify malicious behavior of mobile malware is important and necessary. Using such a specification, malware detectors can subsequently be built to help ascertain the presence of malicious apps.

© Springer International Publishing Switzerland 2015
G. Pernul et al. (Eds.): ESORICS 2015, Part II, LNCS 9327, pp. 355–375, 2015.
DOI: 10.1007/978-3-319-24177-7_18

This paper presents a novel way of accurately specifying malicious behavior in mobile environments. The specification is concise, convenient to write, and sufficiently expressive to capture a wide range of malicious behavior patterns that are commonly exhibited by mobile malware. Our specification declaratively expresses a malicious behavioral action as a sequence of security-sensitive operations, where any two consecutive operations are connected by either a control or taint flow. It also captures the invocation context of an operation, including the one that intercepts a broadcast-based system event, within its Android app-component type and lifecycle/callback method. Additionally, exclusion of operations that are invoked from UI-related callback methods can be specified on selective parts of a malicious action to indicate the absence of user involvement. We show how our specification is at least as expressive as existing specification schemes in describing malicious behavior in mobile environments.

To demonstrate the usefulness of our specification, we use it to compile an initial list of malicious and security-relevant behavior patterns in Android, which serves as a representative platform of our analysis and implementation. We then develop a static analyzer to utilize the pattern base and characterize apps in terms of their applicable pattern entries. Based on the uncovered entries of target apps, the analyzer associates the apps with a set of known malware families. Our goal here is to empirically demonstrate that the specification can facilitate a compilation of malicious pattern base, which can be used by an analyzer to derive stable, distinctive and obfuscation-resistant behavior patterns of existing malware families. Experiments show that the analyzer can associate malware samples with their correct family with an accuracy of 97.2 %. When tested on presumably-benign top free apps from Google Play, it can differentiate the apps from profiled malware with an accuracy of 97.6 %. App similarity techniques [7, 19, 20] can additionally be employed for a higher combined association accuracy.

In summary, our work makes the following contributions to mobile security:

– We propose a novel malware specification language, which can handily capture a wide range of malicious behavior in mobile environments (see Sect. 2).
– We analyze and compare the scheme's expressiveness and usage convenience with other existing specification techniques (see Sect. 3).
– We build a static analyzer that utilizes a database of malicious and security-sensitive patterns, which are declaratively written using the specification, to characterize an app and correlate it with known malware families (see Sect. 4).
– Using a set of experiments, we demonstrate how the analyzer can perform an association of malware samples with their correct family with a high accuracy of 97.2 % (see Sect. 5). Benign apps are deemed different from the profiled malware families with an accuracy of 97.6 %. We additionally show how the analyzer is robust against various code obfuscation attacks, which significantly reduce the average detection rate of 55 other anti-malware systems connected to VirusTotal from 70.8 % to 34.4 %.

The remainder of this paper is organized as follows. Section 2 elaborates our specification scheme. Section 3 analyzes and compares it with other schemes. Section 4 explains the design and implementation of our static analyzer, while

Sect. 5 reports its experiments. Section 6 gives additional discussions on our specification and analyzer. Section 7 mentions related work and Sect. 8 concludes this paper.

2 Malicious Behavior Specification for Mobile Environments

2.1 Goals, Rules and Notation of Specification Scheme

In Android, operations that access protected resources are invoked through permission-guarded API calls. To accomplish a certain security-sensitive action, multiple security-sensitive operations may be required. For instance, to record audio, an app needs to successively invoke the following methods of an object of `android.media.MediaRecorder` class: `setAudioSource()`, `setOutputFormat()`, `setOutputFile()`, and `start()`. Throughout its execution, an untrusted Android app may execute a number of possibly independent security-sensitive actions.

In this work, we call a sequence of API calls that can independently realize a security-sensitive action as a *malicious behavior pattern*. Our proposed specification scheme, called *Sequence-based Malicious Behavior Specification (SeqMalSpec)*, declaratively specifies malicious behavior patterns that are commonly or potentially exhibited by mobile malware in an accurate and convenient manner. Although we specifically target the Android platform[1], our specification scheme in principle can be easily adapted to other systems employing permission-guarded API calls for accessing protected resources.

We specifically take a sequence-based specification approach in order to yield a semantically-aware scheme that is both convenient for formulation and interpretation by human analysts, and is amenable to processing by automated analyzers. The specification intentionally avoids referring to any user-supplied identifiers so that it is robust against identifier renaming attacks [14]. App behavior characterization using *SeqMalSpec* is defined in a top-down fashion as follows.

- *A malicious app* (*maliciousApp*) is defined in terms of a set of its applicable malicious behavior patterns.
- *A malicious behavior pattern* (*maliciousPattern*) represents a path (sequence) of security sensitive operations, where any two consecutive operations in the path are connected by either:
  - $\rightsquigarrow$: a control-flow based sub-path of length ≥ 1, which may contain non security-sensitive operations in between;
  - $\rightarrow$: a control-flow based sub-path of length 1;
  - $\approx\!\rhd$: a taint-flow based sub-path of length ≥ 1, which may contain non security-sensitive operations;
  - $\Rightarrow$: a taint-flow based sub-path of length 1.

[1] We limit the scope of our behavior specification in this paper to operations at the Java/Dalvik code level. Operations that are performed by native code are thus beyond the scope of this paper, and may be addressed by future work.

As can be seen above, our specification allows multiple occurrences of a taint-flow based sub-path. Based on analyzing the attack threat in Android, we however observe that taint-flow related behavior of Android malware is mostly pertinent to private information leakage. Hence, in practice, only one taint-flow based sub-path is present at the end of a malicious pattern. It links up a private-information access operation with the corresponding exfiltration operation. Notice that this taint-flow sub-path, however, can be part of a longer pattern containing multiple preceding control-flow based sub-paths.

- *A security-sensitive operation* (*sensitiveOp*) is defined as a tuple $\langle x, y, z \rangle$, with:
  x: a non-empty element of the set of all possible combinations of Android app-component types from which operation z is invoked. We can thus write $x \in \mathcal{P}(\mathbb{X}) - \{\varnothing\}$, where $\mathbb{X} = \{activity, service, broadcastReceiver\}$. When $x = \mathbb{X}$, we can write a notational shorthand "*" instead.
  y: a non-empty element of the set of all possible combinations of lifecycle and callback methods. That is, $y \in \mathcal{P}(\mathbb{Y}) - \{\varnothing\}$, where $\mathbb{Y}$ is the set of all lifecycle and callback methods. When $y = \mathbb{Y}$, we can write "*". For convenience, we can also specify the set of lifecycle or callback methods m that should not be present as $!m$. In other words, $y = \mathbb{Y} - m$. This shorthand is useful, for instance, to exclude API invocations from a particular set of methods, such as UI-related callback methods.
  z: either an API call (*APICall*), a similar API call set (*similarAPICallSet*), or a system-event interception (*eventInterception*) operation.
- *An API call* (*APICall*) is defined based on its class type, method signature, and possible argument values to match. It is expressed as the following tuple:

$$APICall = \langle className, returnType, APICallName$$
$$(parameterType_1 = value_1, \ldots, parameterType_n = value_n) \rangle.$$

  For $value_i$, where $1 \leq i \leq n$, we can specify a special generic value "*any*" if the corresponding parameter does not need to be matched. An example of a security-sensitive operation of *APICall* type is $\langle \{service, broadcastReceiver\}, *, \langle$`android.telephony.TelephonyManager`, `java.lang.String`, `getDevice-Id()`$\rangle\rangle$. This corresponds to an invocation of `getDeviceId()` from any lifecycle method of a service or broadcast receiver, which runs in the background.
- *A similar API-call set* (*similarAPICallSet*) is a set of API calls sharing the same functionality, or an API call that has different argument signatures. Multiple API calls can have the same functionality in Android, for instance, when a new API call is used to replace deprecated one(s). An API can have different argument signatures when it is overloaded with different arguments.
- *A system-event interception operation* (*eventInterception*) is defined for each broadcast intent that is related to a system event, such as for `android.provider.Telephony.SMS_RECEIVED` intent. Since such an event interception occurs within the `onReceive()` method of a broadcast receiver, the tuple of a system-event interception operation is set with $x = \{broadcastReceiver\}$ and $y = \{$`onReceive()`$\}$. In an analyzed app, the presence of a system-event interception operation is assumed whenever:

1. There exists a broadcast receiver that is registered, either statically in `AndroidManifest.xml` or dynamically by invoking `registerReceiver()`, to receive the corresponding system intent in its intent filters.
2. The `onReceive()` lifecycle method is present (i.e. overridden) within that broadcast receiver.

A static analyzer that analyzes an app for malicious behavior patterns, such as ours described in Sect. 4, may assume these system-event interception operations at the beginning of the pertinent `onReceive()` methods.

– *A method-exclusion constraint* (*methodExclusionConstraint*) can be defined on a control- or taint-flow based sub-path of length ≥ 2 by specifying a set of methods to be excluded along the sub-path. That is, along the sub-path, the constraint disallows the presence of *any* operations that are invoked from within any methods in the set.[2] While we can specify any methods to be excluded in a sub-path, in practice we are concerned only with UI callback methods, such as `onClick()`, `onLongClick()` or `onKey()`. By specifying a set of all UI callback methods, referred to as *UICallbackSet*, on a sub-path, we thus require the sub-path to consist of operations that are performed without any user interactions. Notationally, we can write a method-exclusion constraint c with its excluded method set m by putting $!m$ on top of the control-flow based sub-path (i.e. $\overset{!m}{\rightsquigarrow}$) or taint-flow based sub-path (i.e. $\overset{!m}{\approx\!\rhd}$).

SeqMalSpec can be described in the Extended Backus-Naur Form (EBNF) notation as shown in Table 1 of Appendix A.

2.2 Sample Specified Malicious Patterns

The following are two commonly-exhibited malicious behavior patterns in Android environment that are expressed using *SeqMalSpec*. For easier reading, we omit the parameters of some API calls (denoted as "...") in these patterns:

– An automatic opening of the camera that is followed by the trigger of an image capture within the `onReceive()` method of a broadcast receiver, without any user interaction in between:
$\langle\{\mathit{broadcastReceiver}\}, \{\texttt{onReceive}()\}, \langle\texttt{android.hardware.Camera}, \texttt{android.hardware.Camera}, \texttt{open}()\rangle\rangle \overset{!\mathit{UICallbackSet}}{\rightsquigarrow} \langle\{\mathit{broadcastReceiver}\},\{\texttt{onReceive}()\}, \langle\texttt{android.hardware.Camera}, \texttt{void}, \texttt{takePicture}(\ldots)\rangle\rangle$.
– A sending of the phone's IMEI number to the Internet upon receipt of an SMS without any user interaction, which represents a behavior pattern of GoldDream malware that is previously specified using predicates in [8]:
$\langle\{\mathit{broadcastReceiver}\}, \{\texttt{onReceive}()\}, \texttt{SMS_RECEIVED_INTERCEPTION}()\rangle$

[2] If desired, one can define variations of exclusion constraint depending on which part of a sub-path that must satisfy the exclusion. That is, we may have $!\mathit{prefix}(n, m)$ and $!\mathit{suffix}(n, m)$, which disallow operations that are invoked from methods in set m within the first and last n operations of the sub-path, respectively. Our constraint that disallows all operations throughout a sub-path can be renamed as $!\mathit{all}(m)$.

$\overset{!UICallbackSet}{\rightsquigarrow}$ $\langle\{broadcastReceiver\}, \{$`onReceive()`$\}, \langle$`android.content.Context`, `android.content.ComponentName`, `startService(...)`$\rangle\rangle$ $\overset{!UICallbackSet}{\rightsquigarrow}$ $\langle\{service\}, \{$`onStartCommand()`$\}, \langle$`android.telephony.TelephonyManager`, `java.lang.String`, `getDeviceId()`$\rangle\rangle$ $\overset{!UICallbackSet}{\approx\!\!\rhd}$ $\langle\{service\}, \{$`onStartCommand()`, $\langle$`org.apache.http.client.HttpClient`, `org.apache.http.HttpResponse`, `execute(...)`$\rangle\rangle$.

3 Expressiveness of *SeqMalSpec* and Its Comparison

3.1 Expressiveness of *SeqMalSpec*

We give an analysis of the expressiveness of *SeqMalSpec* by asserting the following two claims, whose (sketch of) proof is given in Appendix B.

Claim 1. In a system where accesses to protected resources are invoked through a finite set of well-defined API calls, *SeqMalSpec* is able to express the following types of malicious action[3]:

1. A finite series of API calls that realizes an action to a protected resource;
2. A finite series of API calls that obtains a piece of information from a protected resource and subsequently performs other operations on it, including ultimately releasing it out of the system via a communication channel.

Defining a malicious action in terms of the Android-level API calls as in our specification allows us to express a more accurate semantic description than that based on the OS-level API/system calls in the traditional desktop environment. This is because the Android-level API calls are defined with more relevant operational semantics, which are directly pertinent to the protected resources on a mobile device and their access permission models. As a result, our specification can yield a more accurate and clearer behavior specification of Android malware compared to schemes that operate on the OS-level API calls.

Claim 2. Suppose we have an event-driven system, where each user interaction with an app raises a UI event. For each raised UI event, the system invokes a registered UI callback method, which is either an overridden correspondingly-named method of a registered event-handler object, or other arbitrarily-named handler method that is registered to process the event. On such a system model, on which Android is based, *SeqMalSpec* is able to express variants of malicious action described in Claim 1, whose any two API calls are executed through a series of operations that involve no user interaction.

[3] We remark that the use of Java reflection, together with string encryption, in Android may hinder static malware detectors in determining an invoked API call. As such, they may not be able to match a pattern whose series of API calls are explicitly named, thus apparently limiting the use of the specification. This is, in fact, a widely known limitation of static analyzers. To deal with it, one can incorporate a dynamic analyzer to uncover the invoked API calls. A static analyzer with such a runtime information feedback then would be able to inspect the app and match the pattern.

With regard to Claim 2, we would like to make the following important remarks. A pattern in *SeqMalSpec* whose all API calls are invoked from non UI-related callback methods, and also specifies UI-related method exclusion constraint, means that no user interaction should appear along the pattern. This is used to specify stealthy actions. An execution path *involving* a user interaction, however, does not necessarily mean that the action is *intended* or *consented* by the user. This is because a malware may perform malicious actions while the user is legitimately interacting with its activities. This subtle point highlights that a flow connector with an added UI-related exclusion constraint is a stricter version of its unconstrained one. It is to be specified when we know that a particular malware sample performs the pertinent patterns in a totally stealthy manner.

3.2 Comparison with Other Malware Specification Schemes

We now compare *SeqMalSpec* with other existing malware specification schemes. There exist various ways of specifying malicious behavior. Most of them [2,4–6,9,12,13], however, pre-date modern mobile OSes and are designed primarily for desktop security. As such, they work mostly at the native code level, where the higher-level operational semantics at the mobile OS level cannot be fully utilized. Below, we compare *SeqMalSpec* with other schemes that are specifically proposed for mobile setting with respect to expressiveness power and usage convenience.

Predicate Based Specification. Feng et al. [8] recently proposed Apposcopy, which specifies the signatures of Android malware in Horn-clause based Datalog language. For this purpose, a number of unary and binary predicates are introduced. A malicious pattern is considered present in an app if all its specified predicates evaluate to true, possibly through a unification process. While Datalog-based predicates are suitable to identify relations, usually between two API operations, our sequence-based specification allows us to naturally express a chain of *any number* consecutive operations, together with the context of each operation invocation. As a result, we can easily specify multiple context-based operations that must appear in order, including affixing possible UI-exclusion constraints on selective parts of a sequence. Since Apposcopy can define new predicates, it can extend its specification to mimic our newly-proposed invocation context and constraints. Yet *SeqMalSpec*, in our view, look more natural to human analysts since a pattern's operations are expressed using the original Android API calls rather than newly-defined predicate-based expressions. Section 7 additionally mentions further differences between our work and [8] with respect to the signature derivation and static analyzer implementation.

Temporal-Logic Based Specification. Model checking systems use a behavior signature expressed as a temporal logic formula. This formula can be based on Computational Tree Logic (CTL) or Linear Temporal Logic (LTL); or their extensions, such as CTPL [12] or SCTPL/SLTPL [15]. While previous model-checking based detectors work at the native code level [12] or on a generic platform [2], a recent work [15] applies model checking to Android apps.

Although a temporal logic formula can describe various temporal-based correlation of events, its usage in specifying malicious behavior, including the one in [12,15], is typically limited to describing the existence of a sequence of related operations. Consequently, the relevant formulae employ linear-time temporal operators **F** (*finally/eventually*) and **U** (*until*); or appear as a CTL-based formula in the form of $\mathbf{EF}(\phi_1, \mathbf{EF}(\phi_2))$ or $\mathbf{E}(\phi_1 \, \mathbf{U} \, \phi_2)$. As reasoned above, *SeqMalSpec* is able to express such formulae using a more intuitive notation. While the extended temporal logic used in [15] can deal with variables to identify the reading of a private information and its subsequent exfiltration, we instead use the notion of taint-flow relationship between a set of source and sink API calls as in [1]. The use of API call sequence in *SeqMalSpec* additionally allows us to selectively encode the context of an API call invocation (i.e. using *sensitiveOp*) and to impose the exclusion of UI-related operations (i.e. using *methodExclusionConstraint*), which are both lacking in the existing logic-based specifications.

4 Static Analyzer Utilizing *SeqMalSpec*

4.1 Goal and Approch

To demonstrate the usefulness of *SeqMalSpec* and how one can leverage on it, we have developed a static analyzer that uncovers the presence of behavior patterns within Android apps by taking a list of *SeqMalSpec*-based specifications as an input. Unlike the static analyzer in [8], which determines if an app exhibits the behavior patterns of a particular malware family that are manually-specified by human experts, we instead devise our analyzer to automatically derive behavior patterns of each existing malware family. To this end, using *SeqMalSpec*, we compile a list of security-sensitive behavior patterns that are commonly exhibited by Android malware. We also include other behaviors that are potentially relevant from the security analysis viewpoint, such as inter-component activation operations. Given this compiled pattern database, our static analyzer inspects an app and reports the presence/absence of each pattern entry in the database.

Following this app characterization, the analyzer then associates an app with a set of known malware families by reporting the app's similarity distance to

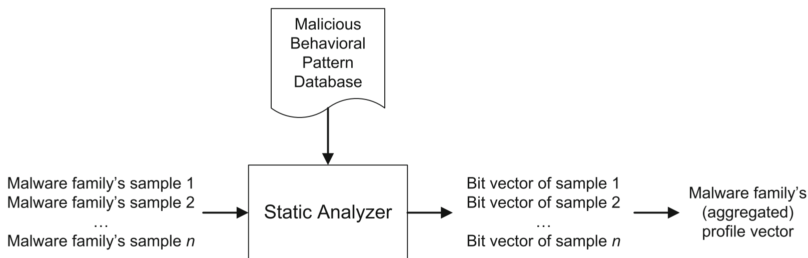

Fig. 1. Profiling a malware family for its malicious behavior pattern profile.

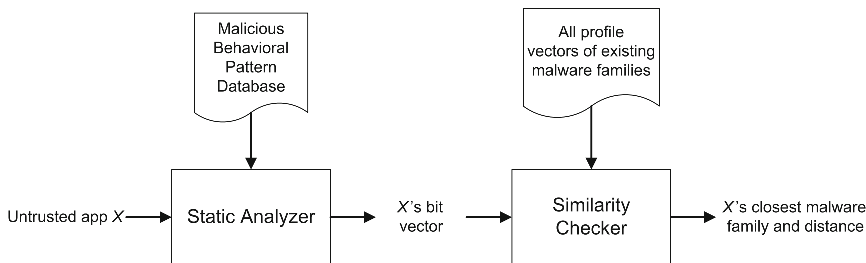

Fig. 2. Analysis of an untrusted app to determine its closest existing malware family.

the closest profiled malware family. This association is carried out to empirically show that the proposed specification, which is our main contribution in this work, allows for a derivation of stable, distinctive and obfuscation-resistant behavior profile of existing malware families. Figure 1 illustrates the process workflow of profiling an existing malware family. Figure 2 depicts how an association of an untrusted app is performed. The details of all these steps are elaborated below.

4.2 System Design and Implementation

Compilation of Malicious Pattern Database. We compile a behavior pattern database by examining how Android malware can launch various attack modalities on a device. For this, we analyze numerous existing security advisories on existing Android malware, as well as examine Android permissions to see how they can possibly be abused by apps. Our approach to identifying these patterns is thus a human-determined one. We take this approach since we specifically want the patterns to be accurate, accountable and explainable.

While this compilation effort requires the enumeration of all potentially relevant sensitive operations, we argue that producing a relatively comprehensive pattern database for Android is feasible owing to the following reasons:

- Android permissions, which guard a device's protected resources, are limited.
- Apps invoke a known set of API calls to access these permission-guarded resources. While malware writers may craft their samples to perform various processing steps, including for obfuscation purposes, the API calls represent a well-guarded gateway to performing the samples' payloads.

We remark that once such a pattern database is compiled, it can be shared with the security community for a crowd-sourcing based extension or refinement.[4]

The database used in our experiments includes patterns that perform the following types of operations: system-event interception, broadcast-intent related processing (e.g. `android.content.BroadcastReceiver:abortBroadcast()`),

[4] Correspondingly, we do not assume that the uncovered patterns of a malware family in our experimentation give a complete specification of the family. This is because the completeness level depends on the employed pattern database.

incoming message processing (e.g. `android.os.Bundle:get("pdus")`), app component activation, audio/video/camera processing, access of private information, information release through SMS and data network, network management operations (e.g. reconnecting a WiFi network), and alarm-related operations.

Detection of Behavior Patterns. To uncover the presence of entries of the compiled behavior pattern database, we leverage on FlowDroid [1], which is built on top of the Soot framework. FlowDroid is a precise static analysis for Android apps, which finds potential privacy leaks between a list of source and sink API calls. We use FlowDroid to obtain the callgraph and all intra-procedural graphs of an Android app, as well as to perform a taint-flow analysis between a given source and sink method within a behavior pattern.

Note that by making use of our expressive behavior specification, which covers both control- and data-flow aspects of an app, our analyzer can characterize malware behavior more precisely than FlowDroid. In addition, we also made the following enhancements to FlowDroid in order to detect the compiled patterns:

- Identification of system-event interception operations of an app by scanning both its statically- and dynamically-registered broadcast receivers. All identified dynamic broadcast receivers are added as the app's entry points.
- Control-flow based, i.e. $\rightsquigarrow$ sub-path, reachability analysis of a pattern entry.
- Utilization of control-flow based reachability to filter out any source and sink pairs that are known to be unconnected. This avoids extra taint-flow checking by FlowDroid, which is computationally more expensive.
- Argument value determination of a number of parameterized API calls, possibly through a number of preceding intermediary assignment statements.

Our current prototype does not impose any method-exclusion constraints. Yet, it can be extended to apply the constraints as discussed in Sect. 6.

Profiling of Existing Malware Families. As can be seen in Fig. 1, we profile a malware family by having the static analyzer check all the samples within the family. For each sample, we generate its bit vector v of length ℓ, where ℓ is the number of pattern entries in the database. An entry at index i, i.e. $v[i]$, is set to 1 if the i-th pattern is present in the analyzed sample; or it is set to 0 otherwise.

Once we produce the bit vectors for all the samples of a malware family, we can derive the profile vector for that family, whose entries are real numbers between 0 and 1 (inclusive), as follows. Let us denote k as the number of samples in the family; and v_i as the bit vector of the i-th sample in the family, with $1 \leq i \leq k$. The malware family's profile vector p is derived by setting its entry at index j, for all $1 \leq j \leq \ell$, as follows:

$$p[j] = \frac{1}{k} \cdot \sum_{i=1}^{k} v_i[j]. \quad (1)$$

An entry $p[j]$ thus quantifies the *presence rate* of a malicious behavior pattern j across all the samples within the malware family.[5]

Association of Apps with Profiled Malware. Figure 2 shows how an association of an untrusted app is performed by comparing its bit vector against the profile vectors of all known malware families. Its detailed steps are as follows.

First, we want to associate each behavior pattern entry with a weight that indicates its usage prevalence for solely malicious purposes. The more a pattern is used more exclusively by malicious apps, the higher its weight is to be set. To achieve this, we take an automated weight-generation approach to determine b_i, with $0 \le b_i \le 1$, as the occurrence rate of pattern i among (presumably) benign apps. Then, we can derive a vector w of length ℓ, where $w[i] = 1 - b_i$, for $1 \le i \le \ell$.

Let us now denote f as the number of all known malware families. The profile vectors of all malware families can be considered as a real-valued matrix M of dimension $f \times \ell$. A cell entry $M[i][j]$ represents the index j (with $1 \le j \le \ell$) of the profile vector belonging to malware family i (with $1 \le i \le f$).

We calculate the *weighted Euclidean distance* between the input app (with its bit vector a) and malware family i (with its profile vector $M[i]$) as follows:

$$distance_{ai} = \sqrt{\sum_{j=1}^{\ell} w[j] \cdot (a[j] - M[i][j])^2} \ . \tag{2}$$

Once we have calculated the target app's distance scores against all malware families, which form a multiset $\{distance_{ai},\ 1 \le i \le f\}$, we can determine the set of the closest malware families for the app, called $closestFamilySet_a$, as follows:

$$closestFamilySet_a = \{x, 1 \le x \le f \mid \forall y, 1 \le y \le f : distance_{ax} \le distance_{ay}\}. \tag{3}$$

Note that while we define $closestFamilySet_a$ as a set, which may have multiple elements that all share a common similarity score, we however expect it to be a singleton, i.e. $|closestFamilySet_a| = 1$. In the case where $distance_{ax} > \tau$, with τ serving as a distance threshold, we then view the app to be sufficiently different from all the profiled malware families. Note, however, that we only compare apps and known malware solely based on their exhibited malicious patterns. Our similarity checking thus can be complemented by other app similarity techniques [7,11,19,20], which analyze different app modalities, to further ascertain if an app is really similar to a known malware family. Section 7 further discusses this point.

We build our analyzer module that generates the profiles of existing malware as in (1) and associates app with the profiled malware as in (2–3) in Python.

[5] In the case where a malware family actually consist of a few sub-families with significantly different behavior (see our empirical findings in Sect. 5.1), we may thus want to first perform a clustering on the family to partition it into several sub-families. Hence, each sub-family will have its own more accurate profile vector. The similarity checking step is then done against the profile vectors of the formed sub-families.

5 Experimentation Results

This section reports the experimentation results of our analyzer with regards to its association results. In the following, we successively explain the used malware dataset, experimentation objectives, taken methodology, and obtained results.

5.1 Used Malware Dataset

We evaluate our static analyzer using real-world malware samples from the Android Malware Genome Project [21], which in its entirely consists of 1,260 malware samples from 49 families. The distribution of malware samples among the families are, however, unequal. There are families that contain very few samples. Since we need to evaluate the analyzer by dividing each family's samples into profiling and testing samples, we thus omit malware families that have only six or less samples. We also exclude BaseBridge and Asroot, which perform an update attack and a root exploit with no observable payload execution within its Java code, respectively [21]. The following 22 families constitute our experimental dataset: ADRD, AnserverBot, BeanBot, Bgserv, DroidDream, DroidDreamLight, DroidKungFu1, DroidKungFu2, DroidKungFu3, DroidKungFu4, Geinimi, GoldDream, Gone60, jSMSHider, KMin, Pjapps, Plankton, RogueSPPush, SndApps, YZHC, zHash and Zsone. Out of 1,083 total samples from these 22 families, 125 (11.5 %) samples apparently did not run to completion during the taint-flow analysis using FlowDroid (see additional discussion in Sect. 6). Hence, 958 samples, or 76 % of the total samples in the Android Malware Genome Project, form our analyzed malware samples.

When we characterized the listed families to build their profile vectors, we observed that some families seem to consist of different sub-families. From [21], we learn that among the 1,260 malware samples in the Android Malware Genome Project, 1,083 (86.0 %) of them are repackaged. Thus, while the samples under the same family share a common payload, they may actually stem from a few variants of repackaged apps. The carrier apps may have other additional operations, including those security-sensitive ones. We, however, cannot fully ascertain this inference since the information of the exact mechanisms used to classify the samples into families is unavailable to us. To capture the existence of sub-families, we thus performed a clustering on the bit vectors of malware samples within a family. Based on our experimentation with the employed parameterized clustering technique, we empirically formed a cluster for each five samples within a family.[6] We found that the performed clustering on the families gave well-partitioned sub-families, thus supporting our hypothesis of the existence of sub-families.

[6] Notice that, for our purpose of associating an app with a set of profiled malware sub-families, separating samples belonging to the same malware sub-family into two different clusters will not affect the association result. This is because the derivation of the set $closestFamilySet_a$ in (3) will yield either a single sub-family or multiple (possibly separated) sub-families with the same smallest distance score.

5.2 Experimentation Objectives and Obtained Results

We aim to evaluate our developed analyzer with the following three objectives:

1. To test the association of malware samples with their correct families;
2. To test the association of presumably benign apps with the malware families;
3. To test the robustness of the analyzer against code obfuscation.

Objective 1 (Associating Malware Samples into Correct Families). For each malware family in the dataset, we randomly select 80 % of the samples to derive the profile of the family, and leave the remaining 20 % to form the testing set. Since we form sub-families of malware, we compare each test sample against all sub-families, and report the family whose sub-family produces the smallest distance. We empirically set $\tau = 2.45$ as an approximate midpoint that separates the results of malicious samples (objective 1) and benign apps (objective 2). Using this threshold value, our analyzer can correctly associate the test samples with an accuracy of 97.18 %. The weighted Euclidean distances of the test samples range from 0.00 to 7.03, with an average distance of 0.64.

Objective 2 (Association of Benign Apps with Profiled Families). We also test if presumably benign apps listed as the top free apps on Google Play can be sufficiently similar to any of the profiled malware families. Analyzing 546 apps using the same threshold value gives an accuracy of 97.62 %. In other words, only 2.38 % of the tested benign apps is inaccurately determined to be similar to one of the profiled malware families. Upon inspection, we find that these apps are all inaccurately associated with Gone60 malware family. The generated profile vectors reveal that Gone60 exhibits only a few applicable patterns. A malware characterization in [21] lists Gone60 to perform only an SMS-based personal information stealing. This may explain why a number of benign apps can share similar patterns with Gone60. The weighted Euclidean distances of the tested benign apps range from 2.42 to 25.47, with an average distance of 7.03.

Objective 3 (Robustness Against Transformation Attacks). To show the robustness of our analyzer against malware transformation attacks, we compare the bit vectors of original malware samples with those of the transformed ones. If each vector pair always matches, that means our analyzer is resistant against the applied transformations. For this, we select 8 families from our dataset (i.e. BeanBot, Bgserv, DroidDream, Geinimi, GoldDream, Pjapps, Sndapps, Zsone), each with 4 random samples, for variant generation and detection. We use `apktool` to produce an app's disassembled smali code, and then modify the code to apply a sequence of transformations as listed in Table 2 of Appendix C.

The results show that our analyzer *always* produces the same bit vector for each transformed and original sample pair. The robustness of our analyzer stems from the following two important features of *SeqMalSpec*:

- Its avoidance of using any developer-supplied identifiers.
- The control- or taint-flow reachability property between two operations, which is robust against possible control-flow based obfuscation.

While the applied obfuscation methods are still limited, they are sufficient to deceive many anti-malware systems connected to VirusTotal (https://www.virustotal.com). Table 3 in Appendix D shows the results of transforming randomly-selected 5 malware families, with 2 samples in each family.

6 Discussions

6.1 Threats to Validity

We now address possible threats to the validity of our analyzer evaluation:

- *The used features of SeqMalSpec:* Our compiled pattern database exercises a simplified usage of *SeqMalSpec* in that only a single taint-flow connector ($\approx\!\rhd$) is present to link a source and sink API call. We can implement a more expressive usage of the taint-flow connector by allowing a successive occurrences of taint-flow connected operations, where: (i) its beginning and end API calls represent the source and sink operations, respectively; and (ii) the intermediary API calls represents the 'pass-through' operations along the taint flow. In our implementation, we however choose to see how the simplified scheme can work in profiling and associating malware samples.
- *Malicious behavior database compilation:* Our compiled pattern database might not be sufficiently comprehensive. In fact, generating a sufficiently complete database may require a collective and cumulative effort. We however believe that a sufficiently good database is feasible to be constructed, which can then be refined over time, preferably in a crowd-sourced manner.
- *The developed analyzer:* Our analyzer relies on FlowDroid to perform its taint-flow analysis. While FlowDroid represents a state-of-the-art tool in performing taint analysis for Android apps, we encountered some apps that took a rather long time (i.e. hours or even a few days) of taint analysis processing on our machine. A number of apps throwed exceptions, including the memory-insufficiency related ones. In addition, the FlowDroid's option to output multiple paths between a detected source and sink pair seems to be very time- and memory-consuming. Any extensible tools that can give the same or even higher level of analysis precision as FlowDroid's, but with lower processing and memory footprint, will thus be useful. Since our specification and app association technique are independent of any implementation platforms, they can be realized using other tools as they become available.
- *Testing methodology:* For the experimentation, there is always a concern of not having sufficient samples in the dataset. We have tested our analyzer against most malware samples in the Android Malware Genome Project as well as more than five hundreds widely-used top free apps from Google Play. Further testing with more samples, especially recent ones, however will always be good to be carried out. We also assume that the top free apps downloaded from Google Play are benign, which may not always be the case. As mentioned in Sect. 5.1, the observed need for partitioning malware families into their

sub-families may warrant manual inspection to ascertain the presence of sub-families in the dataset. Lastly, we set the distance threshold value τ as an approximate midpoint that gives almost equal distance separation on both the malicious samples (objective 1) and benign apps (objective 2). Deciding a more fitting threshold value warrants further investigation, and is ideally to be done on a large number of analyzed apps.
- *Known challenges to static analysis:* Lastly, we also mention the widely-known challenges that may hinder any static analysis systems, namely the use of native code and Java reflection. Our system currently does not deal with these challenges, which may be best handled by dynamic analysis or other security techniques.

6.2 Future Work

Our experiments show that our specification and analyzer can derive patterns that form a stable and distinctive profile of a malware family. Nonetheless, they are less useful in profiling malware families that perform update attacks or dynamic code loading, such as BaseBridge. They also cannot effectively characterize malware families that do not execute their malicious payloads at the Java-based Android code, such as Asroot. To deal with this issue, our association can be complemented by another round of similarity checking that examines app structure similarity. Our app association, however, are useful in establishing app similarity with respect to the compiled pattern base, with an added benefit of being able to report explainable and comprehensible uncovered patterns.

Other possible future work that can improve our prototype system include:

- Our current prototype does not implement the UI-method exclusion constraints yet. We can implement the defined $!m$ (i.e. $!all(m)$) constraint rather easily by removing all excluded methods in the callgraph of an analyzed app. To implement $!prefix(n, m)$ and $!suffix(n, m)$, however, we need to ensure that a constructed path must avoid using any operations in the excluded methods, either in the beginning or ending part of the path as desired.
- As mentioned earlier, we can implement an analyzer that detects a pattern with multiple occurrences of the taint-flow connector ($\approx\!\rhd$). For this, we need to ensure that a taint-flow must pass a number of intermediary operations.
- We can further measure the robustness of our prototype system against obfuscation attacks by applying and testing more app transformations.

7 Related Work

The comparison of *SeqMalSpec* with other existing mobile malware specification schemes is given in Sect. 3.2. Below, we highlight further differences with other specification work with regard to the associated detector implementation.

The design and implementation of our static analyzer differs from that in Apposcopy [8] in the two following aspects:

1. Apposcopy implements its own custom static analyzer, with a significant effort spent on developing its taint-flow tracker. In contrast, we leverage on FlowDroid [1], which is known to perform a highly precise static taint analysis for Android apps. We additionally perform a number of enhancements to FlowDroid as described in Sect. 4.2.
2. Apposcopy requires its authors to manually inspect malware samples and craft the signature for each malware family. Its experimentation was then carried out to check whether a set of existing malware samples and benign apps match all the predicates in the manually-crafted signatures. In contrast, we need to compile a generic pattern base only once, from which our analyzer then automatically profiles all existing malware families. Hence, our analyzer not only checks the existence of certain behavior patterns within target apps, but also profiles all existing malware families and then associates a sample with its correct family in an automated manner as reported in Sect. 5.

DroidMiner [17] generates a behavior graph in order to mine segments of the graph that might correspond to known suspicious behavior, which are called *modalities* in the work. While our specification makes use of declarative, human-formulated operation sequences to be searched on samples from the control- and taint-flow viewpoints, DroidMiner extracts graph-reduction based modalities to be further processed by a classifier or associated with the rule mining process. Due to this, our approach in specifying malicious behavior is thus more in line with how human analysts work in analyzing a malicious app.

RiskRanker [10] detects malware samples, including possible zero-day ones, that invoke known root exploit, illegal cost creation and privacy-violation exploit patterns. DroidRanger [22] analyzes apps based on their permission-based behavioral footprint. While the two systems describe and scan for behavior patterns, they however lack a generic declarative behavior model that can concisely specify behavior patterns, and is also robust against transformation attacks.

FlowDroid [1] is a highly precise static taint analysis tool for Android apps, which is context, flow, field and object sensitive. Our work extends FlowDroid, which implicitly detects only privacy-leakage operations involving a pair of source and sink, to deal with any general sequence-based operations. Our improved analyzer not only reports privacy leakages, but also analyzes and characterizes an untrusted app, and then associates it with a known malware family.

Pegasus [3] detects malicious behavior that violates the temporal properties of safe interactions between an app and the Android event system. It thus can detect, for instance, if an operation is invoked without the prerequisite GUI-based interaction that indicates the user's consent. Meanwhile, AppIntent [18] checks if a data transmission in an app is intended by the user. Similar to these two systems, our work considers operations that are invoked without user involvement. Our analyzer can implement a feature that looks for a sequence of operations, whose sub-path(s) exclude any operations from within UI-related callback methods. The presence of these patterns, which are declaratively specified, are used by our analyzer to characterize and classify a malware sample.

8 Conclusion

We have presented our sequence-based specification scheme called *SeqMalSpec*, which is concise, convenient and sufficiently expressive to capture malicious behavior in mobile environments. We have also demonstrated how *SeqMalSpec* can be utilized by a static analyzer to characterize apps in terms of their malicious behavior patterns. Experiments have shown that the analyzer can associate a malicious app with its correct malware family with a high accuracy of 97.2 %, while still being able to differentiate benign apps from the profiled malware families with an accuracy of 97.6 %. Lastly, we have also demonstrated the analyzer's robustness against various code obfuscation attacks. The proposed *SeqMalSpec*, as we foresee it, will thus open up various other effective approaches to mitigating malware in mobile environments, including the malware-plagued Android platform that commands a huge user base.

Appendix A: *SeqMalSpec* Scheme in Extended BNF

Malicious behavior specification using *SeqMalSpec* can be described in the EBNF notation as shown in Table 1. In the notation, we take the liberty of expressing the terminals belonging to a defined set using a natural language description (written within "[ ]") instead of explicitly listing all the set elements.

Appendix B: Sketch of Proof of *SeqMalSpec* Expressiveness

The sketch of proof for the two expressiveness claims in Sect. 3 is as follows.

Proof of Claim 1: Suppose there exists a malicious action that is inexpressible using *SeqMalSpec*. Due to the assumed system, where all protected resources are guarded by a set of well-defined API calls, the malicious action must manifest itself as a series, i.e. sequence, of security-sensitive API calls:

- If the action involves no taint flow: *SeqMalSpec* is able to express that action using all control-flow based connectors. This leads to a contradiction.
- If it involves a taint flow from one operation to the other(s): *SeqMalSpec* is also able to express that action using a combination of control- and taint-flow based connectors. This also leads to a contradiction.

Hence, *SeqMalSpec* is able to express the actions described in Claim 1. □

Proof of Claim 2: Suppose there is an action that involves no user interaction. We will show that *SeqMalSpec* is able to describe this action. Based on Claim 1, we know that the action is expressible with a sequence of API calls that are connected with the defined connectors. We can then add a non-UI method exclusion constraint on each of the used control- or taint-flow based connector. We additionally specify that all the security-sensitive API calls are not invoked from any UI-related callback methods. This means that *no* operation of the action is ever

Table 1. The notation of *SeqMalSpec* in Extended Backus-Naur Form (EBNF).

$maliciousApp ::= \{maliciousPattern+\}$,

$maliciousPattern ::= sensitiveOp \mid (sensitiveOp\ connector)+\ sensitiveOp$,

$connector ::= \rightarrow \mid \overset{!methodExclusionSet}{\rightsquigarrow} \mid \Rightarrow \mid \overset{!methodExclusionSet}{\approx\!\!\!\vartriangleright}$,

$methodExclusionSet ::= \{excludedMethod+\}$,

$excludedMethod$ ::= "[all context methods from which an invocation is to be excluded]",

$sensitiveOp ::=$

$\quad \langle appComponentTypeSet, invocationMethodSet, similarAPICallSet \rangle \mid$

$\quad \langle appComponentTypeSet, invocationMethodSet, APICall \rangle \mid$

$\quad \langle \{broadcastReceiver\}, \{\texttt{onReceive()}\}, eventInterception \rangle$,

$appComponentTypeSet ::= \{appComponentType+\}$,

$appComponentType$ ::= "*activity*" | "*service*" | "*broadcastReceiver*",

$invocationMethodSet ::= \{lifecycleOrUICallbackMethod+\}$,

$lifecycleOrUICallbackMethod ::= lifecycleMethod \mid UICallbackMethod$,

$lifecycleMethod$ ::= "[all lifecycle methods belonging to elements of $appComponentType$]",

$UICallbackMethod$ ::= "[all recognized UI callback methods]",

$similarAPICallSet ::= \{APICall+\}$,

$APICall ::= \langle className, returnType, APICallName(valuedParameters) \rangle$,

$className$ ::= "[all Android classes containing elements of $APICallName$]",

$returnType$ ::= "[all possible return types]",

$APICallName$ ::= "[all security-sensitive API calls]",

$valuedParameters ::= paramType = value \mid (paramType = value,)+\ paramType = value$,

$paramType$ ::= "[all possible parameter types]",

$value$ ::= "[all possible values to parameter types, including a special value called 'any']",

$eventInterception$ ::= "[all system-event interception operations]".

invoked from UI-related callback methods, which are triggered by the assumed event-driven system in the event of user interaction with the app. If the action runs automatically upon a broadcast system event, *SeqMalSpec* is also able to describe a system event interception in its pattern. This shows that *SeqMalSpec* is able to express the stealthy action. □

Appendix C: Applied App Obfuscation Attacks

Table 2 lists a sequence of app obfuscations that are applied to malware samples as discussed in Sect. 5.2.

Table 2. The sequence of app obfuscations that are applied to a malicious sample in order to generate its new variants.

Step	Transformation	Transformation details
1	Package name renaming	Replace a sample's package name, which can be found in its `AndroidManifest.xml`, with random English words found in the dictionary
2	Identifier renaming	Replace all method names and strings with random English words found in the dictionary. We however do not rename identifiers with a single and dual characters (e.g. 'a', 'b', 'aa', 'ab'), which could have been subject to previous obfuscation by ProGuard
3	Junk code insertion	Insert junk code following arithmetical-operation based opaque predicates
4	Control-flow obfuscation	Relocate the invocation points of sensitive Android API calls by using an indirect method invocation
5	Reassembling and repacking	Reassemble the transformed smali code into an APK file using `apktool`, and then sign the file with a new custom key

Appendix D: Evaluation Results of Obfuscation Attacks on Other Anti-Malware Systems

Table 3 shows the results of evaluating 55 anti-malware systems that are connected to VirusTotal (https://www.virustotal.com) as explained in Sect. 5.2.

Table 3. Detection comparison between the transformed and their original samples on 55 anti-malware systems connected to VirusTotal.

Malware family	Average detection rate		Detection rate reduction
	Original samples	Transformed samples	
BeanBot	66 %	35 %	31 %
Bgserv	70 %	35 %	35 %
GoldDream	73 %	45 %	28 %
Sndapps	70 %	27 %	43 %
Zsone	75 %	30 %	45 %
Average	70.8 %	34.4 %	36.4 %

References

1. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: 35th Conference on Programming Language Design and Implementation (2014)
2. Beaucamps, P., Gnaedig, I., Marion, J.-Y.: Abstraction-based malware analysis using rewriting and model checking. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 806–823. Springer, Heidelberg (2012)
3. Chen, K.Z., Johnson, N., D'Silva, V., Dai, S., MacNamara, K., Magrino, T., Wu, E., Rinard, M., Song, D.: Contextual policy enforcement in Android applications with permission event graphs. In: 20th Network and Distributed System Security Symposium (2013)
4. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: 12th USENIX Security Symposium (2003)
5. Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In: 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (2007)
6. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: 2005 IEEE Symposium on Security and Privacy (2005)
7. Crussell, J., Gibler, C., Chen, H.: Attack of the clones: Detecting cloned applications on Android markets. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 37–54. Springer, Heidelberg (2012)
8. Feng, Y., Anand, S., Dillig, I., Aiken, A.: Apposcopy: Semantics-based detection of Android malware through static analysis. In: 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (2014)
9. Fredrikson, M., Jha, S., Christodorescu, M., Sailer, R., Yan, X.: Synthesizing near-optimal malware specifications from suspicious behaviors. In: 31st IEEE Symposium on Security and Privacy (2010)
10. Grace, M., Zhou, Y., Zhang, Q., Zou, S., Jiang, X.: Riskranker: Scalable and accurate zero-day Android malware detection. In: 10th International Conference on Mobile Systems, Applications, and Services (2012)
11. Hanna, S., Huang, L., Wu, E., Li, S., Chen, C., Song, D.: Juxtapp: A scalable system for detecting code reuse among Android applications. In: Flegel, U., Markatos, E., Robertson, W. (eds.) DIMVA 2012. LNCS, vol. 7591, pp. 62–81. Springer, Heidelberg (2012)
12. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting malicious code by model checking. In: Julisch, K., Kruegel, C. (eds.) DIMVA 2005. LNCS, vol. 3548, pp. 174–187. Springer, Heidelberg (2005)
13. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic worm detection using structural information of executables. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 207–226. Springer, Heidelberg (2006)
14. Rastogi, V., Chen, Y., Jiang, X.: DroidChameleon: Evaluating Android anti-malware against transformation attacks. In: 8th ACM Symposium on Information, Computer and Communications Security (2013)
15. Song, F., Touili, T.: Model-checking for Android malware detection. In: Garrigue, J. (ed.) APLAS 2014. LNCS, vol. 8858, pp. 216–235. Springer, Heidelberg (2014)
16. Sufatrio, Tan, D.J.J., Chua, T.W., Thing, V.L.L.: Securing Android: a survey, taxonomy, and challenges. ACM Comput. Surv. **47**(4), 45 (2015). Article 58

17. Yang, C., Xu, Z., Gu, G., Yegneswaran, V., Porras, P.: DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in Android applications. In: Kutyłowski, M., Vaidya, J. (eds.) ESORICS 2014, Part I. LNCS, vol. 8712, pp. 163–182. Springer, Heidelberg (2014)
18. Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., Wang, X.S.: Appintent: Analyzing sensitive data transmission in Android for privacy leakage detection. In: 20th ACM Conference on Computer and Communications Security (2013)
19. Zhou, W., Zhou, Y., Grace, M., Jiang, X., Zou, S.: Fast, scalable detection of 'piggybacked' mobile applications. In: 3rd ACM Conference on Data and Application Security and Privacy (2013)
20. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party Android marketplaces. In: 2nd ACM Conference on Data and Application Security and Privacy (2012)
21. Zhou, Y., Jiang, X.: Dissecting Android malware: Characterization and evolution. In: 33rd IEEE Symposium on Security and Privacy (2012)
22. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In: 19th Network and Distributed System Security Symposium (2012)