Towards Understanding the Role of Execution Context for Observing Malicious Behavior in Android Malware

Catherine Boileau[1], François Gagnon[2(✉)], Jérémie Poisson[2], Simon Frenette[2], and Mohammed Mejri[1]

[1] Université Laval, Québec, Canada
catherine.boileau.1@ulaval.ca, mohamed.mejri@ift.ulaval.ca
[2] CybersecLab at Cégep de Sainte-Foy, Québec, Canada
cybersecurity@cegep-ste-foy.qc.ca

Abstract. Favorite target of mobile malware, Android operating system can now rely on numerous tools, instrumentations and sandbox environments to fight back the malware threat. Sandboxing is a popular dynamic approach to detect malware, where an application is submitted to a plethora of tests in order to determine the presence of malicious behavior. Such existing sandboxes usually performed analysis on a malware sample once, given the tremendous amount of applications to analyze. In order to further study what trigger malware behavior, we decided to submit a malware sample multiple times to our sandbox, each time with slightly different experiment parameters, such as level of user simulation, the number of user actions performed, and the network configuration. Our results show that a proper configuration of these parameters will yield more information about the sample under study.

1 Introduction

Android was the most popular platform in 2015 for mobile devices and malware designers alike, as 97% of malware were designed for Android, who holds 85% of the market share in the mobile world. Besides its popularity, the burgeoning of many third-party app stores is helping to distribute malware around the world, as some of them may contain up to 8% of malicious applications [1].

In order to countenance this threat and protect the Android operating system, tools and systems are developed to study and detect malicious behavior in applications. In recent years, sandboxing, defined as the execution of malware in a closed and virtual environment [2], has been used to analyze Android applications and monitor malicious behavior.

Most of the systems are focusing on detecting malicious applications, by submitting an application to a sandbox. Once the experiment is completed, the results are saved in a report and the application is not analyzed again, as analysis is time-consuming and new applications hit markets at an ever faster rate.

Since the literature is currently bent on detecting malware by a one-time submission to a sandbox, we explored a different approach, studying the same

M.S. Obaidat (Ed): ICETE 2016, CCIS 764, pp. 48–71, 2017.
https://doi.org/10.1007/978-3-319-67876-4_3

malware repeatedly in slightly different environments. These variations would allow us to determine the optimal environment to discover malicious behavior and possibly help to detect more malware on their first submission to a sandbox. For example, applications depending on specific events or parameters (e.g. time-bombs or location-specific apps) to expose their malicious behavior could be detected in repeated experiments where a parameter (e.g. the location, the time) is changed to different values.

In our preliminary group of experiments, we started with 3 variations in application usage: only installing the application, installing and starting the application and finally, installing, starting and simulating the application [3]. In this paper, we repeat our preliminary paper experiment, and add 3 new groups of experiments where the variation is the number of actions performed in user simulation, the rest period after simulation and the network configuration. Other parameters are still on our to-do list, such as using a different Android version of the emulator and different types of emulators.

Therefore, we will first present, in Sect. 2, related work about publicly available sandboxes performing dynamic analysis and then, a brief description will follow of our sandbox environment and its possible configurations in Sect. 3. Afterwards, a description of the experiment will be presented in Sect. 4 as well as the scenarios used. Finally, results will be discussed in Sect. 5, together with future work in Sect. 6, and concluding remarks in Sect. 7.

2 Related Work

Before the first Android malware was discovered in August 2010 [4], sandboxing techniques were already useful to fight PC-based digital threats [5–7]. Malware then started to spread to the mobile world, and static [8–11] and dynamic [12–14] analysis tools were adapted to face the new challenge. Afterwards, hybrid systems using both static and dynamic analysis [15,16] sprang to life and matured into the following public sandboxes analyzing Android malwares.

First among them is ANDRUBIS [17], a system performing both static and dynamic analysis on a large scale. Using similar tools as our sandbox, and keeping tracks of numerous metrics, ANDRUBIS analyzes an app once, and if submitted again, the report from the first run is presented to the user. Mobile-sandbox [18] is another hybrid analysis system tracking native API calls. As their goal is to detect malware via a new metric, they process their apps once.

CopperDroid [19] is also part of the dynamic analysis sandbox family and their approach is closer to our own, as they compare applications behaviour with and without user simulation. They are thus able to demonstrate the usefulness of simulation during an experiment, whereas we pushed this logic to multiple scenarios and are interested in putting results into perspective. Tracedroid [20] is also a hybrid analysis service, based on method traces as an extension on the virtual machine. As the other tools, they aim at processing a quantity of apps, to then label and sort out malware. To our knowledge, Android sandboxing is mostly used to detect malware from good applications and not to observe

behaviour in different contexts. Moreover, once a malware is processed, it is not analyzed again.

As our approach implies multiple runs with variations in parameters, environments and network configurations, we find that fuzzing is somehow related to our work. Per definition, fuzzing is an automated technique that provides boundary test cases and massive inputs of data in order to find vulnerabilities [21]. In the Android world, the framework AndroidFuzzer was developed in the cloud to that intent. Some tools focused on a particular area of testing, such as permissions [21], activities or intents [22,23]. However, to the best of our knowledge, fuzzing has not been extended to the analysis or detection of malwares.

3 Methodology

In order to observe variations in application behaviour, we use a client-server sandbox [24,25] that executes an experiment in a particular context. An experiment is defined as the execution of a scenario (a series of actions to configure, install, test, and collect data on apps) applied to all samples of our dataset. The server side of the sandbox is responsible for the management of the experiment (i.e., managing the clients pool, the distribution to clients, etc.), whereas clients manage runs (i.e., the execution of the selected scenario for one particular app of the pool). The client-server architecture was selected for its scalability, for it is as simple as adding a client to process more applications in the same time. A server-controlled experiment also ensures that all runs in an experiment will execute the same scenario with the same parameters (network configuration, android version, etc.), since the configuration of the experiment is done on the server and then forwarded to each available client. All machines, server and clients are connected to the same dedicated subnet, policied by our fake server to provide the same network context to all clients. Thus, the sandbox architecture lowers the variability in an experiment to allow a sound comparison between runs of a same experiments and between experiments on the same malware.

3.1 Client-Server Architecture

First, our sandbox server works as a controller over the whole experiment. To begin with, all parameters are defined in the configuration file, to prepare the environment (network configuration and android version) and the parameters (scenario to use, dataset to analyze, etc.) needed for the experiment. Once configured, the server will start and monitor the pool of clients. Each new client will register itself to the server pool of clients, thus notifying the server of its availability to perform a part of the experiment. As soon as the server detects that a client is ready to execute a run, it will give that client the scenario to execute and the application to analyze.

At that moment, the client starts its run by loading the scenario and the application to install. The scenario is parsed by the client, to extract its parameters and actions to perform (see following Sect. 3.2 for more details). Following

that, the analysis phase starts with the sequence of actions to execute. Once the client concludes the action phase, data collected (see following Sect. 3.4) by multiple tools is bundled into a result file and the server is notified that results are ready to be stored. Finally, the client changes its status back to available, thus letting the server know it is ready to execute another run.

Thus, the server pushes runs to available clients until all the applications in the dataset are processed, completing the experiment. Gathered data is then stored into our results database, where it is available for post-processing and analysis.

3.2 Experiment Scenario

The scenario, an XML document, contains two parts: the environment configuration and the action sequence to execute. The environment configuration indicates which Android virtual device (AVD) must be used, thus specifying the version of Android to use, and what network services should be provided (e.g. DNS and HTTP proxies). The action sequence is an ordered list of commands that is launched by the client to successfully perform a credible simulation of the application. Each scenario starts with the same set of instructions, to prepare the environment for the experiment. For example, starting the AVD, waiting for the boot sequence to complete and starting the monitoring of metrics are parts of the initialization sequence.

Once this preparation phase is over, the second phase, the installation of the app to test and user simulation, starts. When a scenario includes user simulation steps, the Application Exerciser Monkey[1] tool is used to perform the number of actions contained in the simulation step. Its capacity to generate pseudo-random and system-level events to navigate the application under scrutiny makes for a basic simulation. The final user simulation action is a rest period, where no simulation is done, but the application continues to run. Finally, once all actions have been executed, the data collection phase is launched and a report with all the metrics is sent back to the sandbox server.

Finally, in order to simulate a basic network connectivity, our sandbox, server and clients, can run a DNS proxy (relaying DNS queries/responses to a real server) as well as a gateway to redirect all outgoing network traffic generated by the mobile applications to our own fake server, who can complete TCP handshake, to let apps perform requests (i.e. HTTP GET requests).

3.3 Applications Datasets

We performed our preliminary experiments [3], which are presented in Sect. 4, on a different dataset than the new experiments presented in this paper. The first dataset, labelled D1, was composed of 5519 applications, both legitimate and malicious, as shown in Table 1. Of these applications, 3519 were tagged as

[1] http://developer.android.com/tools/help/monkey.html.

malware, coming from 2 different sources: the Malware Genome project[2] provided 1260 apps while DroidAnalytics [11] provided the remaining 2259 malware. The rest of the dataset comprehend 2000 legitimate applications, where the first thousand were collected on the Google Play Store[3] and the remaining were pulled from the application store AppsAPK[4]. In all figures, application sources were labelled as follows: GooglePlay, for the Google Play Store, and MalGenome, for the Malware Genome Project, while AppsAPK and DroidAnalytics remained the same.

Table 1. Composition of datasets D1 and D2.

Source	Dataset1 (D1)	Dataset2 (D2)	Type
MalGenome	1260	100	Malware
DroidAnalytics	2259	100	Malware
Contagio	0	100	Malware
Source-2013	0	100	Malware
Source-2014	0	100	Malware
GooglePlay	1000	100	Legitimate
AppsAPK	1000	100	Legitimate
Total	5519	700	Both

Our second dataset (referred to as D2) was composed of 700 applications (see Table 1), coming from 7 different sources. We randomly selected 100 applications from each source in D1, dividing applications into 200 malware and 200 legitimate applications. Furthermore, we added 3 other sources of malware, that provided 100 applications each: Contagio[5], and a private source with malware from 2013 and 2014, which were named Source-2013 and Source-2014. These 3 new sources brought the number of malware to 500, creating a bias toward malware in Dataset2. In all figures, these 3 sources are labelled with their respective names.

We created a second dataset following the first experiments, in order to confirm trends from our first observations but also to increase the number of experiments performed in the same amount of time. On a smaller number of applications, more experiments were possible and, therefore, we were able to study a wider range of parameters in our experiments.

3.4 Data Collection

This client-server sandbox relies on different tools to collect static and dynamic information during the experiments. A processing of the Android manifest for

[2] http://www.malgenomeproject.org/.
[3] https://play.google.com/store.
[4] http://www.appsapk.com/.
[5] http://contagiodump.blogspot.ca/.

permissions used, broadcast receivers and intents is performed. Of the dynamic analysis tools used, Taintdroid [13] monitors the sensitive data leakages to public channels. Equally, outgoing SMS and phone calls are recorded, through an instrumented version of the Google Emulator. Finally, the rest of the network traffic is recorded for post-experiment analysis of protocols and requests used.

4 Experiment

For this paper, we performed 13 experiments that are listed in Table 2. For each experiment, we specify the context by showing the value of each of the 4 variable parameters, namely the application usage, the number of actions in user simulation, the rest period after simulation and the network configuration of the sandbox. We regroup experiments with the same variation in context in five different groups, presented below.

Table 2. List of experiments.

Experiment label	Dataset	Application usage	Number of actions in user simulation	Rest period	Network context
E1	D1	Install	0	5 min	Full network
E2	D1	Start	0	5 min	Full network
E3	D1	Simulation	5000	5 min	Full network
E4	D2	Install	0	5 min	Full network
E5	D2	Start	0	5 min	Full network
E6	D2	Simulation	5000	5 min	Full network
E7	D2	Simulation	5	5 min	Full network
E8	D2	Simulation	50	5 min	Full network
E9	D2	Simulation	500	5 min	Full network
E10	D2	Simulation	5000	1 min	Full network
E11	D2	Simulation	5000	5 min	No network
E12	D2	Simulation	5000	5 min	DNS
E13	D2	Simulation	5000	5 min	DNS + TCP

Representing a preliminary study [3], group G1 comprises experiments E1, E2 and E3, with application usage as a variation parameter, as shown on Table 3. In this group, an application was only installed in experiment E1, was installed and launched with no other interaction in experiment E2 and finally, was installed, started and simulated with 5000 random actions. These experiments E1, E2 and E3 are all configured with a 5-min rest period, a full network configuration and performed with dataset D1.

Table 3. Context of G1.

Experiment label	Application usage	Number of actions	Rest period	Network context	Dataset
E1	Install	0	5 min	Full network	D1
E2	Start	0			
E3	Simulation	5000			

In the beginning, experiments in G1 were selected to constitute a proof-of-concept that variations of parameters would lead to a means of comparing malware behaviour. Since most of the aforementioned dynamic analysis tools are researching ways to expand their simulation engine, a difference in the level of simulation appeared as a natural starting point for our first experiments.

In second place, as displayed on Table 4, a second group G2 with experiments E4, E5 and E6 aims at confirming results from group G1. Therefore, group G2 is identical to group G1 (where E1 $\equiv$ E4, E2 $\equiv$ E5 and E3 $\equiv$ E6) for all variation parameters but a different dataset, D2.

Table 4. Context of G2.

Experiment label	Application usage	Number of actions	Rest period	Network context	Dataset
E4	Install	0	5 min	Full network	D2
E5	Start	0			
E6	Simulation	5000			

As a logical next step, group G3 variable parameter is the number of actions performed in user simulation. As shown on Table 5, G3 regroups experiments E7, E8, E9 and E6, where all perform user simulation, but with a number of actions respectively set to 5, 50, 500 and 5000 actions. Again, for all experiments in G3, there is a 5-min rest period after user simulation, a full network configuration and samples are from dataset D2.

Table 5. Context of G3.

Experiment label	Application usage	Number of actions	Rest period	Network context	Dataset
E7	Simulation	5	5 min	Full network	D2
E8		50			
E9		500			
E6		5000			

Next, as displayed in Table 6, group G4 is composed of E10 and E6, with the length of the rest period as a variation parameter. Experiment E10 configuration is a 1-min rest period, while E6 has a 5-min rest period. The rest of the context is the same for both experiments, namely both perform user simulation with 5000 actions in a full network configuration on dataset D2.

Table 6. Context of G4.

Experiment label	Application usage	Number of actions	Rest period	Network context	Dataset
E10	Simulation	5000	1 min	Full network	D2
E6			5 min		

Finally, the last group, G5, tests applications in a different network context. As shown on Table 7, G5 includes experiments E11, E12, E13 and E6, where E11 has no network access, E12 evolves in a network where there is only a DNS active[6], E13 has not only a DNS server active, but also completes TCP handshake connections[7] and finally, where E13 has a full network configuration. The rest of the variation parameters are equals for all experiments: user simulation with 5000 actions is performed, a 5-min rest period occurs after simulation and all applications are from dataset D2.

Table 7. Context of G5.

Experiment label	Application usage	Number of actions	Rest period	Network context	Dataset
E11	Simulation	5000	5 min	No network	D2
E12				DNS Only	
E13				DNS + TCP	
E6				Full network	

5 Results

Results from each experiment are compared using metrics presented in this section. Failure rate of experiment in Sect. 5.1, SMS activity in Sect. 5.2, data

[6] All IP traffic other than DNS queries is sinkholed to an IP not on the network.

[7] All IP traffic other than DNS queries is sinkholed to an IP for which no ports are open but fake TCP SynAck packets are sent back in response to any TPC Syn (thus properly completing the 3-way handshake).

leaks in Sect. 5.3 and network traffic (DNS and HTTP requests) in Sect. 5.4, help to give a picture of different application behaviour in different contexts.

All results that have a graphic figure associated display the metric calculated on each source of the dataset used, except for the last column, labelled 'Total'. This column represents the metric on all samples as a whole, often qualified as the overall metric.

5.1 Failure Rate

The first metric we use is the failure rate of an experiments, which is defined by the number of runs that do not successfully complete a scenario. To qualify as a success, all scenario steps of a run must be completed, otherwise, the run is considered a failure (a step may not be completed, may stop or fail to execute in the allotted time).

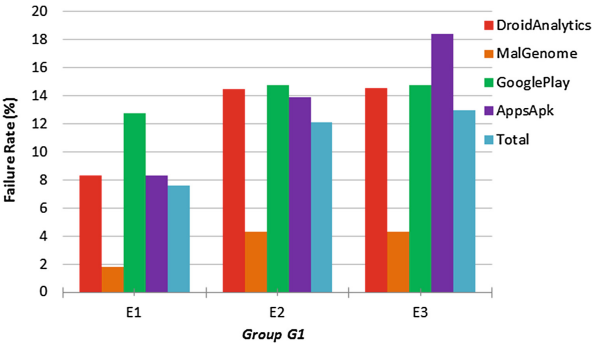

Fig. 1. Failure rate for group G1 (application usage on D1) [3].

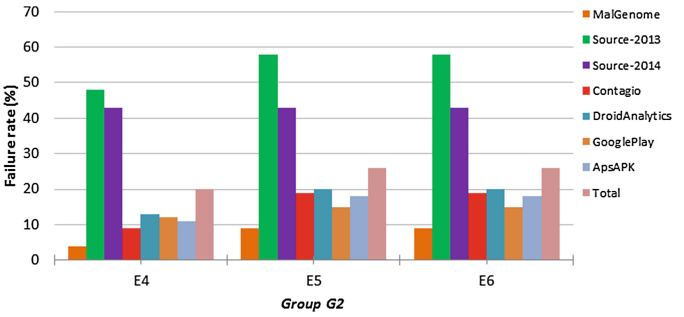

Fig. 2. Failure rate for group G2 (application usage on D2).

For group G1, overall failure rate is at 7.7% for experiment E1, while E2 is at 11.8% and experiment E3 fails in 12.9% of the cases, as it is shown on Fig. 1. The interesting point is that simulation do not raise the failure rate significantly, as only a small percentage (1.1%) of applications are launched successfully, but are not able to pass through the user simulation phase.

Within group G2, with which we sought to confirm our results, overall failure rate for experiment E4 is at 20%, and is slightly higher, at 26% for both experiments E5 and E6, as is shown on Fig. 2. Failure rates recorded for experiments in group G2 confirm observations from experiments in G1, for all sources. User simulation do not modify the failure rate when compared to starting the application, as experiments E5 and E6 show a similar failure rate, while the number recorded for experiment E4 is significantly lower.

As experiments in group G3, G4 and G5 all have the same usage application (simulation), failure rates are almost identical to numbers presented for experiment E6.

Therefore, in all experiments, simulation of user actions does not significantly increase failure rate when compared to starting an app, but launching an application does when compared to only installing it. Moreover, varying the other parameters (number of action, rest period and network configuration) does not affect failure rate.

5.2 Sending SMS

First, for experiments in group G1, we observe that only malicious applications send unauthorized SMS, and that text activity increases with application usage, as displayed on Fig. 3. Indeed, not even one application sent an SMS in experiment E1. However, in experiment E2, the overall percentage of applications sending SMS is at 0.74% and is up to 1.2% for some sources. Then, in experiment E3, the overall percentage is 4.60% and for some sources, the percentage is as high as 8.8% [3].

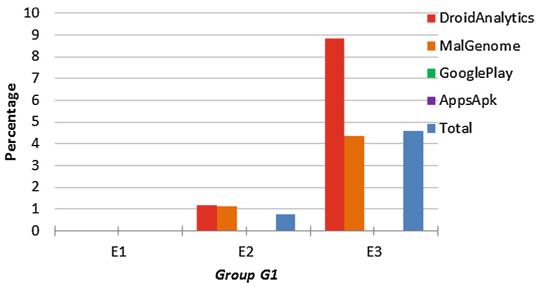

Fig. 3. Percentage of apps sending SMS for Group G1 (application usage on D1) [3].

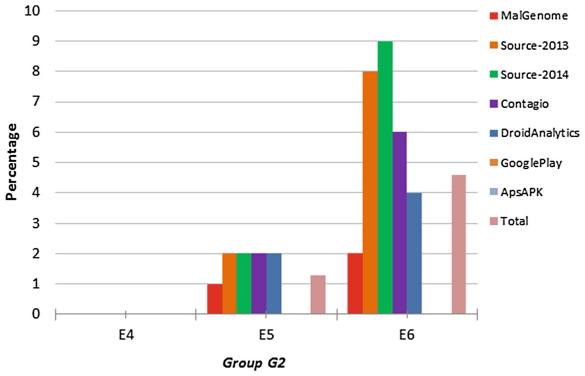

Fig. 4. Percentage of apps sending SMS for Group G2 (application usage on D2).

With group G2, we again seek confirmation of what is observed in experiments from group G1. As displayed on Fig. 4, only malicious applications send unauthorized SMS. Also, when application usage is installation only, in experiment E4, text activity is null. Only in experiment E5 can we see a beginning of text activity, as 1.3% of all applications send SMS. With experiment E6, the percentage climbs to 4.6% of all applications and reaches up to 11% of applications for some sources. Therefore, our preliminary observations for experiments in group G1 are confirmed. Only malware send unauthorized SMS and text activity is most often triggered by user simulation.

Following is group G3, to verify how the number of actions in user simulation would influence the percentage of applications sending SMS. As is shown on Fig. 5, the percentage of applications sending SMS with 5 actions (that is, experiment E7) is no higher than 1.4%, but climbs at 2.9% in experiment E8 (with 50 actions) and 3.9% with 500 actions, in experiment E9. Although experiment E6, with 5000 actions, has a percentage of 4.6%, if we look at percentages per source, for all sources but Source-2014, the percentage are identical. Malware from Source-2014 show a higher percentage of 9% in experiment E6 compared to 7% in experiment E9. Therefore, a high number of actions reveals text activity better, and it seems that, in general, 500 actions is sufficient to detect most of activity.

Based on the previous observations for text activity, when looking at experiments in group G4, we see that a different rest period after simulation does not have a significant incidence on the percentage of applications that send SMS. Whether the rest period is set to 1 or 5 min, the percentage of applications sending SMS is the same for each source, as demonstrated on Fig. 6. It is expected, given that user simulation is such a factor to detect text activity, that a rest period of any length of time do not raise the percentage of applications sending SMS. As part of our future work, we intend to confirm this conclusion with new experiments with longer rest periods.

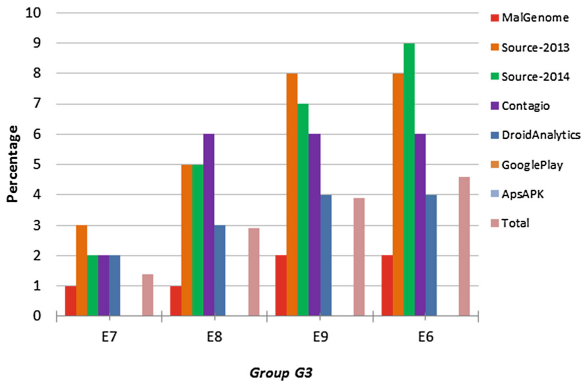

Fig. 5. Percentage of apps sending SMS for Group G3 (number of actions).

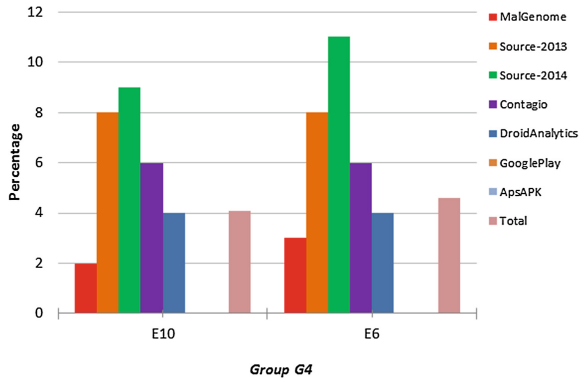

Fig. 6. Percentage of apps sending SMS for Group G4 (rest period).

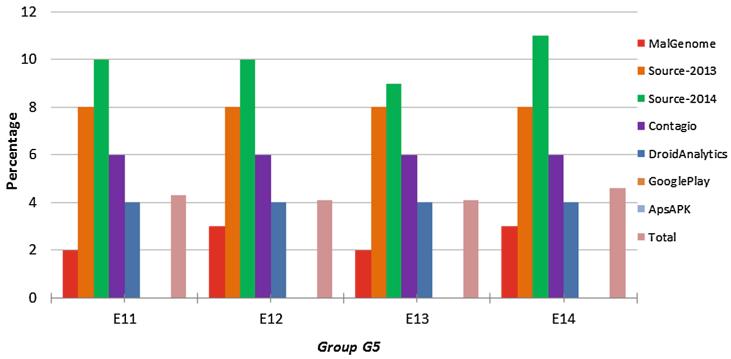

Fig. 7. Percentage of apps sending SMS for Group G5 (network configuration).

Moreover, in experiments of group G5, the network configuration has no incidence on text activity. Indeed, as shown on Fig. 7, the percentage of applications sending SMS are identical in experiments E11, E12, E13 and E6, for all sources except MalGenome and Source-2014, no matter what network configuration is selected and varies only slightly for Malgenome and Source-2014. As for both sources with variations, since the percentages for MalGenome are respectively 2%, 3%, 2% and 3% for experiments E11, E12, E13 and E6 and are respectively 10%, 10%, 9% and 11% for Source-2104, we conclude that the randomness of actions, and not the network configuration, changed the percentages slightly. The problem of randomness is discussed in Sect. 5.5.

Also, other information related to SMS has been recorded in experiments of group G1, to extend the knowledge base on malware using SMS. When text activity is detected, we estimate that 2.5 SMS are sent per application on average. Convergence about SMS numbers and text content could also be observed, as only 23 different numbers and 50 different texts were found in 254 text messages. As shown in Table 8, SMS numbers and content show a recurring pattern. No comparison has been done on these metrics so far, as few apps in our sample turn out to be sending SMS. However, on a larger sample, this information may further help to compare and sort SMS-sending applications. It may also prove valuable when comparing metrics for families of malware.

Table 8. SMS numbers and texts sent by some malware [3].

Malware ID	SMS Number	SMS Text
1	10621900	YXX1, YXX4, YXX2
1	10626213	C*X1, C*X2
1	1066185829	921X1, 921X2, 921X4
2	3353	70 + 224761
2	3354	70 + 224761
3	6005	jafun 806 1656764
3	6006	jafun 806 1575475
3	6008	jafan 806 2237145

5.3 Data Leakages

When data leaks are measured in experiments of group G1, we find that starting an application is necessary but sufficient to discover most applications, as is shown on Fig. 8. Even if the percentage of applications leaking data is slightly higher in experiment E3, it is not significantly so. Moreover, data leakages are registered in malware as well as legitimate applications, in very different proportions. Of all applications leaking data, 91% were malware while only 9% were legitimate applications.

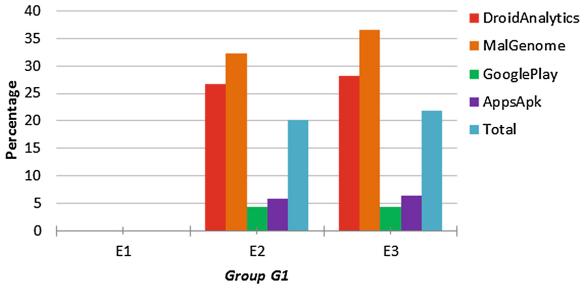

Fig. 8. Percentage of apps leaking sensitive data for Group G1 (application usage on D1) [3].

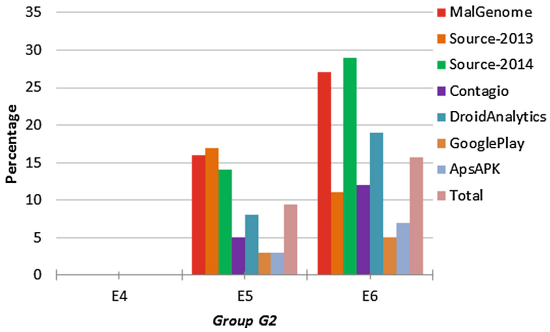

Fig. 9. Percentage of apps leaking sensitive data for Group G2 (application usage on D2).

First looking at experiments in group G2 for confirmation, we are able to confirm partially our observations. Indeed, as shown on Fig. 9, malware are far more prone to leak data (89% of all leakages are registered on malicious applications). Experiments in group G2 also confirm that it is necessary to launch the application to register leakages, but contrary to experiments in G1, comparison between experiment E5 and E6 shows that user simulation helps to find more data leaks. For all sources but two, percentages are higher by at least 4% in experiment E6. For source GooglePlay, the percentage is also higher in E6, but only by 2%. Contrary to what was observed for group G1, we conclude that, although a large part of the data leakages are caught when starting the application, user simulation is in order for a better result. We discuss the odd result in E2 for Source-2013 (more leaks are observed without user simulation, E5 vs E6) in Sect. 5.5.

With this new observation, it is interesting to look at experiments in group G3, where the number of actions in user simulation is the variation parameter. Number of actions starts at 5 actions in experiment E7 and increases to 50, 500 and 5000 actions in experiments E8, E9 and E6. As displayed on Fig. 10, the higher percentages are reached in experiment E6, for all sources.

Also interesting are the numbers of experiments in group G4, where the length of the rest period is changed. A longer rest period does not influence the failure rate (Sect. 5.1) or text activity (Sect. 5.2), the first two metrics in this article. However, for data leaks, this parameter does change the percentage of applications leaking data. As shown on Fig. 11, percentages for all sources are equals to or higher with a rest period of 5 min (experiment E6) instead of 1 min (experiment E10). The percentage of data leaks for legitimate application is either equals to (for GooglePlay samples) or higher by 1% (for AppsApk samples), but the difference is between 1% and 6% for malicious applications. Therefore, a longer rest period after user simulation is susceptible to help discover more leaky applications.

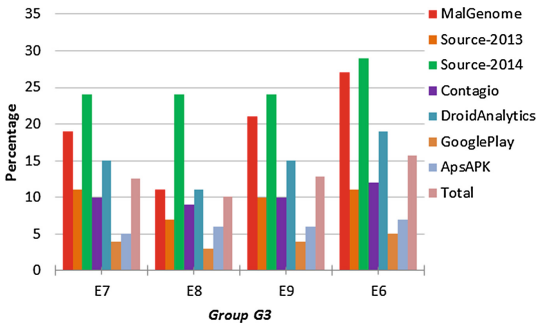

Fig. 10. Percentage of apps leaking sensitive data for Group G3 (number of actions).

Finally, data leakages is measured in experiments of group G5, to show the effects of network configuration on this metric. In Fig. 12, leaks of sensitive information are higher when a full network configuration is active (experiment E6), for all sources of applications except one. For that source, Source-2013, the percentage is higher in experiment E13 (17% instead of 11% in E6), which will be discussed in Sect. 5.5. For some other sources (GooglePlay, AppsAPK, DroidAnalytics and Contagio), the percentage of leaky applications are only slightly different (2% or less) between the experiment E13 and E6. However, for applications from MalGenome and Source-2014, there is respectively a 9% and 12% difference between E13 and E6, that allows us to conclude that a full network configuration is best to find data leaks.

5.4 Network Traffic

During an experiment, network traffic is closely monitored, to gather information about malware servers, addresses they may contact, etc. For all experiments, we analyze DNS and HTTP requests that are presented in the following sections.

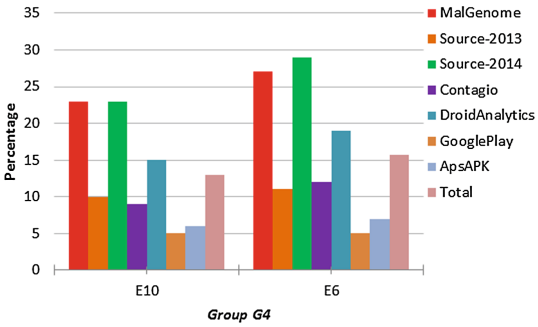

Fig. 11. Percentage of apps leaking sensitive data for Group G4 (rest period).

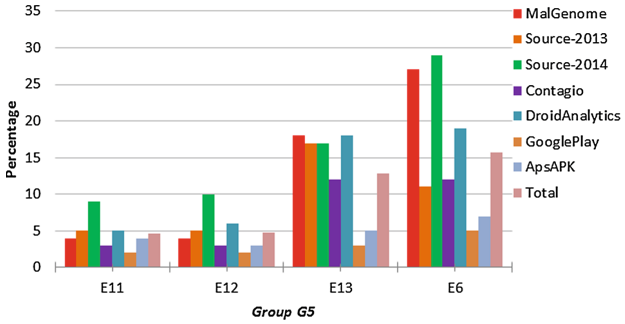

Fig. 12. Percentage of apps leaking sensitive data for Group G5 (network configuration).

DNS Requests. The metric considered for DNS requests is the average number of requests made by an app. We first look for confirmation of results observed in experiments of group G1, in which an overall average of 0.002 request is made in experiment E1, of 2.62 requests in E2 and of 4.86 requests in E3, as displayed on Fig. 13. These results show that the average number of DNS requests is higher in experiment E3, that is when user simulation is used. As is shown on Fig. 14, results from experiments in group G2 show that the same conclusions can be reached. In experiment E4, the average number of DNS requests is 0.01 request, number that climbs to an average of 0.4 request in experiment E5 and reaches a high mark of 1.3 request per application in experiment E6.

Also, we observe in experiments of group G1 that malware from all sources have a higher average of DNS requests than legitimate applications. However, in experiments of group G2, legitimate applications from both sources have a higher average than applications from malware sources, except for Source-2014. Average number of DNS requests for applications of Source-2014 are at 2.33 requests per application, while applications from GooglePlay are at 2.06 requests per application and apps from AppsAPK are at 1.38 request per application. Malware from all other sources are showing an average below 1.2 DNS request per application,

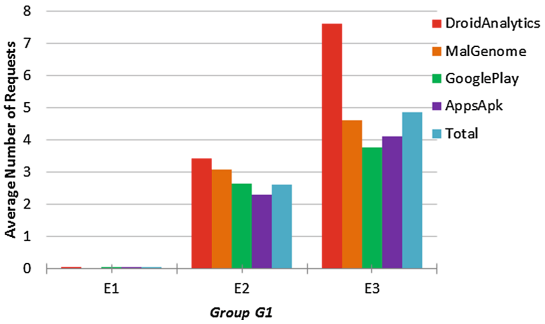

Fig. 13. Average number of DNS requests per application, for Group G1 (application usage on D1) [3].

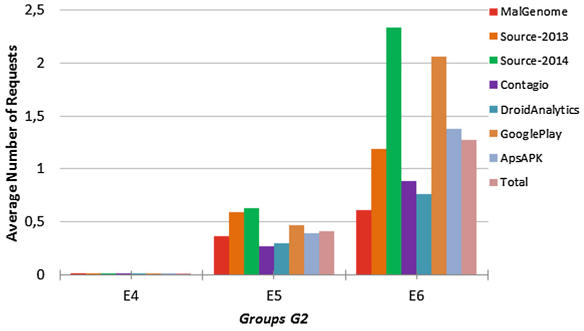

Fig. 14. Average number of DNS requests per application, for Group G2 (application usage on D2).

making their averages lower than the average of legitimate applications. Hence, our previous observation is not confirmed by the new experiment.

Moreover, when looking at results from experiments in group G3, displayed on Fig. 15, where the number of actions in user simulation is the variation parameter, the highest average is always obtained in experiment E9 or E6, respectively configured with 500 and 5000 actions. Therefore, we can say that a high number of actions is required to get a significant average for DNS requests, but the number of actions, when sufficiently high, reaches a critical point where more actions will not significantly increase the average number of DNS requests made by an application.

Now, for experiments of group G4, where the rest period after user simulation was changed, the longer rest period of 5 min in experiment E6 yielded the highest average in general. As shown on Fig. 16, the overall average of DNS requests for experiment E10 is 1.1 request while experiment E6 has an overall average of 1.3 request.

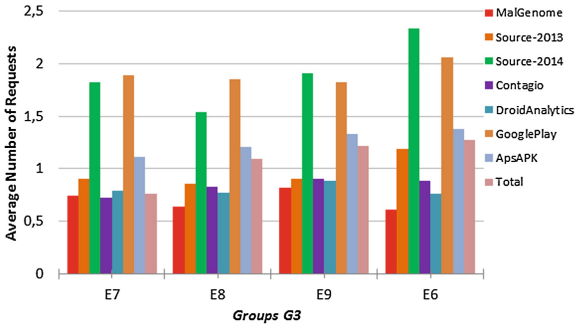

Fig. 15. Average number of DNS requests per application, for Group G3 (number of actions).

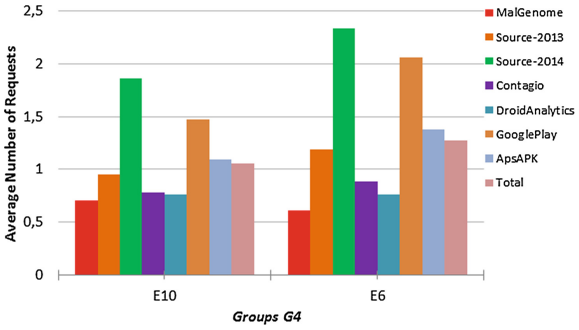

Fig. 16. Average number of DNS requests per application, for Group G4 (rest period).

Finally, in experiments of group G5, the variation parameter is the network configuration. Unsurprisingly, experiment E11 has the lowest average number of DNS requests, as shown on Fig. 17. Experiment E12 is comparable to experiment E11, with slightly, but not significantly, higher averages for all sources. With experiment E6, however, the highest average of DNS requests was reached for all sources, by a large margin for some sources. Therefore, we can conclude that a full network configuration is best to gather the maximum number of DNS requests. Lower results from experiment E13 will be discussed in Sect. 5.5.

HTTP Requests. In this section, the metric is similar than for DNS requests, namely the average number of HTTP requests per application. In experiments of group G1, shown on Fig. 18, experience E3, with user simulation, shows an average of 12.2 requests per application, while experiment E2 (starting the application) displays an average of 6.11 requests. No HTTP requests are recorded in experiment E1, where an application is installed only. As a confirmation of our results, experiments in group G2 do not show any HTTP requests in experiment E4 and the average number of HTTP requests for experiment E5 is 0.63

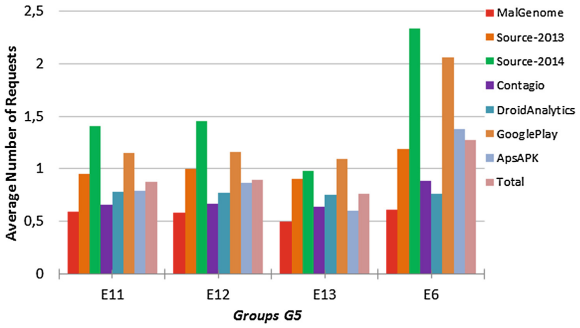

Fig. 17. Average number of DNS requests per application, for Group G5 (network configuration).

request. The overall average climbs to 4.18 requests in experiment E6, confirming that a proper user simulation helps to record more HTTP requests. Results are presented in Fig. 19.

In our preliminary paper, we mention that a high average may be symptomatic of a malware, as a significantly higher average was recorded for a malware dataset. In experiment of group G2, the highest average recorded for a dataset is 11.72 requests per application, while the second- and third-highest are respectively standing at 8.23 and 5.48 requests per application. Since the highest average number of HTTP requests per application is registered on the GooglePlay dataset, we can no longer conclude that a high average hints at malicious applications.

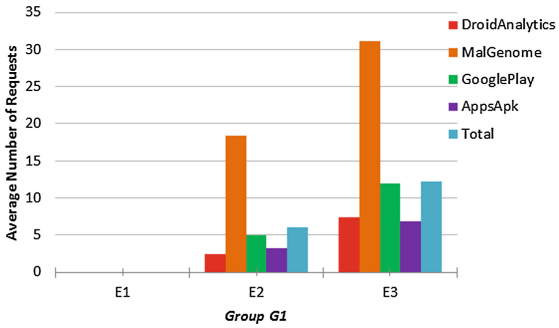

Fig. 18. Average number of HTTP requests per application, for Group G1 (application usage on D1) [3].

Furthermore, when looking at the number of actions in user simulation (group G3) in Fig. 20, the average number of requests for experiment E7, E8 and E9 is respectively 2.3, 2.2 and 2.6 requests per application. Therefore, whether we

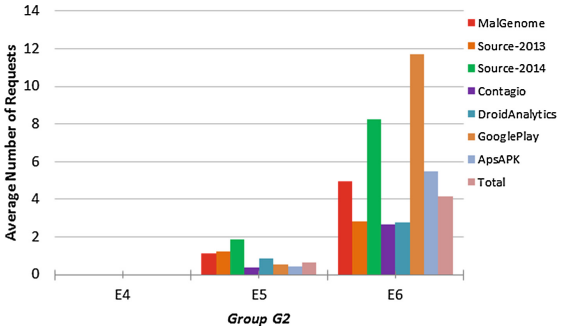

Fig. 19. Average number of HTTP requests per application, for Group G2 (application usage on D2).

perform 5, 50 or 500 actions, the variation does not have a significant effect on the results. However, the average number of requests in experiment E6 reaches 4.18 requests per application, a significantly higher average than the other 3 experiments in group G3. So, not only is user simulation essential, but also a high number of actions helps to trigger more HTTP requests. Also, in experiments of group G4, displayed in Fig. 21, an overall average of 2.39 HTTP requests are sent, per application, when using a rest period of 1 min. That average climbs to 4.18 requests per application when using a rest period of 5 min, allowing us to conclude that a longer rest period increases the average number of HTTP requests per application.

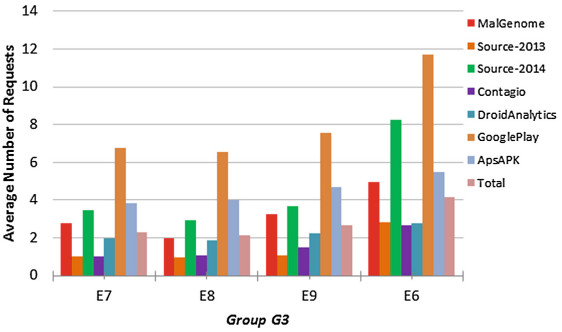

Fig. 20. Average number of HTTP requests per application, for Group G3 (number of actions).

Finally, when network configuration is changed in experiments of group G5 (shown on Fig. 22), no HTTP request is recorded with scenario in experiments E11 and E12, as expected. When enabling TCP handshake completion, in experiment E13, an average of 1.41 HTTP request per application is recorded, while

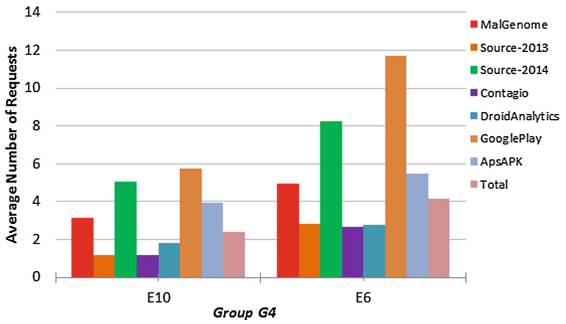

Fig. 21. Average number of HTTP requests per application, for Group G4 (rest period).

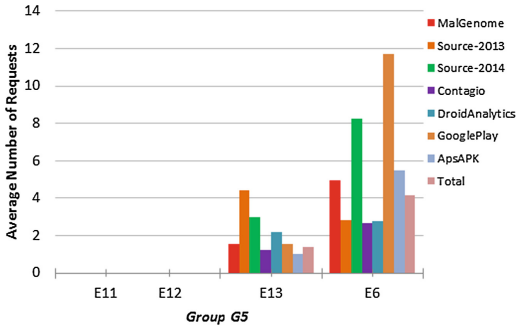

Fig. 22. Average number of HTTP requests per application, for Group G5 (network configuration).

the average is 4.18 requests in experiment E6. Unsurprisingly, the better network configuration is, the higher the average number of HTTP requests is.

5.5 Odd Results

Through the result analysis, we observed some odd results. This section tries to partially explain those oddities. Most odd results are probably explained by the fact that we use a random sequence of action to stimulate the application and there is a new sequence for every run. Thus, an action sequence may fail to trigger the malicious behavior in a rich execution context while another action sequence will have succeeded in more limited context (e.g., Fig. 6 indicates in E13 vs E12 for MalGenome dataset that some apps sent SMS with only a DNS on the network but they did not with a DNS+ a fake TCP server). Section 6 will provide insight regarding how we plan to address these shortcomings.

Another explanation may apply to a subset of those odd results and is worth considering. Figure 10 (E13 vs E6) indicates that for the source-2013 dataset, some apps leaked information with a network config consisting of a DNS server

and a Fake TCP handshake server (E13) but not with a full Internet access (E6). This could be explained by the different random action sequences as above, but the discrepancy is important (6% difference) so we consider a second explanation. Let's assume that the server the malware tries to contact on the Internet is unavailable (e.g., C&C has been taken down), the malware would not leak the information (connection cannot be established with destination). On the other hand, with a fake internal server the communication can be established (fake TCP handshake) and the malware could leak the information as the first packet sent to the fake server (before noticing that the server is not responding past the TCP handshake) and we would see it in our simulated network. Further investigation is required, but, although running malware in a live network is usually considered better for dynamic analysis, it might be a good idea for a sandbox to integrate a fake server that take over whenever the online server does not respond.

6 Future Work

Following the discussion in Sect. 5.5, we will work towards a repeatable sequences of user actions and perform our current set of experiments with this change. We will also look into testing a longer rest period, to confirm observations on text activity in Sect. 5.2.

Moreover, as stated in our preliminary paper [3], we would like to introduce new variation parameters, like the Android OS version and types of emulators. It would also be interesting to look at other parameters, like location. Equally, it is our intention to refine our current metrics and add others, to get a clearer picture of analyzed applications.

Finally, increasing our application dataset with more samples is still in our to-do list, as is the classification of malware per families.

7 Conclusion

In this paper, we compare experiments with different contexts, in order to study the influence of such contexts on the behavior of applications from multiples sources. To achieve this goal, we designed 13 experiments, separated in 5 different groups, where each group was assigned a variation parameter and a dataset. These parameters currently are application usage, number of actions in simulation, rest period after simulation and network configuration.

Then, experiments were executed in a virtual sandbox with dynamic analysis revolving around different metrics like failure rate, SMS activity, data leakages, DNS and HTTP activity. With these metrics, we compare each experiment and its variation parameters, to help increase our knowledge of application behavior.

In general, variation in application usage will influence all metrics, as shown in Table 9. A different number of actions will influence SMS, DNS and HTTP activity, but will have no incidence on failure rate and the percentage of applications leaking sensitive data. Finally, both network configuration and rest period

Table 9. Influence of variations on metrics.

	Application usage	Number of actions	Network configuration	Rest period
Failure rate	✔			
SMS activity	✔	✔		
Data leaks	✔		✔	✔
DNS activity	✔	✔	✔	✔
HTTP activity	✔	✔	✔	✔

will modify DNS and HTTP activity, as well as the percentage of application leaking data, but will not influence the failure rate or SMS activity.

Results also demonstrate that simulation with a high number of actions, a full network configuration and a longer rest period will yield best results on metrics they have incidence on, with leaks of sensitive data being an exception. We intend to continue on this path, refining our current work to precise our global comprehension of Android application behavior.

References

1. PulseSecure: 2015 Mobile Threat Report. Technical report, Pulse Secure Mobile Threat Center (2015)
2. Blasing, T., Batyuk, L., Schmidt, A.D., Camtepe, S.A., Albayrak, S.: An android application sandbox system for suspicious software detection. In: Proceedings of the 5th International Conference of Malicious and Unwanted Software, pp. 56–62 (2010)
3. Boileau, C., Gagnon, F., Poisson, J., Frenette, S., Mejri, M.: A comparative study of android malware behavior in different contexts. In: Proceedings of the 13th International Joint Conference on e-Business and Telecommunications, vol. 1, pp. 47–54. DCNET (2016)
4. Dunham, K., Hartman, S., Morales, J.A., Quintans, M., Strazzere, T.: Android Malware and Analysis. Auerbach Publications, Boston (2014)
5. Bayer, U., Habibi, I., Balzarotti, D., Kirda, E., Kruegel, C.: A view on current malware behaviors. In: LEET (2009)
6. Bayer, U., Kruegel, C., Kirda, E.: TTAnalyze: a tool for analyzing malware (2006)
7. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using CWSandbox. IEEE Secur. Priv. **5**, 32–39 (2007)
8. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.: DREBIN: effective and explainable detection of android malware in your pocket. In: Proceedings of the 2013 Network and Distributed System Security (NDSS) Symposium (2014)
9. Arzt, S., Rasthofer, S., Christian Fritz, E.B., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P.: FlowDroid: precise context, flow, field, object-sensitive and lifecyle-aware taint analysis for android apps. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 259–269 (2014)

10. Gonzalez, H., Stakhanova, N., Ghorbani, A.A.: DroidKin: lightweight detection of android apps similarity. In: Tian, J., Jing, J., Srivatsa, M. (eds.) SecureComm 2014. LNICST, vol. 152, pp. 436–453. Springer, Cham (2015). doi:10.1007/978-3-319-23829-6_30
11. Zheng, M., Sun, M.: DroidAnalytics: a signature based analytic system to collect, extract, analyze and associate android malware. In: Proceedings of 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 163–171 (2013)
12. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: CrowDroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 15–26 (2011)
13. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. (TOCS) **32** (2014)
14. Rastogi, V., Chen, Y., Enck, W.: AppsPlayground: automatic security analysis of smartphone applications. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, pp. 209–220 (2013)
15. 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: NDSS (2012)
16. Eder, T., Rodler, M., Vymazal, D., Zeilinger, M.: Ananas-a framework for analyzing android applications. In: 2013 Eighth International Conference on Availability, Reliability and Security (ARES), pp. 711–719. IEEE (2013)
17. Neugschwandtner, M., Lindorder, M., Fratantonio, Y., van der Veen, V., Platzer, C.: ANDRUBIS - 1,000,000 apps later: a view on current android malware behaviors. In: Proceedings of the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security, pp. 161–190 (2014)
18. Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., Hoffmann, J.: Mobile-sandbox: having a deeper look into android applications. In: Proceedings of the 28th Symposium on Applied Computing, pp. 1808–1815 (2013)
19. Reina, A., Fattori, A., Cavallaro, L.: A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In: Proceedings of 6th European Workshop on Systems Security (2013)
20. van der Veen, V., Bos, H., Rossow, C.: Dynamic analysis of android malware. Internet & Web Technology Master thesis, VU University Amsterdam (2013)
21. Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: PScout: analyzing the android permission specification. In: Proceedings of the 2012 ACM conference on Computer and communications security, pp. 217–228. ACM (2012)
22. Sasnauskas, R., Regehr, J.: Intent fuzzer: crafting intents of death. In: Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA), pp. 1–5. ACM (2014)
23. Ye, H., Cheng, S., Zhang, L., Jiang, F.: Droidfuzzer: fuzzing the android apps with intent-filter tag. In: Proceedings of International Conference on Advances in Mobile Computing & Multimedia, p. 68. ACM (2013)
24. Gagnon, F., Lafrance, F., Frenette, S., Hall, S.: AVP-an android virtual playground. In: DCNET, pp. 13–20 (2014)
25. Gagnon, F., Poisson, J., Frenette, S., Lafrance, F., Hallé, S., Michaud, F.: Blueprints of an automated android test-bed. In: Obaidat, M.S., Holzinger, A., Filipe, J. (eds.) ICETE 2014. CCIS, vol. 554, pp. 3–25. Springer, Cham (2015). doi:10.1007/978-3-319-25915-4_1