View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UCL Discovery

A Family of Droids–Android Malware Detection via Behavioral Modeling: Static vs Dynamic Analysis

Lucky Onwuzurike†, Mario Almeida‡, Enrico Mariconti†, Jeremy Blackburn*,
Gianluca Stringhini†, and Emiliano De Cristofaro†
†University College London, ‡Polytechnic University of Catalonia, *University of Alabama at Birmingham

***Abstract*— Following the increasing popularity of the mobile ecosystem, cybercriminals have increasingly targeted mobile ecosystems, designing and distributing malicious apps that steal information or cause harm to the device's owner. Aiming to counter them, detection techniques based on either static or dynamic analysis that model Android malware, have been proposed. While the pros and cons of these analysis techniques are known, they are usually compared in the context of their limitations e.g., static analysis is not able to capture runtime behaviors, full code coverage is usually not achieved during dynamic analysis, etc. Whereas, in this paper, we analyze the performance of static and dynamic analysis methods in the detection of Android malware and attempt to compare them in terms of their detection performance, using the same modeling approach.**

To this end, we build on MAMADROID, a state-of-the-art detection system that relies on static analysis to create a behavioral model from the sequences of abstracted API calls. Then, aiming to apply the same technique in a dynamic analysis setting, we modify CHIMP, a platform recently proposed to crowdsource human inputs for app testing, in order to extract API calls' sequences from the traces produced while executing the app on a CHIMP virtual device. We call this system AUNTIEDROID and instantiate it by using both automated (Monkey) and user-generated inputs. We find that combining both static and dynamic analysis yields the best performance, with F-measure reaching 0.92. We also show that static analysis is at least as effective as dynamic analysis, depending on how apps are stimulated during execution, and investigate the reasons for inconsistent misclassifications across methods.

I. INTRODUCTION

In today's digital society, individuals rely on smart mobile devices and "apps" for a plethora of social, productivity, and work activities. Inevitably, this makes them valuable targets for cybercriminals, thus, more and more malware are developed every year exclusively targeting mobile operating systems [10] and, given its market share [33], Android in particular. Compared to desktop malware, malicious apps pose new threats as attackers might be able to, e.g., defeat two-factor authentication of banking systems [37] or continuously spy on victims through their phone camera or microphone [36].

As a result, the research community has proposed a number of techniques to detect and block Android malware based on either *static* or *dynamic* analysis. With the former, the code is recovered from the apk, and features are extracted to train machine learning classifiers; with the latter, apps are executed in a controlled environment, usually on an emulator or a virtual device, via a real person or an automatic input generator such as Monkey [17]. In particular, a few approaches have been recently proposed aiming to improve accuracy of malware detection. *(1) Behavioral Modeling:* Mariconti et al.'s MAMADROID [25] builds from static analysis, a behavioral model of malware samples, relying on the *sequences* of abstracted API calls; this yields higher accuracy than state of the art, while also providing higher resilience to API changes and reducing the need to re-train models. *(2) Input Generators:* previous work [4, 8, 23] has introduced input generators that aim to mimic app usage by humans, more effectively than the standard Android pseudorandom input generator (Monkey), thus improving the chances of triggering malicious code during execution. *(3) Hybrid Analysis:* by combining static and dynamic analysis, hybrid analysis has been used to try and get the best of the two worlds, typically, following two possible strategies. One approach is to use static analysis to gather information about the apps under analysis (e.g., intent filters an app listens for, execution paths to API calls, etc.) and then ensuring that all execution paths of interest are triggered during the dynamic analysis stage [8, 38]; in the other, features extracted using static analysis (e.g., permissions, API calls, etc.) are combined with those from dynamic analysis (e.g., file access, networking events, etc.), and used to train an ensemble machine learning model [21, 22].

Motivation. Overall, despite a large body of work proposing various Android malware detection tools, the research community's stance on whether to use static or dynamic analysis primarily stems from the systems limitations and the vulnerabilities to possible evasion techniques faced by each approach. For instance, static analysis methods that extract features from permissions requested by apps often yield high false positive rates, since benign apps may actually need to request permissions classified as dangerous [14], while systems that perform classification based on the frequency of API calls [1] often require constant retraining; moreover, reflection and dynamic code loading can be used to evade static analysis based detection. On the other hand, the accuracy of dynamic analysis is greatly dependent on whether malicious code is actually triggered during test execution, and in general dynamic analysis often does not scale. Nonetheless, we still lack a deep understanding of the advantages and disadvantages of each method in terms of simple detection performance.

Roadmap. In this paper, we aim to fill this research gap by addressing the following research questions: (1) Can we extend malware detection techniques based on behavioral modeling (in static analysis, as per MAMADROID [25]) to dynamic

analysis? (2) How do different malware analysis methods (i.e., static, dynamic, and hybrid analysis) compare to each other, in terms of detection performance, when the same technique is used to build malware detection models? Why? (3) Does having humans test apps during dynamic analysis improve malware detection compared to pseudorandom input generators such as Monkey [17]?

Aiming to answer these questions, we first of all modify CHIMP [2], a platform allowing to crowdsource human inputs to test Android apps, to support building a behavioral model based malware detection system (as per MAMADROID [25]). That is, we use the same approach as MAMADROID to extract sequences of abstracted API calls from the traces produced while executing the app in a virtual device (instead of the apk). We call this system AUNTIEDROID and instantiate it by using both automated (Monkey) and user-generated inputs. Then, we evaluate each analysis method, using the same modeling approach (i.e., a behavioral model relying on Markov chains built from the sequences of abstracted API calls), the same features, and the same machine learning classifier.

Contributions. Overall, we make several contributions. First, we introduce AUNTIEDROID, a virtual device that extends CHIMP [2] and allows for the collection of the method traces (from which features are extracted) produced by an app when executed. Second, we build and evaluate a hybrid system combining behavioral-based static and dynamic analysis features. Finally, we compare the different methods, showing that hybrid analysis performs best and that static analysis is at least as effective as dynamic analysis.

II. RELATED WORK

A. Static Analysis

Android malware detection based on static analysis aims to classify an app as malicious or benign by relying on features extracted from the app's apk, i.e., its source code. Techniques presented in [14, 18, 32] build features from the *permissions* requested by the apps, leveraging the fact that malware often tend to request dangerous/unneeded permissions. This approach, however, might be prone to false positives, as benign apps might also request dangerous permissions [14]. Moreover, since Android 6.0, the permission model allows users to grant permissions at run-time, when they are required, thus some dangerous permissions might never actually be granted (in fact, app developers often request permissions that are never used [16]). Drebin [5] combines several features extracted from the apps' manifest as well as disassembled code to train a classifier. Alas, techniques based on decompiled code can be evaded using dynamic code loading, reflection, and the use of native code [28, 30].

Other tools rely on *API calls*. DroidAPIMiner [1] performs classification based on the API calls more frequently used by malware. However, due to changes in the Android API, as well as the evolution of malware, this requires frequent retraining of the system as new APIs are released and new types of malware are developed. Deprecation and/or addition of API calls with new API releases is quite common, and this might prompt malware developers to switch to different API calls.

Also based on static analysis is MAMADROID [25], which uses behavioral models built from the *sequences*, rather than the frequency, of API calls. Specifically, it operates by characterizing the transitions between different API calls, involving the following four stages: (1) It extracts the call graph of an app, i.e., the control flow graph of the API calls in the apk; (2) It parses the call graph as sequences of API calls, which are abstracted to one of two modes, to either their "family" or package names. In `package` mode, an API call is abstracted to its package name using the list of around 338 packages from the Android and Google APIs, whereas in `family` mode, to the `google`, `java`, `javax`, `android`, `xml`, `apache`, `junit`, `json`, or `dom` families. Obfuscated and developer specific API calls are abstracted to `obfuscated` and `self-defined`, respectively; (3) Next, it models the sequences of (abstracted) calls as Markov chains, and extracts as features, the transition probabilities between states; and finally (4) it trains a machine learning classifier geared to label samples as benign or malicious.

MAMADROID achieves high detection accuracy (up to 0.99 F1-score), and preserves it for longer periods of time compared to [1], as it builds models that are more resilient to API changes and malware evolution. In this paper, for the static analysis part, we build on MAMADROID, re-using the source code publicly available from [26], to perform and compare malware detection using a behavioral model built from API sequences, while using both static, dynamic and hybrid analysis (see Section IV).

B. Dynamic Analysis

Dynamic analysis based techniques attempt to detect malware by capturing the runtime behavior of an app, targeting either generic malware behaviors or specific ones.

DroidTrace [39] uses ptrace (a system call often used by debuggers to control processes) to monitor selected system calls, allowing to run dynamic payloads and classify their behavior as, e.g., file access, network connection, inter-process communication, or privilege escalation. Canfora et al. [7] extract features from the sequence of system calls by executing apps on a VM, while Lageman et al. [20] models an app's behavior during execution on a VM using both system calls and `logcat` logs. CopperDroid [35] uses dynamic analysis to reconstruct malware behavior by observing executed system calls. While CrowDroid [6], a client running on the device, captures system calls generated by apps and sends them to a central server, which builds a behavioral model of each app. Whereas, we build a behavioral model of each app from the sequences of API calls invoked (rather than whether an API was invoked or not) during execution of the apps.

C. Hybrid Analysis

A few tools combine static and dynamic analysis, e.g., by using the former to analyze an apk and the latter to determine what execution paths to traverse, or by combining features extracted using both static and dynamic analysis.

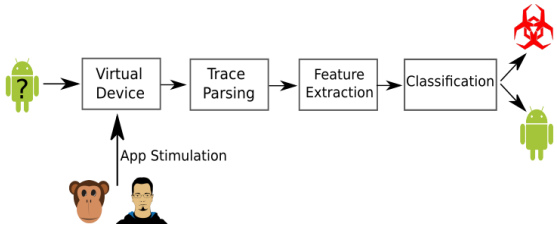


Fig. 1: High-level overview of AUNTIEDROID. An apk sample is run in a virtual device, using either Monkey or human. Then, the APIs called during execution are parsed and used for feature extraction. Finally, the app is classified as either benign or malicious.

Andrubis [22] is a malware analysis sandbox that dynamically builds a behavioral profile of an app using static (permissions, services, package name etc.) and dynamic features (reading/writing to files, sending SMS etc.). Note that, although we perform method tracing, similar to Andrubis, we use a virtual device and allow humans (and not only Monkey) to test the apps, as the latter perform a random sequence of actions/events which do not necessarily reflect how humans use the apps and may not trigger certain malicious code. Also, we build the behavioral profile of the apps from *all* API calls observed in the method traces, rather than selected APIs.

Marvin [21] uses features from both static and dynamic analysis to award malice scores (ranging from 0 to 10) to an app and classify as malware apps with scores greater than 5, while CuriousDroid [8], an automated user interface (UI) interaction for Android apps, integrates Andrubis [22] as its dynamic module in order to detect malware. It decomposes an app's UI on-the-fly and creates a context-based model generating series of interactions that aim to emulate real human interaction. IntelliDroid [38] introduces a targeted input generator that integrates with TaintDroid [13] aiming to track sensitive information flow from a source (e.g., a content provider such as contact list database) to a sink (e.g., network socket). It allows the dynamic analysis tool to specify APIs to target and generates inputs in a precise order that can be used to stimulate the Application Under Analysis (AUA) to observe potential malicious behavior.

Since there are several entry points into an Android app (e.g., via an activity, service, and broadcast), dynamically stimulating an AUA is usually done using tools like Monkey or MonkeyRunner, or humans. Targeted input generation tools such as CuriousDroid and Intellidroid aim to provide an alternative stimulation of apps that is closer to stimulation by humans and more intelligent than Monkey and MonkeyRunner.

Finally, we refer the reader seeking more details on the large body of work on Android malware to useful surveys of Android malware families and detection tools in [3, 15, 34] as well as an assessment of Android analysis techniques in [31].

III. AUNTIEDROID: BEHAVIORAL MODELING ON A VIRTUAL DEVICE

We now present AUNTIEDROID, a system performing Android malware detection based on behavioral models extracted via dynamic analysis. Our main objective is to compare its performance to its static analysis counterpart, i.e., MAMADROID [25]. In fact, we build on it, in that we again model the sequences of (abstracted) calls as Markov chains, and use the transition probabilities between states as features.

In order to build the behavioral model in dynamic analysis, we modify a virtual device to allow us to capture the sequence of API calls from the runtime execution trace of apps. We call the resulting system AUNTIEDROID, and summarize its operation in Fig. 1. First, we execute apps in a virtual device, stimulated by either an automated program (Monkey) or a human. We then parse the traces generated by the executions, and extract features for classification. The rest of this section presents the details of each component.

A. Virtual Device

As mentioned above, the first step in AUNTIEDROID is to execute apk samples in a virtual device with either (i) human users or (ii) an UI automation tool like the Monkey [17]. Our virtual device testbed, described in detail below, builds on CHIMP, an Android testing system recently presented in [2] which can be used to collect human inputs from mobile apps.

CHIMP [2]. CHIMP virtualizes Android devices using the Android-x86 platform, running behind a QEMU instance on a Linux server. Although it uses an x86 Android image, CHIMP actually supports two application binary interfaces (ABI), i.e., both ARM and x86 instruction sets are supported. Once running, the virtualized device can be stimulated by either a locally running automated tool (e.g., Monkey), or the UI can be streamed to a remote browser, allowing real humans to interact with it. CHIMP can be used to collect a wide range of data (user interactions, network traffic, performance, etc.) as well as explicit user feedback; however, for the sake of AUNTIEDROID, we modify it to generate and collect *run-time traces*, i.e., the call graph of an app's interactive execution.

Modifications to CHIMP. To effectively monitor malware execution, we substantially modify CHIMP from the prototype presented in [2], which was primarily designed to enable large-scale, human testing of benign apps. In fact, the original prototype supports code instrumentation via a Java code coverage library called EMMA, unfortunately, EMMA requires an app's source code to be instrumented, which is often not accessible for closed-source apps such as those analyzed in our work. Therefore, we modify CHIMP to get access to debug level run-time information from un-instrumented code. Note that in Android, each app runs on a dedicated VM which opens a debugger port using Java's Debug Wire Protocol (JWDP). As long as the *device* is set as debuggable (`ro.debuggable` property), we can connect to the VM's JWDP port to activate VM level *method tracing*.

We also have to activate tracing: in Android, one can either use Android's Activity Manager (AM) or the DDM Service. Both end up enabling the same functionality – i.e., `startMethodTracing` on `dalvik.system.VMDebug` and `android.os.Debug` – but through different approaches. That is, AM (via `adb am`) exposes a limited API that eventually reaches the app via Inter-Process Communication (IPC), while

the DDM Service (DDMS, as used by Android Studio) opens a connection directly to the VM's debugger, providing fine grain control over the tracing parameters. We choose the second approach since it is parameterizable, allowing us to set the trace buffer size, which by default (8MB) can only hold a few seconds of method traces. Hence, we implement a new DDM Service in CHIMP, using the ddmlib library [11] to communicate with the VMs and activate tracing. Our DDM service multiplexes all tracing requests through a single debugger and we further modify the ddmlib tracing methods to dump traces to the VM file system, and set the trace buffer size to 128MB. However, apps tested on the virtual device can generate more than 128MB of traces, thus, we add a background job that retrieves and removes traces from the VMs every 30s. Besides preventing the tracing buffer from filling up, this lets us capture partial traces for apps that might crash during stimulation.

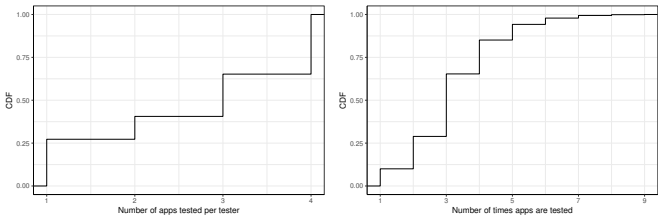
B. App Stimulation

As mentioned, to stimulate the AUA, we use both Monkey and humans.

Monkey [17]. Monkey is Android's de-facto standard UI automation tool used to generate inputs. In AUNTIEDROID, "Monkeys" (i.e., more than one Monkey instance) are deployed on the same machine that the virtual devices are running on. We set Monkeys to run a single app for 5 minutes (one virtual device VM per app): each Monkey is setup to generate events every 100ms and ignore timeouts, crashes, and security exceptions (although we still log and process them). Setting Monkey to generate input for 5 minutes only should not adversely affect code coverage, as prior work [9] reports that most input generators achieve maximum coverage between 5 to 10 minutes. As Monkey may generate events at a higher frequency than some apps can process, we also re-run offending apps with a decreased rate (300ms). As discussed in Section IV-C, some apps fail to execute, for one of three reasons: (i) they fail to install, (ii) crash, or (iii) have no interactive elements (e.g., background apps), as observed through `logcat` and from the Monkey output itself.

Humans. In order to have real users stimulate the samples, we recruited about 5k workers (5,030) from the Crowdflower.com crowdsourcing platform that are "historically trustworthy". We let them interact with the virtual device by streaming its UI to their browser via an HTML5 client. The client transmits user actions back to AUNTIEDROID, which translates them to Android inputs and forwards them to the virtual device. In addition to the virtual device UI, user controls were provided to, e.g., move to the next app in the testing session.

Each user was given 4 randomly selected apps from our dataset and told to explore as much of each app's functionality as possible before proceeding to the next app. CHIMP already provides heuristics to discard users with low engagement, and we do not enforce a lower bound on the time users must spend testing apps, since given the nature of our sample, some apps might have limited interaction opportunities. Consequently, we aim to have a median of at least three different users stimulate each app. In Fig. 2(a) and 2(b), respectively, we plot the CDF of the number of apps each user tests and the number of times an app is tested. We also limit app install time to 40s to avoid frustrating the users. We run the test sessions between August 9th and 11th, 2017 and we pay each user $0.12 per session.



(a) (b)

Fig. 2: Cumulative distribution function of the (a) number of apps tested per tester, and (b) number times apps are tested.

Ethics. For the experiments involving humans, we have obtained approval through our institution's ethical review process. Although we requested basic demographic data (age, gender, country), we did not collect privacy sensitive information, and users were instructed not to enter any real, personal information, e.g., account details. Also note that we provided email credentials to use when required, so that they did not have to use their own credentials or other contact information.

C. Trace Parsing

As discussed above, our virtual device component takes care of collecting method traces, network packets, and event logs generated when the app is running. To parse these traces, one could use different strategies, for instance, tracking data flow from selected sources (e.g., the device id using the `getDeviceID()` API) to sinks (e.g., a data output stream using the `writeBytes()` API), or using frequency analysis to derive commonly used API calls by malware (as in DroidAPIMiner [1]).

AUNTIEDROID follows the behavioral model based approach of MAMADROID, based on the *sequences* of API calls that the app performs at runtime, rather than statically extracting it from the apk. This way, we aim to capture different behavior when benign and malicious apps invoke API calls. For instance, a benign SMS app might receive an SMS, get the message body using `getMessageBody()` and afterwards, display the message to a user via a view by executing, in sequence, `setText(String msg)` and `show()` methods of the view. A malicious app, however, might exfiltrate all received SMSs by executing `sendTextMessage()` for every message before displaying it.

To derive the API call sequences, we collect the method traces and transform them into a call graph using *dmtracedump* [29]. From the call graph, we then extract the sequences using a custom script, while preserving the number of times an API call is executed as a multiplier in each sequence. As discussed above, to avoid losing traces when the trace buffer is full, we collect virtual device traces every 30s, and clear the buffer for incoming traces. Along the same lines, we have a median of three different users run

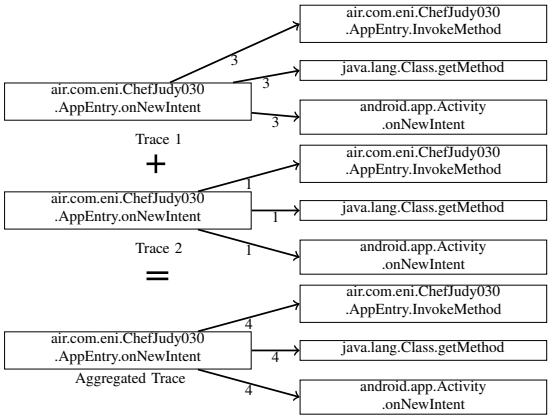


Fig. 3: Aggregated sequence of API calls showing the direct children of call `air.com.eni.ChefJudy030.AppEntry.onNewIntent`, and the number of times they are called (numbers on the arrow).

the same app to improve the quality of the traces gathered. As a result, we aggregate the sequences of API calls they generate for the same app into a single sequence. In Fig. 3, we provide an example of the sequence for the API call `air.com.eni.ChefJudy030.AppEntry.onNewIntent` when aggregated from two other sequences. We do not show the params and return type to ease presentation. Also, in some cases *Trace 1* may contain calls in a sequence that is not called in *Trace 2*, hence, the aggregated trace also reflects such calls.

D. Feature Extraction

As in MAMADROID [25], which operates in one of two modes i.e., `family` or `package`, AUNTIEDROID also abstracts each API call in the parsed trace to its corresponding family and package names using the Android API packages from API level 26 and the latest Google API packages. The abstraction allows for resilience to API changes in the Android framework as packages are added or deprecated less frequently compared to single API calls. It also helps to reduce the feature set size as the feature vector of each app is the square of the number of states in the Markov chain.

Also note that we modify MAMADROID's method of abstracting API calls: before performing abstraction to packages or families, we first abstract an API call to its class using a whitelist approach. We do this to avoid abstracting an API call to the wrong package or family in case an app prefixes its package name with one from the Android or Google APIs.

We then build a *behavioral model* from the abstracted sequences of API calls by building a Markov chain that models the sequences from which we extract features used to classify an app as either benign or malicious. More specifically, features are the probability of transitioning from one state (i.e., API call) to another in the Markov chain representing the API call sequences in an app.

E. Classification

Finally, we perform classification, labeling an app as benign or malware using a supervised machine learning classifier. More specifically, we use Random Forests with 10-fold cross validation. We choose Random Forests since it performs well on binary classification over high-dimension feature spaces.

IV. EXPERIMENTAL SETUP

A. Overview of the experiments

As discussed in Section I, we aim to perform a comparative analysis of Android malware detection systems based on behavioral models, using static, dynamic, and hybrid analysis. To this end, we perform three sets of experiments. (1) *Static:* We evaluate MAMADROID, which performs Android malware detection based on behavioral modeling in static analysis; (2) *Dynamic:* We analyze the detection performance of AUNTIEDROID (see Section V-B), which uses dynamic analysis, while also comparing automated input generation (Monkey) and human-generated input; (3) *Hybrid:* We combine static and dynamic analysis by merging the sequences of API calls from both methods, once again comparing Monkey and human based input generation.

All methods operate in one of two levels of abstraction, i.e., API calls are abstracted to either their family or package names. Overall, we use the same modeling technique and the same machine learning classifier. More specifically, we use the Random Forests classifier and in `family` (resp., `package`) mode, we use a configuration of 51 (resp., 101) trees with depth equal to 8 (resp., 32).

B. Datasets

Our evaluation uses two datasets: recent malware samples and a dataset of random benign apps, as discussed below.

Benign Samples. For consistency, we opt to re-use the set of 2,568 benign apps labeled as "newbenign" in [25]. In June 2017, we re-downloaded all the apps in order to ensure we have working apps and their latest version, obtaining 2,242 (87%) apps. We complement this list with a 33% sample of the top 49 apps (as of June 2017) from the 29 categories listed on the Google Play Store, adding an additional 481 samples. Overall, our benign dataset includes a total of 2,723 apps.

Malware Samples. Our malware dataset includes samples obtained in June 2017 from VirusShare – a repository of apps that are likely to be malicious. More precisely, VirusShare contains samples that have been detected as malware on various OS platforms, including Android. To obtain only Android malware, we check that each sample is correctly zipped and packaged as an apk, contains a Manifest file, and has a package name. Using this method, we gather 2,692 valid Android samples labeled as malware in 2017 by the antivirus engines on VirusShare. In addition, we add two more apps (Chef Judy and Dress Up Musa) from the Google Play Store reported as malware in the news and later removed from the play store.[1] In total, our malware dataset includes 2,694 apps.

C. Data Pre-Processing

Static Analysis. For static analysis, we re-use the source code of MAMADROID available on bitbucket. We set a timeout limit of six hours for call graph extraction, and are unable to

[1] https://goo.gl/hBjm0T and https://goo.gl/IQprtP

Failure	Benign	Malware
Already installed	10	9
Contains native code not compatible with the device's CPU	0	2
App's dex files could not be optimized and validated	0	1
Apk could not be unarchived by Android aapt	3	4
Shared library requested by app is not available on the device	0	1
Does not support the SDK (version 4.4.2) on the device	36	6
Requests a shared user already installed on the device	0	1
Android's failure to parse the app's certificate	0	4
Fails to complete installation within time limit (40s)	39	23
Total:	**88**	**51**

TABLE I: Reasons why apps fail to install on the virtual device.

obtain the call graphs for 98 (3.6%) and 251 (9.3%) apps in the benign and malware datasets, respectively. This is consistent with experiments reported in [25], due to the timeout but also to samples exceeding memory requirement (we allocate 16GB for the JVM heap space).

Dynamic Analysis. During dynamic analysis, before running the apps on the virtual device, we process them statically using androguard[2] to determine whether they have activities. Out of the total 5,417 apps in our datasets, we find that 82 apps contain no activity. As interaction with an Android app requires visuals that users can click, tap, or touch to trigger events, we therefore exclude these from the samples to be stimulated using Monkey or humans. We also remove 244 apps that do not have a launcher activity, since launching these apps on the virtual device will have no visual effect; i.e., no UI will be displayed to the tester. Finally, we do not include 139 apps which fail to install on the virtual device for one of the reasons shown in Table I.

Hybrid Analysis. To obtain a hybrid detection system, we merge the sequences of abstracted API calls (from which we extract features) obtained using both static and dynamic analysis. More specifically, we merge the sequences of API calls following the same strategy used to aggregate the traces discussed in Section III-C. Naturally, for hybrid analysis, we use samples for which we have traces for both static and dynamic analysis.

Final Datasets. In Table II we report, in the right-most column, the final number of samples in each dataset, for each method of analysis. During dynamic analysis, we fail to obtain traces for 724 apps when stimulating with Monkey and 693 when stimulating with humans. This happens for various reasons, and we defer further analysis to the full version of the paper Note that the hybrid analysis method consists of samples for which we obtain traces both statically and dynamically.

V. Evaluation

We now present the results of our experiments, reporting detection performance and, for dynamic analysis, code coverage.

A. Static Analysis

To evaluate the static analysis technique, we use a slightly modified version of MaMaDroid [25]. Also note that, while [25] uses API level 24, we use the more recent API level 26. We run our experiments on the samples (2,625 benign and 2,443 malware) for which we obtain call graphs, and report the F-measure obtained when operating in `family` and `package` modes in the top two rows of Table III. We observe that the latter performs slightly better, achieving F-measure of 0.91, compared to 0.86 in the former which is consistent with the results reported in [25]. The `package` mode achieves higher F-measure than `family` mode as it captures the behavior of apps at a finer granularity which reveals more distinguishing behaviors between malware and benign apps as demonstrated by higher precision and recall (see Table III).

[2]https://github.com/androguard/androguard

Analysis	Stimulator	Category	#Samples	#Traces / Call graphs
Static (MaMaDroid)	–	Benign	2,723	2,625
		Malware	2,694	2,443
Dynamic (AuntieDroid)	Human	Benign	2,596	2,348
		Malware	2,356	1,911
	Monkey	Benign	2,596	2,336
		Malware	2,356	1,892
Hybrid	Static & Human	Benign	2,596	2,235
		Malware	2,356	1,708
	Static & Monkey	Benign	2,596	2,234
		Malware	2,356	1,686

TABLE II: Datasets used to evaluate each method of analysis.

Analysis	Stimulator	Mode	F-measure	Precision	Recall
Static (MaMaDroid)	–	`Family`	**0.86**	0.84	0.88
		`Package`	**0.91**	0.89	0.93
Dynamic (AuntieDroid)	Human	`Family`	**0.85**	0.80	0.90
		`Package`	**0.88**	0.84	0.92
	Monkey	`Family`	**0.86**	0.84	0.89
		`Package`	**0.92**	0.91	0.93
Hybrid	Static & Human	`Family`	**0.87**	0.86	0.88
		`Package`	**0.90**	0.88	0.91
	Static & Monkey	`Family`	**0.88**	0.88	0.89
		`Package`	**0.92**	0.92	0.93

TABLE III: Results achieved by all analysis methods while using human and Monkey as app stimulators during dynamic analysis.

B. Dynamic Analysis

Next, we report the results achieved by dynamic analysis (i.e., using AuntieDroid), comparing between stimulation performed by Monkey and humans.

Detection Performance. For Monkey, we use the dataset shown in Table II, i.e., on 2,336/1,892 samples for benign/malware. When AuntieDroid runs in `family` mode, it achieves F-measure, precision, and recall of 0.86, 0.84, and 0.89, respectively. Whereas in `package` mode, it achieves F-measure, precision, and recall of 0.92, 0.91, and 0.93, respectively, as reported in Table III. When humans stimulate the apps (2,348 benign and 1,911 malware) and AuntieDroid runs in `family` mode, we get F-measure, precision, and recall of 0.85, 0.80, and 0.90, respectively. Whereas when operating in `package` mode, F-measure, precision, and recall go up to 0.88, 0.84, and 0.92, respectively (see Table III).

Overall, lower F-measures in all modes of operation in dynamic analysis compared to static analysis (i.e., AuntieDroid vs MaMaDroid) are due to increases in false positives. In fact, recall is around 0.90 on all experiments, while precision is as low as 0.80 (`family` mode with humans).

Code Coverage. As mentioned, the performance of dynamic analysis tools is affected by whether malicious code is trig-

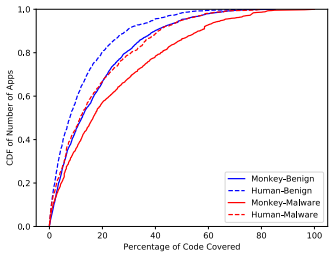
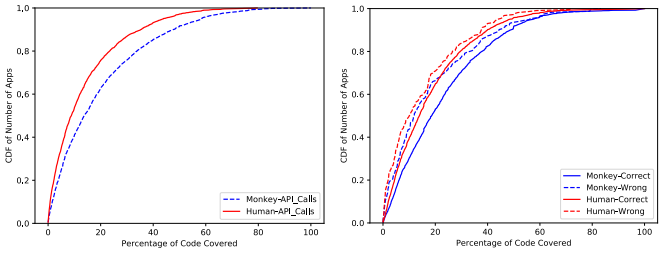


Fig. 4: Cumulative distribution function of the percentage of code covered in benign and malicious apps when they are stimulated by Monkey and human.



(a) (b)

Fig. 5: Cumulative distribution function of the percentage of code covered (a) when apps are stimulated by humans and Monkey, and (b) when the correctly classified and misclassified apps are stimulated by humans and Monkey.

gered during execution. Since AUNTIEDROID relies on the sequences of API calls to detect malware, we analyze code coverage of each app to measure how much of an app's API calls Monkey/humans successfully trigger. Thus, we focus on API calls that begin with the package name of the app.

For the apps for which we obtain traces (85% and 86%, resp., for Monkey and human), Monkey is able to trigger on average, 20% of the API calls. In Fig. 4, we plot the cumulative distribution function (CDF) of the percentage of code covered, showing that for 90% of the benign apps, at least 40% of the API calls are triggered by Monkey. Whereas with respect to the malware samples, at least 57% of the API calls are triggered. As for humans, we find that users are able to trigger, on average, 14% of the API calls. Similarly, Fig. 4 shows that at least 29% of the API calls are triggered in 90% of the benign apps. However, with 90% of the malicious apps, 41% of the API calls are triggered.

With both stimulators, there is a higher percentage of code coverage in the malware apps than in benign apps. This is due to malware apps being smaller in size compared to the benign apps in our dataset. The mean number of API calls in the benign and malware apps are respectively, 43,518 and 16,780. However, with respect to stimulators, Monkey is able to trigger more code in apps compared to humans, which is likely due to Monkey triggering more events than humans in the time each spend testing the apps.

We also investigate the prevalence of dynamic code loading in the wild, as it could be used for malicious purposes [28] (e.g., to evade static analysis). Due to space limitations, we defer findings to the full version of the paper [27].

C. Hybrid Analysis

We now report the results achieved by hybrid analysis, comparing between stimulation performed by Monkey and humans. Recall that only samples for which we have obtained a trace in both static and dynamic analysis, as reported in Table II, are merged and evaluated.

In `family` mode, the hybrid system, using traces produced by Monkey, achieves an F-measure of 0.88, whereas when using traces produced by humans, 0.87. When operating in `package` mode and using Monkey, it achieves an F-measure of 0.92, and 0.90 with humans, as reported in Table III.

Note that we do not report code coverage in hybrid analysis because the traces from static analysis are an overestimation of the API calls in the app. Hence, merged traces do not reflect code covered in each app when executed.

VI. COMPARATIVE ANALYSIS

We now set out to examine and compare: (1) the detection performance of each analysis method, i.e., detecting malware based on a behavioral model built via static, dynamic, or hybrid analysis, (2) the samples that are misclassified in each method, and (3) the samples misclassified in one method but correctly classified by another. Due to each method having inherent limitations, it is not clear from prior work how they compare against each other. Therefore, in this section, we shed light on their comparisons.

A. Detection Performance

We start by comparing the results of the different analysis methods. Recall that we have abstracted each API call to either its family or package name, therefore, each method operates in one of two modes. When operating in `family` mode, with static analysis we achieve an F-measure of 0.86, whereas, with dynamic analysis, we achieve F-measure of 0.86 when apps are stimulated by Monkey and 0.85 when stimulated by humans (See Table III). In `package` mode, we achieve F-measure of 0.91 with static analysis, whereas with dynamic analysis we achieve F-measure of 0.92 when apps are stimulated by Monkey and 0.88 when stimulated by humans (See Table III).

The results show that static analysis is at least as effective as dynamic analysis depending on the app stimulator used during dynamic analysis. We believe this is because the behavioral model used to perform detection primarily leverages API calls. Although static analysis is not able to detect maliciousness when code is loaded dynamically, it provides an overestimation of the API call sequences in the apk. Consequently, all behaviors that can be extracted from the apk are actually captured by the static analysis classifier. On the other hand, dynamic analysis captures only the behavior exhibited by the samples during runtime. Hence, any behavior not observed during runtime is not used in the decision-making of the dynamic analysis classifier.

To verify this hypothesis, we evaluate how the percentage of code covered differs when different app stimulators are employed as well as in correctly classified and misclassified

samples. From Fig. 5(a), we observe that when Monkey is used as the app stimulator, at least 48% of the API calls are triggered in 90% of the samples, compared to 35% when they are stimulated by humans. Similarly, as shown in Fig. 5(b), 49% of the API calls are triggered in 90% of the samples correctly classified when Monkey is used to stimulate apps compared to 44% of API calls in 90% of the apps that are misclassified. When humans are used to stimulate the apps, 40% of the API calls are triggered in 90% of samples that are correctly classified compared to 38% triggered in 90% of samples misclassified. As a result of better code coverage, dynamic analysis performs better when apps are stimulated by Monkey compared to when apps are stimulated by crowdsourced users. Therefore, we find that, other than the non-susceptibility to evasion techniques such as dynamic code loading, dynamic analysis tools based on API calls may have no advantage over static analysis based tools unless the code coverage is improved.

However, when traces from static and dynamic analysis are merged into a hybrid system, in `family` mode, we achieve F-measure of 0.88 using Monkey compared to 0.86 achieved by both static analysis and dynamic analysis (with Monkey) alone. Similarly, we achieve F-measure of 0.87 when the dynamic traces are generated with humans stimulating the apps compared to 0.86 and 0.85 achieved respectively by static and dynamic (humans) analysis alone. In `package` mode, the hybrid system achieves F-measure of 0.92 when the dynamic traces are produced by Monkey and 0.90 with humans. The hybrid system outperforms the dynamic analysis system in all modes (i.e., `family` and `package`), as it also captures behavior not exhibited during runtime execution of the apps as a result of the overestimation from static analysis, while it improves static analysis as it captures frequently used API calls – a behavior that cannot be captured by static analysis – and API calls that are dynamically loaded.

B. Misclassifications within each analysis method

Next, we examine the samples that are misclassified in each method of analysis, aiming to understand the differences in the model of the correctly classified and misclassified samples. We perform our analysis on samples that have been classified by all three methods in `package` mode.

We formulate and verify the hypothesis that misclassifications are due to missing API calls that are considered "important" by the classifiers. To this end, we select the 100 most important features used by each classifier to distinguish between potential malware and benign samples, and evaluate the average number of these features present in each sample. We select the 100 most important features because it represents, at most, about 10% of the features recorded in our experiments. Recall that a feature in our detection technique is the probability of evoking an abstracted API call, and transitions not evoked during the experiments have probability of 0. The maximum number of features with probability > 0 in our dataset is 1,869 (static analysis) and the minimum is 1,022 (dynamic analysis with humans). We expect that samples that are misclassified will have a similar number of important features as those of the opposite class.

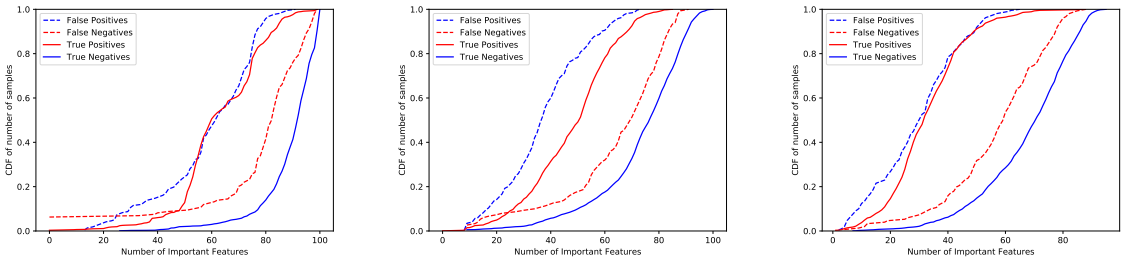
Therefore, using the top 100 features for each classifier, we compare the average number of the features in the true positives (i.e., correctly classified malware samples) to the false negatives (malware classified as benign), as well as true negatives to false positives.

False Positives. We also count the number of false positives (i.e., benign samples classified as malware) in each method of analysis, respectively, when apps are stimulated by humans and by Monkey during dynamic analysis. With the former, there are 215, 317, and 209 false positives, respectively, with static, dynamic, and hybrid analysis. With the latter, we get 217, 178, and 137 false positives with static, dynamic, and hybrid analysis. Using the top 100 features, we find that the false positives in static analysis exhibit similar behavior to that observed in true positives. Specifically, they have, on average, 54.12 ± 22.65 features out of the 100 most important features, which is similar to 59.96 ± 19.46 in true positive samples. The same behavior is also observed in both dynamic and hybrid analysis irrespective of the app stimulator. In Fig. 6, we plot the CDF of the number of features present in each classification type for all analysis methods when humans stimulate apps during dynamic analysis and, in Fig. 7, with Monkey. That is, the behavioral model of the false positives in all analysis methods is similar to that observed on the true positives. For example, in Fig. 6(c) (hybrid analysis) 90% of the false positives have no more than 50 of the 100 most important features (similar to the true positives – 49/100) while true negatives reach 86 features out of 100.

False Negatives. Again, we count the number of false negatives (i.e., malware samples classified as benign) in each analysis method, resp., when apps are stimulated by humans and Monkey. With the former, there are 148, 151, and 153 false negatives, respectively, in static, dynamic, and hybrid analysis, while, with the latter, we get 149, 132, and 126 false negatives. In static analysis, we find that the behavioral model of the false negatives are similar to that observed in the true negatives. In particular, of the 100 most important features used to distinguish malware from benign samples, there are, on average, 82.08 ± 11.75 features per false negative sample. The value is more similar to the 88.91 ± 11.31 important features per true negative sample rather than the 59.96 ± 19.46 important features per true positive sample. The same result is also observed in dynamic analysis irrespective of the stimulator, and in hybrid analysis as well. Recall that, in Fig. 6 and 7, we plot the CDF of the number of features in each classification type when, resp., human and Monkey are used as the stimulator during dynamic analysis; e.g., in Fig. 7(b) (dynamic analysis), 90% of the false negative samples have 84 of the 100 features, a value more similar to 89 features (true negatives) rather than 70 features (true positives).

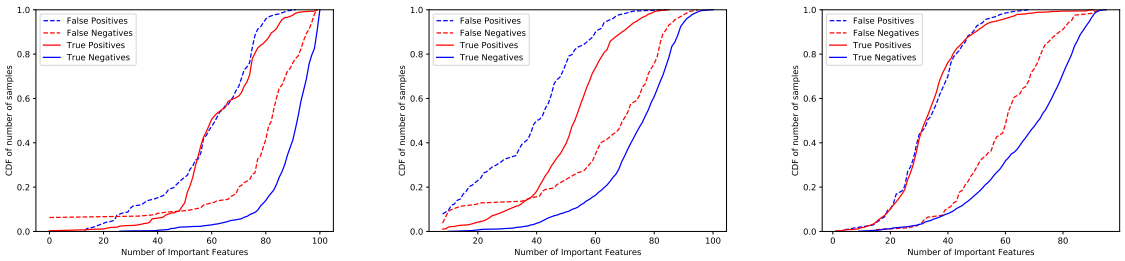
C. Misclassifications across analysis methods

Next, we attempt to clarify why some samples are misclassified by one method of analysis but correctly classified by



(a) Static Analysis (b) Dynamic Analysis (c) Hybrid Analysis

Fig. 6: CDF of the number of features present (out of the 100 most important features) in each classification type for all analysis methods, with **human**, during dynamic analysis.



(a) Static Analysis (b) Dynamic Analysis (c) Hybrid Analysis

Fig. 7: CDF of the number of features present (out of the 100 most important features) in each classification type for all analysis methods, with **Monkey**, during dynamic analysis).

another. The first important difference among the methods is the code coverage: dynamic analysis does not cover the entire code base of an app. Moreover, stimulating with Monkey vs humans yield different code coverage. This might result in a few different scenarios: 1) Dynamic analysis may not have triggered the malicious code that is captured in static analysis; 2) Static analysis may reveal sequences of API calls that are not necessarily malicious, but characterize many malicious apps; 3) API calls not triggered during dynamic analysis may affect the Markov chains leading to training poisoning or misclassification of the sample, depending on whether the sample is part of the training or the test set.

Scenarios (1) and (2) are possible reasons why static analysis correctly detects some samples and dynamic analysis does not, while (3) refers to the opposite. Although the hybrid system captures sequences of API calls from both static and dynamic analysis, it actually results in completely new Markov chains and features for training and classification. While more accurate than the individual methods, as the features values change (i.e., the transition probabilities), it behaves differently.

Another important factor is the presence of loops in the code waiting for user interaction. A good example are the games *Dumb ways to die 1 & 2*. These apps have "minigames" where a user has to click several times on the right spot of the screen at the right time. When executed, the apps enter a loop waiting for user action, and decide the next action based on what happened before returning to waiting for user action. Static analysis would catch the four different outcomes (i.e., execution path) of the loop, i.e., wrong click, correct click, the user won the game, the user lost the game. Dynamic analysis would repeat the loop many times depending on the continuous clicks of the human or of Monkey, and the user/Monkey may never win or lose. Static analysis will record the four possible loop paths without repeating the sequences in its traces, and the user/Monkey may not record all the possible sequences, but have duplicated sequences due to multiple clicks resulting in the same outcome. All these differences characterize the recorded traces, and therefore may result in different Markov chains and decisions among the methods.

VII. Discussion & Conclusion

In this paper, we analyzed different Android malware detection analysis methods, i.e., static, dynamic, and hybrid analysis, using a common modeling approach. Specifically, we built a behavioral model of each sample based on the sequences of abstracted API calls, as done by MaMaDroid [25], as it effectively captures malicious behavior even in the presence of changes in the Android API and evolving malware. We then introduced a dynamic analysis tool, AuntieDroid, which supports app stimulation via both humans (via crowdsourcing [2]) and pseudorandom input generators (Monkey). We also slightly modified MaMaDroid to first abstract an API call to its class, before abstracting to other modes, to avoid abstracting to the wrong package. Then, to build a hybrid system, we merged the sequences of API calls from static and dynamic analysis. All three methods operate in one of two modes, i.e., `family` and `package`, based on the level of abstraction; in `family` mode, static, dynamic (human/Monkey), and hybrid analysis, respectively, achieve F-measures of 0.86, 0.85/0.86, and 0.88. Whereas, in `package` mode, we achieve

0.91, 0.88/0.92, and 0.92.

Overall, our experiments showed that hybrid analysis performs best because it captures the best of static and dynamic analysis, as it is able to capture the sequences of API calls that are actually executed and/or dynamically loaded (from the latter), and capture code not executed during testing due to code overestimation (from the former). Nonetheless, static analysis performs well overall, often better than dynamic analysis; when looking at misclassifications across methods, we found that those occurring in dynamic but not in static analysis are likely due to poor code coverage, thus, the feature vectors in dynamic analysis may not reveal features (e.g., a chunk of benign code in repackaged samples) that characterize malware in our dataset. Finally, we showed that dynamic analysis performs better with Monkey than humans because the former is able to trigger more code than the latter.

Although some characteristics peculiar to AUNTIEDROID's virtual device (e.g., it runs as a hardware assisted virtualization) should prevent evasion by malware that tries to circumvent emulators/virtual devices using environment variables [12, 19, 24], we plan, as part of future work, to update it to use a virtual device that appears as close to a real device as possible. Moreover, we intend to use input generators that target specific behaviors of an app, so as to target certain API calls mostly used by malware rather than trying to improve the code coverage during dynamic analysis. Finally, we plan to detect and measure the prevalence of malware that specifically employs dynamic code loading as an evasion technique.

Acknowledgments. Lucky Onwuzurike was funded by the Petroleum Technology Development Fund (PTDF), while Enrico Mariconti was supported by the EPSRC under grant 1490017.

REFERENCES

[1] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *SecureComm*, 2013.

[2] M. Almeida, M. Bilal, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Varvello, and J. Blackburn. CHIMP: Crowdsourcing Human Inputs for Mobile Phones. In *WWW*, 2018.

[3] A. Amamra, C. Talhi, and J.-M. Robert. Smartphone malware detection: From a survey towards taxonomy. In *IEEE MALWARE*, 2012.

[4] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *ACM FSE*, 2012.

[5] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *NDSS*, 2014.

[6] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-based Malware Detection System for Android. In *SPSM*, 2011.

[7] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio. Detecting Android Malware Using Sequences of System Calls. In *Workshop on Software Development Lifecycle for Mobile*, 2015.

[8] P. Carter, C. Mulliner, M. Lindorfer, W. Robertson, and E. Kirda. Curiousdroid: automated user interface interaction for android application analysis sandboxes. In *FC*, 2016.

[9] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet?(e). In *IEEE ASE*, 2015.

[10] J. Clay. Continued Rise in Mobile Threats for 2016. http://blog.trendmicro.com/continued-rise-in-mobile-threats-for-2016/, 2016.

[11] Ddmlib: APIs for talking with Dalvik VM. https://mvnrepository.com/artifact/com.android.ddmlib/ddmlib, 2017.

[12] W. Diao, X. Liu, Z. Li, and K. Zhang. Evading android runtime analysis through detecting programmed interactions. In *WiSec*, 2016.

[13] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM TOCS*, 32(2), 2014.

[14] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *ACM CCS*, 2009.

[15] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan. Android Security: A Survey of Issues, Malware Penetration, and Defenses. *IEEE Communications Surveys & Tutorials*, 17(2), 2015.

[16] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *ACM CCS*, 2011.

[17] Google. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html, 2017.

[18] C.-Y. Huang, Y.-T. Tsai, and C.-H. Hsu. Performance evaluation on permission-based detection for Android malware. In *Advances in Intelligent Systems and Applications*, 2013.

[19] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu. Morpheus: automatically generating heuristics to detect android emulators. In *ACSAC*, 2014.

[20] N. Lageman, M. Lindsey, and W. Glodek. Detecting malicious android applications from runtime behavior. In *IEEE MILCOM*, 2015.

[21] M. Lindorfer, M. Neugschwandtner, and C. Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *COMPSAC*, 2015.

[22] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. v. d. Veen, and C. Platzer. ANDRUBIS – 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *BADGERS*, 2014.

[23] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An Input Generation System for Android Apps. In *ACM ESEC/FSE*, 2013.

[24] D. Maier, T. Müller, and M. Protsenko. Divide-and-Conquer: Why Android Malware Cannot Be Stopped. In *ARES*, 2014.

[25] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *NDSS*, 2017.

[26] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini. Mamadroid Source Code. https://bitbucket.org/gianluca_students/mamadroid_code, 2017.

[27] L. Onwuzurike, M. Almeida, E. Mariconti, J. Blackburn, G. Stringhini, and E. De Cristofaro. A Family of Droids: Analyzing Behavioral Model based Android Malware Detection via Static and Dynamic Analysis (Extended Version). *arXiv:1803.03448*, 2018.

[28] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *NDSS*, 2014.

[29] Profiling with Traceview and dmtracedump. https://developer.android.com/studio/profile/traceview.html, 2017.

[30] V. Rastogi, Y. Chen, and X. Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE TIFS*, 2014.

[31] B. Reaves, J. Bowers, S. A. Gorski III, O. Anise, R. Bobhate, R. Cho, H. Das, S. Hussain, H. Karachiwala, N. Scaife, et al. *droid: Assessment and Evaluation of Android Application Analysis Tools. *ACM Computing Surveys*, 49(3), 2016.

[32] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. Álvarez. Puma: Permission usage to detect malware in android. In *CISIS*, 2013.

[33] Statista. Global mobile OS market share in sales to end users from 1st quarter 2009 to 1st quarter 2016. https://goo.gl/nZGSbh, 2016.

[34] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro. The evolution of android malware and android analysis techniques. *ACM Computing Surveys*, 49(4), 2017.

[35] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *NDSS*, 2015.

[36] The Independent. Chrysaor: Android Spyware Designed to Hack Smartphone Cameras Discovered. http://ind.pn/2tajLaD, 2017.

[37] The Register. Google AdSense abused to distribute Android spyware. http://www.theregister.co.uk/2016/08/15/android_trojan_abuses_google_adsense/, 2016.

[38] M. Y. Wong and D. Lie. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In *NDSS*, 2016.

[39] M. Zheng, M. Sun, and J. C. Lui. DroidTrace: a ptrace based Android dynamic analysis system with forward execution capability. In *IWCMC*, 2014.