Lumus: Dynamically Uncovering Evasive Android Applications

Vitor Afonso[1,5], Anatoli Kalysch[4] Tilo Müller[4], Daniela Oliveira[3], André Grégio[2], and Paulo Lício de Geus[1]

[1] Institute of Computing, University of Campinas, Brazil,
pgeus@unicamp.br
[2] Department of Informatics, Federal University of Parana, Brazil,
gregio@inf.ufpr.br
[3] Florida Institute for Cybersecurity Research, University of Florida, USA,
daniela@ece.ufl.edu
[4] Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
anatoli.kalysch@fau.de, tilo.mueller@cs.fau.de
[5] Content Keeper, Australia
vitor.afonso@contentkeeper.com

Abstract. Dynamic analysis of Android malware suffers from techniques that identify the analysis environment and prevent the malicious behavior from being observed. While there are many analysis solutions that can thwart evasive malware on Windows, the application of similar techniques for Android has not been studied in-depth. In this paper, we present *Lumus*, a novel technique to uncover evasive malware on Android. *Lumus* compares the execution traces of malware on bare metal and emulated environments. We used *Lumus* to analyze 1,470 Android malware samples and were able to uncover 192 evasive samples. Comparing our approach with other solutions yields better results in terms of accuracy and false positives. We discuss which information are typically used by evasive malware for detecting emulated environments, and conclude on how analysis sandboxes can be strengthened in the future.

1 Introduction

Malicious applications are a major threat to Android users, as they may steal sensitive data, send SMS messages to premium numbers, and manipulate mobile banking transactions [8]. The analysis of an apps behavior is crucial for protecting mobile devices, e.g., to analyze applications before they are published in app stores. Several approaches have been proposed for Android app analysis [16, 27, 6, 24, 7], including the classification into malicious or benign [28, 25, 1, 30, 33, 31, 5].

Analysis techniques are typically divided into static and dynamic approaches. Static approaches become less effective when dealing with highly obfuscated samples [3, 20, 10], or with samples that obtain and execute code at runtime [17, 23]. Dynamic approaches are able to run obfuscated samples and samples that

fetch code dynamically, but can be evaded by malware that employs anti-analysis techniques. Anti-analysis techniques are used to identify emulated environments, and eventually to change an apps behavior to evade detection.

Researchers have identified several anti-analysis techniques that are used by Android apps to distinguish real from analysis environments ([18, 26, 9, 29, 22]). Emulation is used by most analysis systems due to scalability. Hence, an alternative for analyzing apps without being evaded by common anti-analysis techniques is using real devices, as, for example, done by BareDroid [21] and Bolt [7]. However, by studying evasive malware samples in-depth, researchers can also identify ways to make emulated systems resilient to evasive malware.

In this paper, we present *Lumus*, a technique to identify Android malware that exhibits evasive behavior. We analyze the behavior of malware samples by comparing their execution traces on actual devices and emulators. While systems proposed in the literature also identify evasive malware on Windows [15, 12], those techniques cannot directly be applied to Android given the differences between the two operating systems. To demonstrate this, we created detectors based on the Windows techniques proposed in Disarm [15] and Barecloud [12], and compared their results with *Lumus*, which obtained better results.

We analyzed 1,470 Android malware samples selected from different families and identified 192 samples that exhibit evasive behavior. We manually inspected a subset of the detected malware to identify how they evade dynamic analysis. To compare our technique with other solutions, we randomly selected 50 samples from different families and manually analyzed them to validate our results.

2 Approach

To identify evasive malware, systems proposed in non-mobile literature [15], [12] usually compare behavior profiles using distance equations or hierarchically structuring them to compute their similarity—they use system call traces as input data and focus on Windows malware. However, Android malware in general executes far fewer actions than Windows malware and many of the proposed approaches for detecting evasive malware require a minimum number of actions from malware for the detection process (e.g., Disarm [15] requires at least 150 actions). Furthermore, many Android malware are repackaged apps that also perform benign behavior. Therefore, depending on the malware family, infectious actions that make use of anti-analysis features in Android malware can be a very small subset of all possible behaviors, but might just as well comprise most of the app's behavior.

Our approach to identify actual differences of behaviors in Android malware running in bare metal and emulated environments that are not related to idiosyncrasies between these two environments is as follows. First we try to identify the cause of each observed different action through information that is easily obtained in Android, but not for Windows programs. More precisely, we track: (i) executed methods of the app under analysis; (ii) methods from the framework called by the methods identified in (i); (iii) system calls invoked by the app; (iv) interaction of the apps with functionalities that create threads or indirectly

change their execution flow; (v) information provided by the system regarding events that stop the execution of apps; and (vi) information about external stimuli. With all this information, we can trace back the call sequence that led to the behavior observed only in bare metal, identifying the entry point that originated this sequence and possibly the external stimulus that caused it. By comparing the sequence obtained from the bare metal system to the behavior observed in the emulator we can identify the reason for the divergence (e.g., some event not being generated, a difference in some method's execution, the analysis time ending in one of the environments, or the system stopping the app for some reason).

Lumus' work flow is detailed in Figure 1 in form of a process diagram. The events from (i) - (vi) are collected in form of log files for the bare metal and emulator executions. While the bare metal device is brought back to a consistent state the comparison engine uses the collected information to create the divergence report by comparing the information in (i) - (vi) for the emulator and bare metal executions.

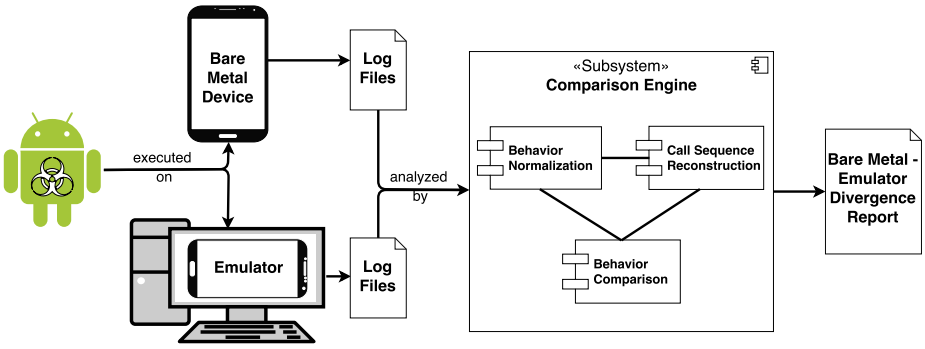

Fig. 1. Abstract process diagram of our approach. For simplicity reasons the snapshot reconstruction subsystem for the bare metal device has been left out of the loop.

3 Behavior Representation

We represent the behavior of an app in a given analysis environment as a set of actions observed during its execution. Each action is a tuple a represented as $a = (action\_type, operation, argument)$, where $action\_type$ is one of $\{Network, File, Intent, Exec, Phone, Dex, Billing, Multimedia\}$. Details about the action types and its operations and arguments are presented in the Appendix.

3.1 Behavior normalization

Filenames written by apps may be randomly generated, which makes multiple executions of the same application produce different behavior profiles. To overcome

this problem we adopt Disarm's approach [15]: for each sandbox—emulated or bare metal—we identify files that were written in only one instance of this sandbox and consider these as possibly random files. Possibly random files in multiple instances of a sandbox that have the same directory and extension are considered as random. We keep the directory name and extension of these actions but replace the file name by $<RANDOM>$. We also normalize file paths related to the SD card, as it can be accessed in different ways. Malware may also randomly select contacts registered in the system as destinations of SMS messages.

Therefore, we inspect actions related to sending SMS messages and, if some destination is a contact registered in the system, we replace it by $<CONTACT>$. Additionally, we filter simple actions that are common to most apps, such as writing to the shared memory device or to the logging device. Another group of actions we filter are related to Androids' Webview, an in-app solution to displaying web pages. Since it relies on libraries only available on bare metal devices the behavior will differ in the emulator, leading to false positives in applications using WebViews. Another challenge of the behavioral analysis problem, especially when the app needs to be executed multiple times, is that certain network behavior might only be observed in some of the runs. To address this challenge, during our analysis when some host is accessed in bare metal and the same DNS name is requested in the emulated context, but this request fails, we add the same failed request to the emulated analysis.

4 Evasive Behavior Identification

To identify whether an app is evasive or not, *Lumus* analyzes its execution in a bare metal environment and in an emulated environment, and then compares the monitored behavior for differences. If the behavior in the two environments is different, *Lumus* identifies the root cause for the divergence, which can be: (i) a variation in the code path executed or (ii) some event that prevented the app from continuing executing in the emulated environment. To increase the app code coverage during dynamic analysis, *Lumus* generates stimuli in the form of GUI interactions and Intents, which can be used to start activities or receivers. *Lumus* provides the same stimuli for both bare metal and emulated environments. Moreover, *Lumus* takes into account non-determinism during app execution, *Lumus* executes each sample three times in each environment.

Let B_i and E_j be the set of actions monitored in the bare metal environment for the ith run and in the emulated environment for the jth run, respectively, with $1 \leq i \leq 3$ and $1 \leq j \leq 3$. Also, let $B = \bigcup_{i=1}^{3} B_i$ and $E = \bigcup_{j=1}^{3} E_j$ be the set of all actions executed in bare metal and in emulated environments, respectively. Since we are interested in finding apps that hide their actions during analysis in the emulated environment, *Lumus* first selects the set A of actions that were only executed in a real device. Thus, $A = B - B \cap E$.

For each action a_k in A, *Lumus* constructs R_k, a set with the Android malware traces obtained from bare metal analysis that contain this action. *Lumus* compares each B_l in R_k to every E_j to identify why a_k was not executed in

the emulated analyses. Since *Lumus* tracks when methods begin and end, it can identify the app's method that executed the action we are interested in. At this point, *Lumus* knows the main entry point for the execution of an app obtained through bare metal analysis (B_l), which will be called from now on M_k. This entry point M_k led to the execution of action a_k, and possibly the external stimulus that caused this action. *Lumus* finds the occurrences of M_k in E_j and, if it knows the stimulus that originated it, *Lumus* compares each M_k in E_j with the B_l call sequence that led to a_k. With this comparison, *Lumus* identifies where is the point of divergence of the execution path, i.e., when the emulated system should also execute the action, but instead chose to follow another path. More precisely, *Lumus* identifies which of the following is the cause of the divergence: (i) difference in execution path; (ii) app not responding; (iii) end of analysis; (iv) fatal exception; or (v) entry point not reached. If the reason for the divergence is a different code path executed, *Lumus* considers the app evasive, otherwise, an execution problem.

4.1 Call sequence reconstruction

Lumus records when each thread of the analyzed app enters and leaves its methods, identifying what method performed a given action. It also logs the methods called, so that the analysis trace can be looked back, starting from the method that executed the action, and sequence of method calls that led to this action is identified. This allows the tracking of actions to an entry point interfacing the Android framework, where the capability to observe direct calls to methods no longer exists. Android classes may have several entry points, which may be executed because of commonly creating activities (e.g., *onCreate*), starting services (e.g., *onStart*), starting receivers (e.g., *onReceive*), running tasks (e.g., *run*), and handling of received messages (e.g., *handleMessage*). One possibility would be to compare all executions of the entry point method in the bare metal and emulated systems, but this could lead to wrong results, because of uncertainty. The phenomenon of uncertainty can happen, for instance, if an activity handles different functionalities, all executed through the same entry point. In that case, our approach is to try to identify the source methods from which the execution changed to the entry points, to perform a more precise comparison of the executions. To accomplish this, *Lumus* investigates Intents sent by the app, the use of several methods that cause indirect changes in the execution flow and the use of external stimuli.

To identify Intents that may have resulted in a specific entry point method being executed, we look for Intents sent by the app that match that method. For example, if the method we are analyzing is *ClassA.onStartCommand(Intent, int, int)*, we assume *ClassA* is a service, since *onStartCommand(Intent, int, int)* is one of the entry points of the class *Service* that can be overwritten. As the Android documentation states, this method is called by the system when a client explicitly starts the service by calling *startService(Intent)*. Thus, to find the source that directed the execution to this method we look for actions that start services using *ClassA* as an argument. Finding sources of entry points to activities is similar to finding services. To find the sources that led to the execution of receivers,

however, we need to inspect the intent filters used by the class and find out which Intents sent match these filters (see Section 5).

Another source of control flow changes performed by *Lumus* is the use of the following groups of methods: (i) methods that schedule a class to be invoked after some delay or periodically (e.g., *Timer.schedule* and *ScheduledThreadPoolExecutor.schedule*), which result in the execution of the methods *run()* or *call()* of the destination class, (ii) methods that send messages to its UI thread (e.g., *Handler.sendMes-sageDelayed*), which result in the execution of *handleMessage(Message)*, and (iii) methods that start a new thread, e.g., *Thread.start()*, which result in the execution of *run()* and *AsyncTask.execute(Params...)*, which in turn may result in the execution of different methods (e.g., *doInBackground (Params...)*). To track these control flow changes, *Lumus* employs instrumentation of the Android framework to assign labels to the messages/tasks sent or to the threads created. To do so, we log the control flow changes when the source method is executed and also when the destination methods are executed. With this information, *Lumus* can track the source call of any of these methods.

The last type of interaction that can cause the execution of entry points is external stimuli. *Lumus* also employs instrumentation of the tools used to create the stimuli, identifying when Intents are sent, keys are pressed, and GUI interactions are performed. These Intents are identified as the source of some entry point, similarly to the approach adopted for Intents sent by the app, explained above. Examples of entry points executed by key strokes are *onKey* and *onKeyDown*. For GUI interactions, examples of common entry points executed are *onClick*, *onTouchEvent* and *onItemClick*.

When tracing the sequence of calls that led to some action, a list of subsequences is created. Along with each subsequence the time of the call that created the next subsequence or that executed the action is kept.

4.2 Comparing sequences

After identifying the list of subsequences of method calls that led to the execution of an action in the bare metal environment, *Lumus* needs to compare this list with the list obtained from the analysis within the emulated system to identify the cause of divergence. We hereafter refer to this list of subsequences as *BareSeq* and to the resulting list of subsequences obtained from the emulated environment as *ResEmu*. *Lumus* iterates over each subsequence $SubBare_i$ of BareSeq, comparing it to its counterpart in ResEmu. For each iteration, $action\_time$ is the time when the call that created the next subsequence was executed or the time when the action was performed. Also, let EP_i be the entry point of $SubBare_i$. *Lumus* finds all occurrences of EP_i in ResEmu that have the same origin as in BareSeq. *Lumus* then proceeds to compare EP_i from BareSeq with each instance of EP_i identified in the emulated results.

Given two entry point methods, *Lumus* finds where they begin and end, obtaining the call sequence in this interval. *Lumus* aligns these two sequences, one from BareSeq and other from ResEmu, using a global alignment algorithm. If $SubBare_i$ is the last subsequence of BareSeq, *Lumus* compares the aligned sequences to determine the divergence that prevented ResEmu from reaching the

```
[78] BARE: 'com.adobe.flashplayer_.AdobeFlashCore.onCreate()' -> %'com.adobe.flashplayer_.
       AdobeFlashCore.writeConfig(java.lang.String, java.lang.String)'
     EMU: 'com.adobe.flashplayer_.AdobeFlashCore.onCreate()' -> %'com.adobe.flashplayer_.
           AdobeFlashCore.writeConfig(java.lang.String, java.lang.String)'
...
[85] BARE: 'com.adobe.flashplayer_.AdobeFlashCore.onCreate()' -> 'java.lang.String.indexOf(
       java.lang.String)'
     EMU: 'com.adobe.flashplayer_.AdobeFlashCore.onCreate()' -> 'java.lang.String.indexOf(java
           .lang.String)'
[86] BARE: 'None'
     EMU: 'com.adobe.flashplayer_.AdobeFlashCore.onCreate()' -> 'java.lang.System.exit(int)'
[87] BARE: 'com.adobe.flashplayer_.AdobeFlashCore.onCreate()' -> 'com.adobe.flashplayer_.
       AdobeFlashCore.isOnline()'
     EMU: 'None'
...
[102] BARE: 'com.adobe.flashplayer_.AdobeFlashCore.onCreate()' -> 'com.adobe.flashplayer_.
       FlashVars.<init>()'
      EMU: 'None'
```

Listing 1.1. Excerpt of the alignment of an evasive sample. The comparison extends to methods and method values to mitigate the downsides of signature only comparisons.

call at *action_time*. Otherwise, let $CallNext$ be the method call in $SubBare_i$ that created the next subsequence of BareSeq. If the app did not reach $CallNext$ in ResEmu, *Lumus* compares the aligned sequences to determine the divergence that prevented ResEmu from reaching $CallNext$. However, if the app reached $CallNext$ in ResEmu, *Lumus* obtains the next subsequence of BareSeq, with entry point EP_{i+1}, and finds this entry point in ResEmu by checking for possible destinations of $CallNext$. If *Lumus* is not able to find an equivalent of EP_{i+1} in ResEmu, it is likely that the execution was interrupted before the call performed at $CallNext$ could take effect, so *Lumus* does not consider it an evasion.

When comparing two aligned sequences, we want to identify the reason for their divergence regarding some action executed at time t_i (either a behavior only observed in BareSeq or some call that created the next subsequence of BareSeq and that was not executed in ResEmu). To do so, *Lumus* iterates over the calls in the aligned sequences and when t_i passes, considering the time of the bare metal calls, *Lumus* checks what was the last call in the emulated sequence. There are three possibilities for this last call: (i) a tag indicating the end of the analysis, (ii) a tag indicating that the system killed the app for not responding or for some other error, (iii) a call to some app's or *Lumus* method.

If *Lumus* identifies case (iii), we assume a divergence in code path taken—an indication of evasive behavior. *Lumus* prints the aligned sequences to help an analyst who needs to manually identify what caused the executions to follow different code paths. Conversely, if the last identified call matches cases (i) or (ii), we assume that an execution error occurred, not an evasion. For illustration, we present an excerpt of the output generated for one sample that is evasive in Listing 1.1. To perform sequence alignment, *Lumus* uses the global alignment algorithm provided by the *swalign* library. We chose a global alignment algorithm because we need to have a global understanding of the sequences, as our analysis depends on the alignment reaching the point in the bare metal sequence where

the target action happened. If the aligned sequence does not reach this point, *Lumus* is unable to identify the cause of divergence.

Arguments selection is an important step when using alignment algorithms. The arguments *Lumus* needs to define to calculate the similarity score between two sequences are: (i) m for matches, with $m > 0$; (ii) mi for mismatches, with $mi < 0$; (iii) g_o for opening gaps, with $g_o < 0$; (iv) g_e for extending gaps, with $g_e < 0$. To prevent mismatches during the analysis, we used a high value for $|mismatch|$ in *Lumus*. We also believe that beginnings and endings of methods of the analyzed app are more important in the alignment than other types of calls, so we assign $2 * m$ for matches of this type. Furthermore, since *Lumus* aims at investigating evasive behavior, we want to prioritize gap extensions over gap openings, so $|g_o| > |g_e|$. In the end, the inequality $|mismatch| > m > |g_o| > |g_e|$ guides the definition of arguments.

5 Monitoring system

To track the behavior of the analyzed apps, *Lumus* monitors which apps' methods were executed, which methods were called from them, and which system calls were executed. To monitor system calls, *Lumus* uses a kernel driver that intercepts them. When a system call is executed, the driver registers its arguments and calls the original system call. To obtain information related to the use of Intents, the driver inspects *ioctl* calls that target the binder device. If the operation performed is a binder transaction (BC_TRANSACTION), *Lumus* logs the destination class, method id and arguments passed. To identify which actual method is represented by the method id, *Lumus* examines the corresponding AIDL file in the Android source code.

To monitor the executed methods, *Lumus* leverages the "method trace" functionality of the Android runtime (ART) and instrument *libart*. Every time the execution goes in and out of a method, *Lumus* registers it. Also, when some method is called, *Lumus* logs the source and destination of such action. This allows *Lumus* to also identify Java methods called from native code. To avoid registering too much information, *Lumus* focuses on new UIDs, so it does not track apps that are already installed in the system when it is in a clean state.

When trying to identify the method call that resulted in the execution of some receiver, *Lumus* needs to identify which intent filters are used by this receiver and look for broadcasts sent that match them. Parsing the app's manifest is not enough to obtain all intent filters that were used, since the app can register others at runtime. To overcome this limitation, *Lumus* also tracks all calls to *Context.registerReceiver*.

The use of threads, tasks and message passing between them introduces a level of indirection that prevents us from tracking the execution flow just by looking at method invocations (Section 4). To be able to reconstruct the call sequence in these cases, *Lumus* needs to track the use of threads, tasks, and messages. To do so, *Lumus* generates a random number that is assigned to a thread (when it is created), to a task (when it is scheduled), and to a message (when it is sent). *Lumus* also logs this identifier when they are actually used or

executed, allowing a parser to match each use or execution of these types to their creation. This matching allows us to track the call sequence in these cases. So, for instance, when the method *Timer.schedule*, which schedules a task for repeated fixed-delay execution, is executed, the system generates a random number, logs it and assigns it to the task. Every time this task is executed, the identification number is logged. To correlate the external stimuli with the app behavior (e.g., clicks and broadcasts), *Lumus* needs to know the time at which each stimulus was provided. To achieve this, *Lumus* employs the instrumentation of the `am` and `input` tools used to create these actions.

Analysis environments. When analyzing malware, it is important to make sure that the environment is not infected before the start of the analysis. Performing an analysis in an infected system may result in wrong results, as one piece of malware can influence the behavior of others. To analyze samples in the emulator, we take advantage of the snapshot functionality, which allows us to restore the system to a clean state after every analysis without incurring any boot time. Analyzing malware in real devices is more challenging as we cannot take advantage of snapshots. One possible way to overcome this problem is to restore the state of the device's partitions after every analysis, as performed by Baredroid [21]. However, this approach is time consuming because the system needs to reboot every time it is restored.

The approach adopted in *Lumus* is to maintain the system clean after each analysis as follows. In Android, apps can only write to a very limited set of directories, which includes mainly the app's dir (*/data/data/<PACKAGE-NAME>/*) and the SD card. When the app is uninstalled after analysis, its directory is deleted by *Lumus*. Files written to the SD card can affect the behavior of other apps that might interact with these files. Thus, all files belonging to the SD card are deleted after every analysis.

Many malware target vulnerabilities in the kernel or privileged processes to operate with root privileges. As the */system* partition, corresponding to kernel code, is mounted as read-only by default, even apps that are able to obtain root privilege first need to remount this partition. To prevent it, our kernel driver blocks all system calls that attempt to remount the */system* partition in writing mode. This protection can be bypassed if the malware manages to access the original `mount` system call. However, we only used the system to test our proposed technique to identify evasive malware. If one wants to use a similar system to receive submissions or analyze apps that could potentially target the system, a better restoration process would be necessary. Furthermore, during our experiments our driver did not actually have to block any calls to `mount`, so we believe that bypass through remount was not a problem.

To increase code coverage during dynamic analysis, it is important to provide GUI interactions and to cause activities, services and receivers to execute. However, as we are comparing multiple executions, it is also important that we provide exactly the same interactions, so that the same code paths are exercised, at least until evasive code is reached or some problem stops the app execution. To accomplish this, we use the Droidbot [14] tool to interact with apps. Droidbot

generates random events, including GUI interactions, broadcasts and specific activities. It also registers the exact events generated and is able to replay them from a file instead of randomly generating them. Thus, in our first bare metal execution of each malware, we randomly generate events and save them into a file. In the following bare metal executions and in the emulated analyses, we make Droidbot read the events from the saved file and replay them.

One way that malware can identify analysis environments is by checking which apps are installed in the system. The lack of the Google Play app, for instance, is a strong indication that the device is not used by an actual user. Hence, we installed in the bare metal environment Open GAPPS, a set of basic apps present in all Android systems. Further, we also installed a few very popular apps and created fake contact information. These apps and contact information make the bare metal and emulated systems different, which could, therefore, cause some apps to behave differently, but not because they intend to evade analysis systems. On the one hand, this could possibly lead our technique to identify such samples as evasive, increasing the number of false-positives. On the other hand, preventing such false-positives may result in false-negatives since the bare metal system could also be detected as an analysis system. We chose to use these techniques and risk increasing false-positives instead of risking increasing false-negatives.

6 Evaluation

For our experiments we used Google's QEMU-based Android emulator, the SDKs' Android Virtual Device (AVD) and an LG G2Mini device. Both the bare metal and the emulated Android systems had our modified version of Android 5.1. Each analysis was executed for at most three minutes in the bare metal environment and at most 10 minutes in the emulated environment—our experiments showed that the emulator environment incurred a 3X performance penalty on the analysis. Since we can identify when a divergence in behavior is caused by one analysis system finishing before the other, this difference in execution time has no negative effects on our technique.

To evaluate *Lumus*, we dynamically analyzed a subset of the samples in our malware dataset obtained from VirusShare, Malgenome [32], contagio mobile, AndroMalShare and Drebin [1]. To select this subset we first obtained their detection label by antivirus software, using Virustotal. We separated them by families, using the results of the ESET-NOD32 anti-virus, and selected at most five samples from each family, resulting in a set of 1,470 samples. We analyzed these samples to obtain their behavior and used *Lumus* to identify which ones have evasive behavior. Since we did not have a ground truth with information about all these samples, we randomly selected 50 samples, all from different families, and manually inspected their results to identify possible false-negatives and false-positives. This manually analyzed samples became the ground-truth for our subsequent analysis.

Our technique detected 7 out of 50 samples (14%) in the subset as evasive. In the following, we provide a detailed analysis of each of the 7 cases. We

consider as false-negatives the samples that did evade analysis but *Lumus* did not identify as evasive, and we consider as false-positives those whose behavior exhibited in the bare metal was different from the emulated without trying to identify if the execution was inside an analysis system. Note that *Lumus* considers evasive those samples that execute some action only in the bare metal system, without executing some similar action in the emulated system, even if this divergence is not caused by a clear identification of the analysis system. For instance, if a sample tries to send SMS messages to contacts stored in the phone and it only shows this behavior in the bare metal because there is no contact registered in the emulated environment, *Lumus* considers it as evasive. We do this because, despite not being a clear sign of anti-analysis behavior, it is successful in preventing some action from being observed in the emulator and so could be employed as an anti-analysis technique. To test our intuition that the existing techniques to identify evasive Windows malware would not present as good results if applied to Android malware, we implemented detectors based on the techniques proposed by Disarm [15] and Barecloud [12]. Since the behavior of Android and Windows malware are different in various aspects, we used our behavior model when implementing these techniques. To make the comparison fairer, we used the threshold that would yield the best results to each of these techniques, instead of the threshold they found for Windows malware. Table 1 presents this comparisons' results, showing that our technique is far more suitable to the Android environment.

Table 1. *Lumus* vs. other Windows-based approaches to uncover evasive malware reimplemented for Android.

Approach	TP	TN	FP	FN	A
Lumus	100%	93.5%	6.5%	0.0%	96.7%
Disarm (t=0.12)	100%	78.3%	21.7%	0.0%	89.1%
Barecloud (t=0.36)	100%	67.4%	32.6%	0.0%	83.7%

6.1 Discussion

We discuss below the samples that *Lumus* identified as evasive, explaining why we consider them as a true-positive (TP) or false-positive (FP). For the TP, we discuss which extra behavior was observed due to the divergence and what difference between the environments was the cause of divergence.

Sample 1: It changes its behavior if */system/xbin/busybox*, */system/bin/busybox* or */bin/busybox* is present in the system. This deviation resulted in the malware writing to the file *shared_prefs/config.xml* and many files in the dir */SD card/LuckyPatcher/*. This may not have been intended as an anti-analysis technique, since most user systems do not have these files. However, because it does prevent some of the malware behavior from being observed in the emulated environment, we considered the behavior as TP;

Sample 2: It identifies if the phone number starts with "15555", whether the IMEI starts with "00000000" or if the IMSI starts with "31026". Upon detection,

it calls `System.exit(0)`. This is a clear case of evasive malware and a TP. The behavior resulting from the divergence is composed of starting a service and starting two alarms that send Intents;

Sample 3: It copies the icons of the apps installed in the system to the directory */data/data/com.pintudog/files/icons/*. Since the list of apps installed in the emulator and in the bare metal environments is not the same, the monitored actions ended up being different. However, at a higher level it is still the same behavior, so we consider this as FP;

Sample 4: It verifies if the IMEI contains the string "000000000000000". If so, the malware calls *System.exit(0)*. Similarly to Sample 2, this is a clear example of anti-analysis and a TP. The actions resulting from the divergence are the following: starting a service, creating a wake lock and connecting to the `dnsproxyd` device to make a DNS request;

Sample 5: The different actions in this sample's behavior are related to a file associated with the graphical interface, as the graphical libraries used in the bare metal and emulated systems are different. Since this is not actually related to the behavior of the malware, we considered this sample as a FP;

Sample 6: During its execution it verified which Wifi networks are available. In the emulated system it does not identify any Wifi network, so it takes a different execution path. The behavior that is executed only in bare metal, as a result of this difference, is writing a file in the SD card. Since this behavior is related to the malware execution and cannot be observed in the emulator unless some update is made to it, we considered this sample as a TP;

Sample 7: This sample randomly chose the domain name to access from a list of predefined names. This resulted in one domain used in bare metal not being used in the emulated analysis. Except for the domain difference, their behavior is the same, so we considered this as FP.

6.2 Employed emulator detection techniques

Out of the complete dataset of 1,470 samples, our technique identified 192 as evasive. A manual inspection of these evasive samples yielded several techniques that were used to discern emulators from bare metal devices. Most of these techniques use static or dynamic artifacts to detect differences from a bare metal.

Static artifacts usually constitute environmental values and configurations that differ between emulators and bare metal devices, and do not change between many complex states. Most of these values in fact remain constant at runtime and do not react much or not at all to environmental stimuli. Dynamic artifacts on the other hand usually result from the emulated interfaces that either show insufficient behavior compared to a bare metal device or are not emulated at all and can thus a variety of states can occur at runtime. This results in inconsistencies that shouldn't occur during normal bare metal execution [21].

We manually inspected the results of some of these samples to understand how they evade analysis. Bellow we describe the anti-analysis techniques we identified that are different from the ones explained before:

Static artifacts often can be queried through simple value lookup mechanisms. This allows for an easy to implement branching control flow, where

depending on the lookups' result the behavior can be benign or invasive. The previously introduced Sample 2 and Sample 4 also exhibited this behavior. Specifically, they leverage telephony related values to detect an emulated environment. Androids' `TelephonyManager`class allows to query specific information about the phones' identifiers, such as the IMEI and the IMSI, and the phones' line1 number. Vidas et al. pointed out that the AVD has the hard coded values of `155552155**` (wildcards in the line1 number are replaced by the system at boot time with the last two digits of the Android debug bridge (ADB) port) as its line1 number, a zeroed IMEI and an IMSI of `310260000000000`. Hence, checking for these values enables an application to detect the AVD as several samples in our dataset did. The downside of using telephony related values is necessary `Phone` permission group, specifically the `READ_PHONE_STATE` permission. Requesting this permission as an application that does not necessarily need it, e.g., a flashlight app, might raise some red flags and speed up discovery by malware analysts.

A permissionless technique is given through Androids' build properties. Applications can user the `android.os.Build` class or query the properties through one of the API methods `ProcessBuilder.start` and `Runtime.exec`. The tell-tale build properties of an emulator the malware probed for were the presence of the string "google_sdk" in the `PRODUCT` or `MODEL` properties, "generic" in the `BRAND` or `DEVICE` properties and "goldfish" ("Goldfish" is the previous canonical name for the QEMU-based AVD; the current canonical name is "Ranchu") in the `HARDWARE` property. If the device `FINGERPRINT` contained any of the strings "qemu", "sdk" or "generic" an emulator was detected as well. These build properties tests have in common that a value from a bare metal device is compared to its changed but present equivalent on an emulator. Another possibility is the examination of values that are only present on one of the two, e.g., the QEMU properties which are only present in Google's QEMU-based emulator. A query for "qemu" being present in the properties is enough to detect an analysis environment in this case, e.g., "qemu.sf.fake_camera", "ro.kernel.qemu" or "ro.kernel.android.qemud". Similar is to the build properties the existence of specific files can be queried. Exemplary the existence of QEMUD, Androids' QEMU Multiplexing Daemon can be assumed if the file */system/bin/qemud* exists or as with sample 1 the existence of */system/bin/busybox* helps detect specific environments.

Lastly, the presence of specific installed packages can be leveraged to make assumptions about the runtime environment. The absence of the Google Service Framework (`com.google.android.gsf`) constitutes an exception to the rule on a bare metal device. However, the AVD does not have its own version preinstalled making the query for this framework a valid choice. In addition the overall number of installed packages is revealing as well. An application can query installed packages through the `PackageManager`.Finding only packages from the domains `com.android` and `com.google` on a device reveals a bogus environment as a normal user would install packages from several other domains.

The **Dynamic artifacts** we encountered often leveraged the reactions to certain stimuli given by the application at runtime to ascertain whether the underlying device is an emulator. Although this sometimes can be handled

through a value lookup mechanism a more stable solution to test dynamic artifacts at runtime is given through error handling. To hide malicious logic through error handling an application could either try to access interfaces or runtime resources unavailable in an emulator or try to trigger an exception only on bare metal devices. Accessing resources unavailable inside an AVD will trigger an exception preventing code after the access request from being executed. Exceptions created only on bare metal devices allow malware authors to place exploitation logic inside the exception handling routine, leading to the same result as above. This blurs the line between regular and evasion logic and is harder to detect, especially through static analysis techniques.

Amongst the analyzed samples the connectivity interfaces `WifiManager` and `BluetoothManager` were frequented to detect a bogus environment dynamically. By default the emulator does not emulate the connectivity interfaces, meaning no other Wifi networks can be encountered around the device and the number of saved Wifi networks is zero. Samples used this to detect emulation if no saved Wifi networks were present and a scan request for Wifi networks came back empty or raised an exception. The `BluetoothManager` was used to retrieve the Handle for a bluetooth adapter used to interact with other bluetooth devices. Google's AVD does not emulate a bluetooth interface, hence the bluetooth adapter returned is a `null` Object and raises a `NullPointerException` if used. Applications can also abuse the emulated network connection itself. Specifically the emulators inability to forward `ICMP` packets can easily be used for detection.

The dynamic counterpart to the static query whether certain packages are installed is the interaction with those packages. Specifically the interaction with Googles' own PlayStore allows the detection of an emulator since by default only the bare minimum of applications are installed, excluding Google Play and as previously mentioned even the Google Services Framework. Interaction with unavailable services and applications, such as an intent trying to start the PlayStore, causes an exception which some of the analyzed samples used to display a benign behavior should an exception occur. Additionally, *any action requiring a fully set up Google Account will fail* as emulators are not set up with an account by default. Google Cloud Messaging is an exemplary service that requires a registered Google account to set up a Cloud Messaging id, also failing with an exception if no account has been set up. Root usage can prove helpful as well in distinguishing emulators from real devices. Specifically the "su" binary was used by some samples to try and gain superuser privileges. If no additional privileges were acquired by the application the *execution path* did not change and the additional malicious actions were not executed.

The last category we will present are sensor-related evasion techniques. The average Android device comes outfitted with several sensor types fitting the broad categories "Motion Sensors", "Environmental Sensors" and "Position Sensors". Similar to the connectivity interfaces the AVD does not emulate all sensors. We compared a LG Nexus 5X emulator to its bare metal counterpart and found 16 sensors emulated on the AVD while the actual model posses 25 sensors, excluding location sensors. A query about the sensor name reveals "Goldfish-*"

(the wildcard represents the sensor name, e.g., Goldfish-Accelerometer for an accelerometer) as opposed to real devices that offer the vendors canonic name, e.g., "BMI160 accelerometer". Another conspicuous detail is that the emulators' sensors all have one of two values as the sensors' vendor, either "The Android Open Source Project" or "AOSP" while bare metal devices feature an existing company, for example "Bosch". Trying to interact with sensors that are not available will lead to exceptions as well as using services that depend on the existence of certain sensors. We discovered malware in our dataset that uses the `LocationManager` for this purpose. A request to register a `LocationListener` results in an Exception on emulators leading to benign behavior.

Current Android Virtual Device Development. The different techniques we discussed were used by our malware samples to display a benign behavior in case an emulated environment was detected. These techniques rely on either differences in the execution environment or differences in the system behavior during execution on bare metal and emulated devices. To increase the resilience of the emulator against detection these differences can be addressed by closing the gap between bare metal devices and emulators. This would have implications on malware analysis, making it a lot harder for malware to detect an analysis environment, and in regards to our approach it would reduce the false positive rate working to our advantage.

Most static artifacts can be addressed through changes in the installation images for the AVD. For example, the hard coded telephony values can be randomized and the build properties can be modeled to simulate a real device either by changing them directly or hooking the methods used to query them [4]. Progress in bridging the gap between emulators and bare metal devices is also made by the Android Studio developers in order to improve the testing conditions for application authors. Versions of Android Studio newer than 2.3.2 started to feature AVD images including the GSF, GAPPS and Google Play, complementary to the system images without GSF. Dynamic artifacts can also be emulated but would require more effort in specific cases. A subset of device sensors is already emulated by the AVD and allows for event input, e.g., fake location providers would allow location spoofing and recoded series of events can simulate a moving device. Obvious red flags such as the vendor name can be changed as well and even missing sensors and connectivity interfaces can be emulated but would require a lot more engineering effort.

6.3 Limitations

Our detection approach relies on identifying differences between the execution of samples in bare metal and an emulator. Therefore, if execution does not reach the code with anti-analysis features, divergences cannot be observed. Insufficient code coverage is a common problem for dynamic analysis systems, as only executed behavior is usually analyzed. To exploit this, malware can delay the execution of anti-analysis code, or can only execute anti-analysis code after a series of complex GUI interactions that automatic interaction tools are unlikely to reach.

Some malware may be able to detect both environments as analysis systems, because despite the bare metal environment being more similar to a real device,

there are still differences that can be exploited, such as information about the user's behavior (e.g., browsing history and SMS history) and user data. Miramirkhani et al. analyzed these so-called wear-and-tear artifacts for Windows-based operating systems and their viability for analysis environment detection is considered very high because emulators and sandboxes are too "spotless", meaning virtually no user data and signs of usage can be detected [19].

When tracing back the origin of some behavior executed in bare metal, we may find some entry point whose source we cannot identify. In these cases we compare this untraced entry point with all instances of the same entry point in the emulated environment. In some cases, this may lead to wrong conclusions. Furthermore, differences in the systems may lead to the execution of different actions that are not related to evading analysis or to the execution of equivalent actions in both systems, but that are considered different in our behavior model. This is the general problem that introduces false positives, e.g., *Lumus* flagging Sample 3 and Sample 5 as evasive. Also, sources of non-determinism that we do not currently handle may lead to the execution of the same high-level behavior, but different actions according to our model. This is the problem that resulted in *Lumus* flagging Sample 7 as evasive. This malware randomly selects the domain name to access from a predefined list, so the domain accessed in bare metal and emulator were different, but the same code path was executed in both cases.

Lastly, the introduction of Google's SafetyNet Attestation API poses a threat to most dynamic analysis systems. The SafetyNet Attestation API allows an application to assess the security and compatibility of the Android environments in which the application is executed. The attestation is handled off-device by Google's servers after specific environmental data from the device is collected and the result is sent to any server specified by the malware's author. This allows malware to implement easy to use server-side runtime environment checks and even application tamper detection.

7 Related Work

Researchers have presented several techniques [18, 9, 29, 22] that Android malware may use to evade detection by making static analysis harder or by circumventing dynamic analysis. Matenaar and Schulz [18] present a method for an app to identify if it is executing inside QEMU, the basis of the Android emulator. Vidas and Christin [29] present anti-analysis techniques based on Android APIs, system properties, network information, QEMU characteristics, performance, hardware and software components. Petsas et al. [22] demonstrate anti-analysis techniques based on Android APIs, system properties, sensors and QEMU characteristics. Instead of manually identifying differences between real and emulated devices, Jing et al. [9] developed Morpheus, a framework that automatically generates heuristics that can identify, based on files, system properties and Android APIs.

Systems that automatically identify malware equipped with anti-analysis techniques have been developed for Windows [2, 15, 13, 12, 11]. Balzarotti et al. [2] propose recording the system calls executed by a sample in a reference environment

and replaying the monitored system calls in an emulator to identify if the observed behavior is different. Lindorfer et al. [15] analyze malware samples in different environments and identify differences in the observed actions. Barecloud [12] is a system that dynamically analyzes malware in four different environments and detects evasive malware by comparing the reports provided by these systems in a hierarchical approach. Kolbitsch et al. [13] detect and mitigate malicious programs that stall before executing their malicious behavior. Malgene [11] combines sequence alignment of system call traces, obtained from a bare metal and an emulated environment, with taint tracking to identify evasion signatures of evasive malware.

8 Conclusions

In this paper, we presented *Lumus*, a novel approach to identify evasive Android malware by comparing its execution on a bare metal analysis system and on an emulated analysis system. For each action executed only in bare metal, *Lumus* identified the basic cause why it was not successfully performed in the emulated environment, differentiating the cases in which there was evasion from the cases in which there was some analysis problem. Our experiments showed that our approach is much more effective for detecting Android malware with anti-analysis features compared to attempting to directly apply existing Windows-based approaches used to detect evasive malware. We analyzed 1,470 malware samples, from which our technique identified 192 as evasive. We presented detected evasion techniques after manually analyzing the samples.

References

1. Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., Rieck, K.: DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In: NDSS (2014)
2. Balzarotti, D., Cova, M., Karlberger, C., Kirda, E., Kruegel, C., Vigna, G.: Efficient detection of split personalities in malware. In: NDSS (2010)
3. Busch, M., Protsenko, M., Müller, T.: A cloud-based compilation and hardening platform for android apps. In: Proc. of the 12th Intl. Conference on Availability, Reliability and Security (ARES). SBA Research, Reggio Calabria, Italy (2017)
4. Dresel, L., Protsenko, M., Müller, T.: Artist: The android runtime instrumentation toolkit. In: Proceedings of the 11th International Conference on Availability, Reliability and Security (ARES). SBA Research, Salzburg, Austria (2016)
5. Elish, K.O., Yao, D., Ryder, B.G.: User-centric dependence analysis for identifying malicious mobile apps. In: MOST (2012)
6. Grace, M., Zhou, Y., Zhang, Q., Zou, S., Jiang, X.: Riskranker: scalable and accurate zero-day android malware detection. In: MOBISYS (2012)
7. Guan, L., Jia, S., Chen, B., Zhang, F., Luo, B., Lin, J., Liu, P., Xing, X., Xia, L.: Supporting transparent snapshot for bare-metal malware analysis on mobile devices. In: Proc. of the 33rd ACSAC. pp. 339–349. ACM (2017)
8. Haupert, V., Müller, T.: On app-based matrix code authentication in online banking. In: Furnell, S., Mori, P., Camp, O. (eds.) Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP). pp. 149–160. SciTePress, Funchal, Madeira, Portugal (2018)

9. Jing, Y., Zhao, Z., Ahn, G.J., Hu, H.: Morpheus: automatically generating heuristics to detect android emulators. In: ACSAC (2014)
10. Kalysch, A., Götzfried, J., Müller, T.: Vmattack: Deobfuscating virtualization-based packed binaries. In: Proceedings of the 12th International Conference on Availability, Reliability and Security. p. 2. ACM (2017)
11. Kirat, D., Vigna, G.: Malgene: Automatic extraction of malware analysis evasion signature. In: ACM CCS (2015)
12. Kirat, D., Vigna, G., Kruegel, C.: Barecloud: bare-metal analysis-based evasive malware detection. In: USENIX Security (2014)
13. Kolbitsch, C., Kirda, E., Kruegel, C.: The power of procrastination: detection and mitigation of execution-stalling malicious code. In: ACM CCS (2011)
14. Li, Y.: Droidbot. `http://honeynet.github.io/droidbot/` (2012)
15. Lindorfer, M., Kolbitsch, C., Milani Comparetti, P.: Detecting Environment-Sensitive Malware. In: RAID (2011)
16. Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., van der Veen, V., Platzer, C.: Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In: BADGERS (2014)
17. Maier, D., Müller, T., Protsenko, M.: Divide-and-conquer: Why android malware cannot be stopped. In: Proc. of the 9th Intl. Conference on Availability, Reliability and Security (ARES). SBA Research, Fribourg, Switzerland (2014)
18. Matenaar, F., Schulz, P.: Detecting android sandboxes. `http://www.dexlabs.org/blog/btdetect`
19. Miramirkhani, N., Appini, M.P., Nikiforakis, N., Polychronakis, M.: Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In: Security and Privacy (SP), 2017 IEEE Symposium on. pp. 1009–1024. IEEE (2017)
20. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: ACSAC. pp. 421–430. IEEE (2007)
21. Mutti, S., Fratantonio, Y., Bianchi, A., Invernizzi, L., Corbetta, J., Kirat, D., Kruegel, C., Vigna, G.: Baredroid: Large-scale analysis of android apps on real devices. In: ACSAC (2015)
22. Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S.: Rage against the virtual machine: hindering dynamic analysis of android malware. In: EUROSEC (2014)
23. Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., Vigna, G.: Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In: NDSS (2014)
24. Reina, A., Fattori, A., Cavallaro, L.: A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In: EUROSEC (2013)
25. Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Bringas, P.G., Alvarez, G.: Puma: Permission usage to detect malware in android. In: CISIS/ICEUTE/SOCO Special Sessions (2012)
26. Spreitzenbarth, M.: The Evil Inside a Droid - Android Malware: Past, Present and Future. In: Baltic Conf. on Network Security & Forensics (2012)
27. Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., Hoffmann, J.: Mobile-sandbox: having a deeper look into android applications. In: ACM SAC (2013)
28. Su, X., Chuah, M., Tan, G.: Smartphone dual defense protection framework: Detecting malicious applications in android markets. In: Intl. Conf. Mobile Ad-hoc and Sensor Networks (2012)
29. Vidas, T., Christin, N.: Evading android runtime analysis via sandbox detection. In: AsiaCCS (2014)

30. Wu, D.J., Mao, C.H., Wei, T.E., Lee, H.M., Wu, K.P.: Droidmat: Android malware detection through manifest and api calls tracing. In: Asia JCIS (2012)
31. Zheng, M., Sun, M., Lui, J.C.: Droidanalytics: A signature based analytic system to collect, extract, analyze and associate android malware. In: TrustCom (2013)
32. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: IEEE Simp. on Security & Privacy (2012)
33. 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)

Appendix - Action types and its operations and arguments

Behavior is represented by actions, where each action is a tuple $a = (action\_type, operation, argument)$, and *action_type* is one of {*Network*, *File*, *Intent*, *Exec*, *Phone*, *Dex*, *Billing*, *Multimedia*}. Action types, as well as its operations and arguments are described below.

Network. For network related actions, operation is one of {*INET*, *UNIX*, *NETLINK*, *BLUETOOTH*}. *INET* operations represent TCP and UDP connections and *argument* is the destination. Since multiple resolutions of the same DNS name may result in different IP addresses, we consider two actions the same if they use the same IP address or the same DNS name as destination. *UNIX* operations represent connections to UNIX sockets and the argument is the filesystem path used. *BLUETOOTH* operations represent the use of the Bluetooth device and the argument is the operation performed with this device. Lastly, *NETLINK* operations represent connections using NETLINK sockets and the argument used is the protocol parameter passed to the socket.

File. The monitored operations on files are *WRITE* and *DELETE*, and the argument of both is the file path.

Intent. Intent-related operations include *ACTIVITY*, *SERVICE*, *BROADCAST* and *ALARM*. The argument for all these operations is the "action" argument of the Intent or the destination class of the Intent. *ALARM* operations refer to the use of *AlarmManager* to send Intents.

Exec. This action type represents the launch of the `execve` system call, which is used by the API methods *ProcessBuilder.start* and *Runtime.exec*. The argument used is the name of the executable file being invoked.

Phone. This action represents the use of phone capabilities. We currently consider only one operation of this type (sending SMS messages); the argument is the destination number of the message.

Dex. This action type represents the use of dynamic code loading and its argument is the path of the file being loaded.

Billing. This action represents the use of the billing functionality; the argument is the type of action performed.

Multimedia. The operations included in this action type are *CAMERA*, *AUDIO* and *WAKELOCK*. The argument in these cases is the type of action being performed, which includes taking pictures, recording videos, recording audio, or acquiring wake locks.