Curious-Monkey: Evolved Monkey for triggering malicious payloads in Android malware

Hayyan Hasan,
Behrouz Tork Ladani
Bahman Zamani

Abstract

Dynamic analysis is a prominent approach in analyzing the behavior of Android apps. To perform dynamic analysis, we need an event generator to provide proper environment for executing the app in an emulator. Monkey is the most popular event generator for Android apps in general, and is used in dynamic analysis of Android malware as well. Monkey provides high code coverage and yet high speed in generating events. However, in the case of malware analysis, Monkey suffers from several limitations. It only considers UI events but no system events, and because of random behavior in generating UI events, it may lose dropping the connectivity of the test environment during the analysis process. Moreover, it provides no defense against malware evasion techniques. In this paper, we try to enhance Monkey by reducing its limitations while preserving its advantages. The proposed approach has been implemented as an extended version of Monkey, named Curious-Monkey. Curious-Monkey provides facilities for handling system events, handling evasion techniques, and keeping the test environment's connectivity up during the analysis process. We conducted many experiments to evaluate the effectiveness of the proposed tool regarding two important criteria in dynamic malware analysis: the ability to trigger malicious payloads and the code coverage. In the evaluation process, we used the Evadroid benchmark and the AMD malware data-set. Moreover, we compared Curious-Monkey with Monkey and Ares tools. The results show that the Curious-Monkey provides better results in case of triggering malicious payloads, as well as better code coverage.

Key words: Curious-Monkey, Monkey, Android, Event generator, Android malware, Dynamic analysis

1. Introduction

A considerable number of Android apps that are daily uploaded to different stores to be downloaded and used by users are classified as malware or malicious codes. These apps may be in various domains including gaming, educational, social, and business. Malware apps can harm users in many different ways, such as stealing user sensitive information, sending them to remote servers, or controlling the user's mobile phone to do malicious behavior. To detect Android malware apps, a special analysis process is adopted. In general, there are three types of the analysis process for Android apps: static analysis, that examines the program structure [1], [2], [3] dynamic analysis that studies the behavior of the program in run time [4], [5], [6], [7], and hybrid analysis that includes both static and dynamic analysis [8], [9].

To study the behavior of the malware apps in dynamic analysis, we need to use a special test environment such as Google's Bouncer [10] to run the app and analyze its behavior. However, Android apps are event-driven, i.e., we need to feed the app with events during the analysis process to execute each part of the code. Feeding events represents the first step that guides the dynamic analysis process of Android apps. Bouncer uses Monkey [11] as a random event generation tool to feed events to the app under test. In addition to Bouncer, many other dynamic analysis frameworks use Monkey to feed events to their apps under test [12].

Several tools are proposed to handle the event generation process for Android malware dynamic analysis [13], [14]. These tools can be considered as an enhancement to the event generation mechanism that is used by dynamic analysis frameworks. A category of these tools mainly tries to make UI events more realistic and related [13], [15]. Al-

Email addresses: h.hasan@eng.ui.ac.ir (Hayyan Hasan), ladani@eng.ui.ac.ir (Behrouz Tork Ladani), zamani@eng.ui.ac.ir (Bahman Zamani).

though generating realistic UI events are essential in case of malware analysis, some other inputs such as system events and fed values to handle evasion techniques are also required. The second category of the malware specific event generation tools rely on finding evidences of the malicious payloads in the code to direct the execution toward these payloads [14], [16], [17]. However, using such tools is time-consuming, i.e., they need a lot of time to inspect the code and direct the execution toward the payloads. Hence, they are not as popular as the former category.

Despite the existence of specific malware event generators, Monkey is yet popular for dynamic malware analysis purpose. The main reason is that it is easy to use and configure, it obtains high coverage, and it provides events very fast, so it makes the analysis time very low, which is very important if the number of malware apps under test are high. However, Monkey has some limitations when it is used for dynamic malware analysis. For example, it generates only UI events but no system events. UI events include interacting with the activity components of the app under test, such as click or touch. In contrast, system events include all other events that can be sent to the Android device, such as changing the battery status or phone status. Moreover, Android malware use evasion techniques, such as anti-emulation techniques to detect the environment in which they are running and to hide the payload if they detect the existence of emulator. Unfortunately, Monkey is not able to generate proper events for handling such techniques. Finally, in Monkey, the generated events are random-based, and this may cause activating unintended or risky activities such as turning on the airplane mode or turning off the internet when analyzing the malware in test environment. This may affect the analysis process because many malware samples communicate with their command and control (C&C) servers or download the payload during the execution.

The proposed approach has been implemented as an extended version of Monkey, named Curious-Monkey to overcome the aforementioned shortcomings. We have made the source codes of Curious-Monkey as an open source software available in GitHub [18]. Our idea is to use Monkey to generate UI events and interact with the app under test while generating different types of system events and handling evasion techniques. Finally, we added a facility for checking the connectivity of the app every time we send events to the test environment.

The generated system events are in two main categories. The first category includes events that are used for setting up the test environments such as adding contacts to the device, adding call logs, and adding history and bookmarks to the browser. The Second category includes sending events during the analysis process such as changing the phone status and rebooting the device. Furthermore, handling evasion techniques is done by using the Xposed module. This module dynamically hooks some invoked APIs as in APIs that are used to detect the test environment and set their returned results with values similar to the real device values. The Xposed module handles most APIs used by malware to evade dynamic analysis such as Android.os.Build fields that provide information about the used device build, Android.telephony.TelephonyManager methods that provide information about the telephony services in the used environments, and sleep method that delays the execution of the payload.

The main advantage of using Curious-Monkey is that it equips Monkey with techniques that makes it more suitable for dynamic malware analysis without affecting its advantages. In other words, adding system events and handling evasion techniques provide a significant contribution because system events are frequently used by malware to trigger malicious payloads, and the handled evasion techniques are mostly used by Android malware to hide their malicious payloads from dynamic analysis.

We conducted several experiments to show the effectiveness of Curious-Monkey. We used Evadroid benchmark [19] and 100 real-world malware samples from 10 different families from the AMD malware data-set [20], [21]. The comparison process is achieved against Monkey [11] and Ares [19]. Note that Ares is the only available tool for us that includes prominent features that are comparable with Curious-Monkey. We used the ability to trigger malicious payload and code coverage as criteria to evaluate the effectiveness of the proposed approach. The ability to trigger malicious payloads can be measured by logging sensitive APIs when they are called by the malware during the execution. We used Droidmon [22] to capture the sensitive APIs, and ACVTool [23] to capture the code coverage of the executed samples. The results showed that the proposed approach provides good results in the case of triggering malicious payloads as well as the obtained code coverage.

The main contributions of this paper are as follows:

- A tool that is used to handle the input generation process for dynamic malware analysis. This tool generates both UI events and system events that are critical in triggering the malicious payloads in malware apps.
- A collection of the most used system events that are used to trigger the malicious payloads in the code based on the empirical study that we conducted on the Contagio mobile dataset [24].
- An Xposed module that can dynamically handle the most used evasion techniques by Android malware. Selection of the evasion techniques to be handled is based on the empirical study that we conducted on the Contagio mobile dataset [24].

The paper is organized as follows. Section 2 illustrates the related work of the proposed approach. Section 3 provides a motivation example to better illustrate the research problem. Section 4 presents the proposed approach. Section 5 is dedicated to empirical evaluation and comparison of Curious-Monkey against Ares and Monkey tools, and finally, Section 6 concludes the paper.

2. RELATED WORK

There are many studies on finding the ways for triggering malicious payloads in the malware codes during their dynamic analysis process. Some of these studies focus on making the generated UI events more realistic as in DroidBot [13], while other studies try to find evidences of the malicious payloads in the code to direct the execution toward these payloads, as in Ares [19], and DirectDroid [14].

DroidBot [13] is a model-based event generator that depends on dynamically building the state transition model of the app GUI, to provide appropriate events for transition between different activities of the sample under test. Alzaylaee et al [15], proposed a hybrid approach by using random-based and model-based techniques for generating UI events to increase both the code coverage and the ability to trigger the malicious payload in the code. Curiousdroid [25] is another model-based approach that provides more complex UI events than Monkey and DroidBot such as filling up the text filed with appropriate input. These approaches are introduced as an enhancement to the UI event generation mechanisms. MEGDroid [26], is another tool that makes use of MDE approach to provide both UI and System events. However, despite the fact that these approaches provide realistic events, but they take relatively long time to build the model and to generate these appropriate events.

Other researchers propose target executing the malware payloads in the code. As an example, DirectDroid [14] is an approach that makes use of fuzzing and on-demand force execution techniques to trigger the malicious payloads in the code. This approach consists of two main components, static analyzer and dynamic executer. The static analyzer performs control flow analysis and symbolic data flow analysis to identify each trigger in the payload path. After static analysis, the dynamic execution is achieved. It repeatedly executes the app using fuzzing and on-demand force execution to reach the location of the payload. However, this approach suffers from many limitations. For example, if the malware code is obfuscated, the static analyzer may not be able to detect each target location, hence it is impossible to identify each call path in the code.

FuzzDroid [17] is a framework that refines the execution environment to reach the malicious payload in the code. It uses a hybrid approach to inspect the code and track the execution path to the malicious payload. This tool repeatedly executes the malware code and adapts the environment in each execution to exceed the guards defined to prevent executing the malicious payloads. However, this tool currently considers the guards located in one component and may fail if the guards are distributed among different code components.

Ares [19] is an approach that forces the execution path to reach the malicious payloads. It uses the static information flow analysis approach that is based on the source-sink concept to detect the guards of the malicious payloads. Moreover, this approach uses binary instrumentation to force executing the guards along the paths to the malicious payload. The source, which they call it fingerprinting source (FS), represents any instruction that gives information about the environment and the sink represents any logical condition. Ares detects every logical condition and checks its source. If the source is an FS, it forces the execution of both sides of this condition. However, using pure static analysis to detect the information flow paths imposes some limitations, if the code is obfuscated. Moreover, force executing the path may cause the app to crash in most cases.

GroddDroid [16] is an event generator that considers the anti-emulation capabilities of the malware. This tool first runs the code on a real device to get a reference execution, and then it inspects the code for malicious payloads. The information from the previous steps will be used to force executing the evasion techniques to reach the malicious parts of the code. However, force executing the paths may cause the app to crash in most cases.

IntellDroid [27] is a tool that makes use of static analysis to detect the call paths in Android apps and the constraints in each path. After that, the detected constraints are solved, and appropriate events are generated to execute the detected paths. However, using static analysis is not sufficient because malware may use anti-static analysis techniques, such as obfuscation, to prevent extracting any information.

The main limitations of the tools that are based on the above approaches is that they cannot cover all the parts of the code and there is no guaranty to execute payloads that are distributed in different places of the code. Moreover, they are very complex and consume plenty of time when generating different inputs for each sample; This causes problems when dealing with large number of samples.Table 1 summarizes the analysis of the main malware specific input generation tools.

Table 1
A comparison among the main existing Android input tools

Approach	Inputs type		Handling evasion techniques			Required time
	UI events	System events	Values	Force execution	Feed values	
Monkey [11]	*					Small
DroidBot [13]	*					Medium
Alzaylaee et al. [15]	*					Medium
Curiousdroid [25]	*					Medium
MEGDroid [26]	*	*				Medium
DirectDroid [14]	*	*	*	*	*	Large
FuzzDroid [17]	*	*	*		*	Large
Ares [19]	*	*		*		Large
GroddDroid [16]	*	*		*		Large
IntellDroid [27]	*	*	*		*	Large
Curious-Monkey	*	*	*		*	Small

In summary, the mentioned malware specific input generation tools have the following shortcomings:

– They do not consider system events or do not handle evasion techniques,

– They try to find the execution path to the payload and try to execute this path, which implies some limitations, such as consuming long time to execute the payload (because of the techniques that are used to find the payload in addition to multi execution process to bypass the evasion techniques), and considering only one part of the code without other parts, i.e., there is no guaranty to execute the payload distributed in different parts of the code.

Curious-Monkey extends our previous work [28], and tries to handle the aforementioned shortcomings. In fact, it provides a simple and fast input generation mechanism that is suitable to handle a large number of Android malware samples in a short time. The generated inputs by Curious-Monkey include both UI and system events in addition to the values that satisfy the evasion techniques along the paths to the payload. Since the proposed approach is random based and yet uses feeding values to handle evasion techniques (i.e., there is no need to multiple executing the sample under test), the time consumed by the proposed approach is very short. These features make Curious-Monkey very suitable to handle a large number of samples in a short time.

3. MOTIVATION EXAMPLE

Most of the modern Android malware use techniques to hinder executing the payload during dynamic analysis. They normally use complex checks along the path to the payload that makes it very hard for the malware analysis tools to bypass these checks to reach the payload. We reverse-engineered many samples in the Contagio mobile dataset [24], in addition to the samples of Evadroid benchmark. Listings 1 and 2 represent an example code that is used in many of these analyzed samples concerning the used API methods to hinder executing the malware payload. In this example, we used Android.os.Build fields as evasion techniques. Some other methods such as getDeviceId and getMacAddress of Android.telephony.TelephonyManager and the get method in Android.os.SystemPropertie can also be used. Listing 1 shows the beginning of the execution path that leads to the payload. As can be seen from Listing 1 , the sample code registers a boot receiver to listen to the Android.intent.action.BOOT_COMPLETED intent. When this intent is received, a dynamic code load will be activated, i.e., the sample will load another APK file with the name app.apk (lines 5 to 7), then a class with the name com.example.test.payload will be loaded (line 8), and the method checkandtrigger from this class will be invoked (line 9).

Listing 1. First sample code to hinder executing the malware payload: waiting for the boot intent to start the execution path to the payload

```
1 public class boot extends BroadcastReceiver {
2 private static final String mACTION =
  "android.intent.action.BOOT_COMPLETED";
3 public void onReceive(Context context,Intent intent) {
4 if (intent.getAction().equals(mACTION)) {
5 File file1 = new File(getDir("dex", 0), "app.apk");
6 File file2 = getDir("outdex", 0);
7 DexClassLoader dLoader =
  new DexClassLoader(file1.getAbsolutePath(),
  file2.getAbsolutePath(), null, getClassLoader());
8 Class<?> c = dLoader.loadClass
  ("com.example.test.payload");
9 c.getDeclaredMethod("checkandtrigger", new
  Class[]{Context.class}).invoke(i,new Object[]{context});
10            } }}
```

Listing 2 represents the com.example.test.payload class, and the checkandtrigger method of this class. As can be seen from Listing 2, the sample uses many evasion checks across the path to the payload. The used checks are Build.MANUFACTURER, and Build.SERIAL (line 4), Build.BOARD (line 5), and Build.HARDWARE (line 10). The used checks determine if the returned values are for emulator or not, for example, if Build.MANUFACTURER equals to EnvironmentCompat.MEDIA_UNKNOWN (line 4), it implies that the sample is running in the emulator, so the check will not be satisfied, and the payload will not be triggered.

Listing 2. Second sample code to hinder executing the malware payload: using Build fields as evasion techniques

```
1   public class payload {
2   public void checkandtrigger(Context context) {
3   boolean check;
4   boolean check2 =
     Boolean.valueOf(Build.MANUFACTURER.equals
    (EnvironmentCompat.MEDIA_UNKNOWN)). booleanValue()
    ||Build.SERIAL.equals(EnvironmentCompat.MEDIA_UNKNOWN);
    5 if (Boolean.valueOf(check2).booleanValue() ||
    Build.BOARD.equals(EnvironmentCompat.MEDIA_UNKNOWN)) {
6   check2 = true;
7  } else {
8  check2 = false;
9         }
10 if (Boolean.valueOf(check2).booleanValue() ||
    Build.HARDWARE.equals("goldfish")) {
11 check = true;
12        }
13 if (!Boolean.valueOf(check).booleanValue()) {
14 Toast.makeText(context, "Payload triggered!", 1).show();
15        } } }
```

When using Monkey or Ares to run the codes in Listings 1 and 2 and other similar samples, none of them can trigger the payload. For the case of Monkey, it generates UI events only, and since the execution path to the payload starts with a system event (reboot the device), this tool will not be able to trigger the payload. In the case of Ares, since the evasion techniques is in the dynamic load code and Ares uses static analysis to detect and defeat the evasion techniques, it is not able to bypass the evasion techniques to reach the payload.

As can be seen from this motivation example, the malicious payload in the code is triggered after sending the boot intent (i.e., reboot the device), and handling the evasion

techniques along the path to the payload. In other words, regardless of the generated UI events, the malicious payload will not be triggered unless the boot intent is received and the checks are satisfied. Consequently, this example shows the important role of system events and handling the evasion techniques during the execution to trigger malicious payloads from the code.

4. THE PROPOSED APPROACH

The proposed approach makes use of Monkey as its UI event generator. Moreover, it is equipped with system events, handling evasion techniques dynamically, and connectivity checks to make Monkey more suitable for dynamic malware analysis. In the following, we will illustrate the steps of the proposed approach, the types of the used connectivity checks, the types of the provided events, and the proposed Xposed module that handles the evasion techniques.

4.1. *The steps of the proposed approach*

We start with installing the malware APK into the test environment, then both system and UI events are sent to execute the app under test. During the execution, the evasion techniques are handled dynamically by the Xposed module. Note that the proposed approach works in the following order:

- Set up the test environment: it includes sending system events that are responsible for setting up the test environments, such as set call log, add history and bookmarks to the browser, and insert different types of accounts (see subsection IV.C).
- Check the connectivity of the test environment: this is achieved to ensure that the malware apps can do many malicious behaviors such as connecting to their C&C servers (either by internet or SMSs). However, the check includes checking both Airplane mode and internet connection. (see subsection IV.B).
- Monkey is set to generate 10000 UI events. Note that we check the connectivity of the test environment every time Monkey finishes producing events.
- The generated UI events will start the execution paths. If some evasion techniques found along the paths, the Xposed module handles them whenever they are invoked and set their returned values to values similar to real device values.
- Send run time system events: such events are sent after launching the apps by Monkey (see subsection IV.C). Note that we check the connectivity of the test environment every time an event has been sent.
- The Xposed module handles the possible evasion techniques which exist along the execution paths that are started by the run time system events.
- Monkey is set to generate next 10000 UI events, and the evasion techniques along the paths will be handled by the Xposed module.
- The logs of this operation are obtained and saved to analyze the results of running the malware.

It should be noted that this order is achieved according to our experimental observation on malware samples from AMD malware dataset. i.e., we tried different orders for sending events to run the apps under test. According to our experiments, the aforementioned order provided the best results regarding the number of logged sensitive APIs and code coverage.

4.2. *Types of the used connectivity checks*

In general, Monkey causes turning on the airplane mode or turning off the internet connection because it generates random UI events that occasionally cause dropping down the notification list and click on either airplane or wireless button. Connectivity plays a vital role in dynamic malware analysis because most malware samples need to connect to their C&C servers using either internet or SMSs. This connection is required to either send instructions to the malware or send the victim's information to the remote server. In other scenarios, malware may download the payloads during the execution time (dynamic loading code). Hence, it is essential to keep the connectivity in the test environment. In the proposed approach, we reuse the checks illustrated in [15] to check the connectivity of both airplane mode and internet connection. So, we wrote two codes, the first one checks if the airplane mode is enabled, and disable it if so. While the second one, checks if the wireless or the data are off, and turn them on if so.

4.3. *The generated events*

Curious-Monkey can generate two types of events: UI and system events. We directly use Monkey to generate UI events because of its high code coverage. However, as mentioned before, for the case of malware analysis, Monkey suffers from a number of shortcomings and needs to be extended with system events. In the following, we will demonstrate both types of the generated events by the proposed approach.

4.3.1. *Generating UI events*

UI events include any event that can interact with the activity components of the app under test. We use Monkey to generate UI events. Monkey is a tool that is developed as a part of the Android toolkit to execute the apps under test. This tool generates different types of UI events, such as clicks, drags, and touches, randomly. We configure Monkey to generate 10000 UI events each time, as we set the proposed approach to run Monkey two times, as illustrated subsection IV.A.

4.3.2. *Generating system events*

System events include any event that does not interact with the activity components of the app under test. In general, these events are produced to respond to the requested permissions and registered receivers in the apps. We added the events mentioned in [29] and the events that we got from extensively analyzing malware in the Contagio mobile dataset [24]. We categorize these events into two types. The first type is used to set up the test environment and is sent after installing the malware APK file, as illustrated in subsection IV.A. This type includes the following events:

- Adding contacts to the test environment
- Adding SMSs.
- Adding browser history and bookmarks.
- Adding different types of files to the test environment.
- Opening the camera and take photos.
- Adding different accounts to the test environment.
- Making different download operations.
- Adding calls log to the test environment.

The main goal of these events is to make the test environment similar to a real device that is used by the user. This is because some malware check to see if the device is used in a tracking and analysis mode or is work in a regular device used by a regular user.

The second type of system events is used to simulate the real device in most cases. It includes sending the events after running Monkey for the first time. This type of events mostly includes sending intents to match the intent filter of the registered receivers or to respond to the requested permissions. However, this type is categorized as the following:

- Changing the battery status such as battery low, full, and charging,
- Sending intents to change the media status such as media injected and media mounted,
- Sending intents to change the package status such as package added, package removed, and package changed,
- Sending boot intent to reboot the device,
- Sending intents to change the power status such as power connected, and power disconnected,
- Sending intents to change the time status parameters such as time zone, and time set,
- Sending intents to change the UMS (USB Mass Storage) status such as UMS connected and UMS disconnected,
- Many other events, such as receiving messages, changing the network status, and replacing the data status.

4.4. *Xposed module*

To handle the evasion techniques used by the malware, we use the Xposed framework [30]. This framework roots the test environment and provides the ability to add different types of modules. Xposed modules dynamically hook predefined APIs and provide many operations to work with the hooked APIs, such as setting their returned results to specific values, getting their arguments (in case of methods), and getting their returned results. Therefore, we developed an Xposed module that hooks some predefined sensitive APIs used for evasion techniques and sets their returned values to values similar to the real device values.

Back to the motivation example, the existing checks along the execution path to the payloads use Android.os.Build fields to detect the existence of an emulator. The proposed Xposed module hooks these fields whenever they are invoked and set their values as the following: 1) the value "google" for Build.MANUFACTURER in line 4, 2) the value "C4F12FDD949F22F" for Build.SERIAL in line 4, 3) the value "samsung" for both Build.BOARD in line 5, and Build.HARDWARE in line 10. This way, the checks along the path to the payload will be handled and as a result the payload will be triggered.

To facilitate adding or removing the predefined APIs to be hooked, we set the Xposed module to read these APIs from a JSON file. However, the hooked APIs in the experiments can be categorized as follows:

- **Telephony:** this category contains Android.telephony. TelephonyManager methods, such as getDeviceId, getSubscriberId, and getSimSerialNumber. Android malware widely use these methods to detect the environment in which they run. For example, Android malware can use the method getDeviceId to get the ID of the test environment and check if the returned value is equal to "000000000000000" or not. Positive answer means that the malware is running on an emulator.
- **Build:** this category contains the fields of the Android.os.Build class. These fields are also widely used by Android malware to get information about the device build, such as MODEL, HARDWARE, and MANUFACTURER.
- **Location:** some malware use GPS to get the device location and run their malicious behavior accordingly.
- **WIFI:** some malware use the MAC address or the IP address of the WIFI to detect the test environment.
- **System properties:** includes using the fields of Android.os.SystemProperties class, such as ro.product.name and ro.serialno to detect the test environment.
- **File:** this category includes reading system files to determine the test environment, such as /proc/tty/drivers that includes information about the device drivers, and /proc/cpuinfo that includes information about the device CPU.
- **Time:** this category differs from other categories as it delays the payload execution until a specific period or determines triggering the payload in a specific time. Examples of this category are the sleep method that delays the execution for a specific period, or getHours method from java.util.Date that determines the hour of the day in which the payload can be triggered.

5. EVALUATION AND COMPARISON

To evaluate Curious-Monkey, we implemented it using Java programming language to generate and send both types of events to the test environment. Also, we used Java and XML to implement the Xposed module as an Android app. We used Genymotion emulator [30] with Google Nexus 5 image on it as the test environment in our experiments. Android Lollipop with API level 21 was used as the Android operating system. Finally, we used a fresh clone of the emulator before running each sample to ensure that the analyzed samples are not affected by each other.

The evaluation is performed to find two important metrics: the coverage metric, which represents the percentage of the lines of code that are executed by the event generator, and the number of sensitive API calls, which represents the event generator's ability to trigger the malicious payload in the code. We used ACVTool [23] to measure the code coverage metric, and Droidmon [22] (which is an open-source Dalvik monitoring framework based on Xposed framework [30]) to hook the calls of sensitive APIs (that reflect the malicious behavior of the sample under test) and to report all the monitoring information in the logs obtained by the Logcat tool [31].

In the evaluation process, we use Evadroid benchmark and 100 samples that are randomly selected from 10 different families from AMD malware dataset (as illustrated in Table 2). Note that, in the Evadroid benchmark, when the payload is triggered, as a result of the generated events and handling the evasion techniques, a Toast message will appear to show that the event generator successfully triggers the payload. While in the AMD dataset, we chose the malicious behaviors introduced in GroddDroid [16] and capture the sensitive APIs that reflect these malicious behaviors. As a result, when these sensitive APIs are captured, it indicates that the payload or part of it, is triggered.

In this section, we conduct experimental evaluations to address the following research questions:
RQ1) How effective is Curious-Monkey?
RQ2) How does Curious-Monkey perform comparing with other tools?

Table 2
The used malware samples in the evaluation process and the number of samples in each family (in parenthesis)

1) Utchi (12)	6) Kuguo (12)
2) Koler (9)	7) Triada (10)
3) FakeInst (8)	8) FakeDoc (8)
4) Fusob (13)	9) SimpleLocker (8)
5) FakeTimer (13)	10) Mmarketpay (7)

Results
In this section, we answered the aforementioned research questions based on the experimental results.

RQ1) The effectiveness of Curious-Monkey
To show the effectiveness of Curious-Monkey, we used two metrics: the code coverage and the ability to trigger the malicious payload or part of it. In this experience, some samples from both Evadroid benchmark and AMD dataset are used.

1- Triggering the malicious payload
In this section, we evaluate the ability of Curious-Monkey in triggering the malicious payload (or part of it). To do that, we use both Evadroid samples and the samples in Table 2. In the Evadroid benchmark case, we ran the samples using Curious-Monkey and wait for the Toast message that indicates if the payload was triggered or not. Table 3 represents the results of using Curious-Monkey in triggering the payload in the Evadroid benchmark samples.

Table 3
Triggering the payloads in Evadroid benchmark

Sample name	Payload triggered?
AccelH	Yes
AdbEnable	Yes
AdbPortDetector	Yes
AtNight	Yes
BatteryCharging	Yes
BatteryFull	Yes
BatteryStatus	Yes
ConstantCalls1	Yes
ConstantCalls2	Yes
Constants1	Yes
Constants2	No
ConstantsDLC	Yes
DivById	Yes
GetIpAddress	Yes
InstalledApps	Yes
LongAction	Yes
PostDelayed	Yes
ProcNetTcp	Yes
QemuFingerprinting	No
SignatureVerification	Yes
Sleep	Yes
Uptime	Yes

As can be seen from Table 3, Curious-Monkey could trigger the payload in 20 samples from Evadroid benchmark, while it failed to trigger the payload in 2 samples: **Constants2** and **QemuFingerprinting**. After investigating these two samples, we found that, in the case of the Constant2 sample, the used checks before reaching the payload

are, Build.MODEL, Build.PRODUCT, Build.BRAND, and Build.FINGERPRINT. The proposed approach could set the returned values for the first three checks to the values similar to real device values, but it failed to set the returned values of Build.FINGERPRINT check because of the limitation in the Xposed framework. While in the case of **QemuFingerprinting**, we could not install this sample into Genymotion emulator.

All samples of a malware family in the selected families from the AMD dataset have common malicious behaviors that are according to the classification represented in GroddDroid [16]. Therefore, we adopted GroddDroid classification and selected the sensitive APIs that reflect each category in this classification. Note that, we represented each malicious category with a number. To get the number of each category we sum up the frequency of invoking sensitive APIs that reflect this category. For example, the SMS category includes two sensitive API methods, sendTextMessage, and sendMultipartTextMessage from Android.telephony.SmsManager class, to get the number of this category, we sum up the frequency of invoking these two API methods from the 100 samples illustrated in Table 2. The result was 2 for sendTextMessage, and 0 for sendMultipartTextMessage, so the number of SMS category is 2. Table 4 represents the results of using Curious-Monkey to trigger the most common malicious categories from the samples introduced in Table 2.

Table 4
Most common malicious categories triggered by Curious-Monkey

Malicious Category	Curious-Monkey
SMS	2
Telephony	78
Binary	0
Dynamic	34
Reflection	20
Crypto	22
Network	22

As can be seen from Table 4, the most triggered category is the Telephony category. This category includes many methods that are commonly used by Android malware to get information about the test environment in which they run. For example, the method getDeviceId is used to get the ID of the test environment, and the method getSubscriberId is used to get the subscribed ID of the test environment.

2- Code coverage

To evaluate the ability of Curious-Monkey to execute as much as possible from malware code lines, we used ACVTool [23] to get the code coverage of the samples illustrated in Table 2 when they are executed by Curious-Monkey. Figure 1 represents the obtained coverage from the aforementioned samples, when they are executed by Curious-Monkey.

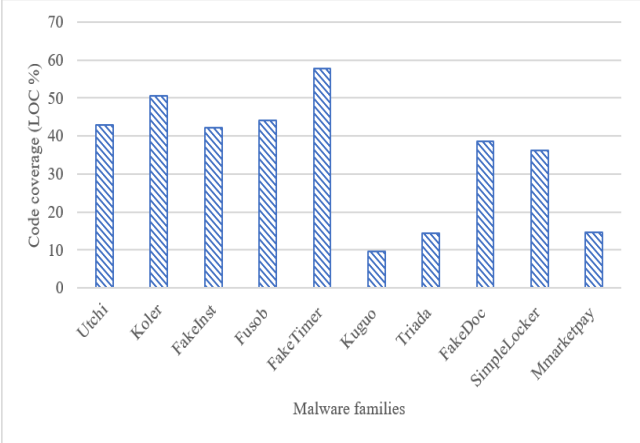

Fig. 1. Code coverage obtained by Curious-Monkey

As can be seen from Figure 1, Curious-Monkey provides code coverage varied between 9.7% and 57.6%, which can be considered as very good in the case of malware. The family with the lowest code coverage was **Kuguo** because it has complex functionality, many complex activities, and many guards that could not be handled by Curious-Monkey such as waiting for a message from the attacker to trigger the payload. On the other hand, the family with the highest code coverage was **FakeTimer**, because the samples of this family have very simple functionality and a small number of activities with simple views in each activity.

RQ2) Comparing with other tools

In this section, we compare Curious-Monkey with Monkey and Ares. In the case of Ares, we use the Evadroid benchmark samples. Moreover, the used metric in this evaluation is the ability to trigger malicious payload. In case of Monkey, we use 100 samples illustrated in Table 2 from the AMD dataset. The used metrics in this evaluation are code coverage and the invoked sensitive APIs that reflect the malicious behavior of the used samples.

a- Comparing with Ares

Table 5 represents a comparison between Curious-Monkey and Ares. As can be seen from Table 5 , Curious-Monkey could detect more payloads than Ares. This is because Curious-Monkey uses dynamic analysis to detect and set the returned values of the invoked method to resist against evasion techniques. In contrast, Ares uses static analysis to detect the evasion techniques which makes it useless in many cases. For example, in case of **Constants-DLC** malware sample (in the Evadroid benchmark) Ares fails to detect the payload because the evasion techniques are located in dynamic load code. Also, similarly in cases of **DivById**, and LongAction samples, Ares again could not detect the payloads because it uses static information flow analysis to detect the evasion techniques and hence bypass

them using force execution. In fact, in the recent two samples, Ares fails to detect the evasion techniques because these samples did not fulfill the information flow analysis requirements used by Ares. Therefore, no force execution is applied and the payload is not detected. On the other hand, as we explained in RQ1, Curious-Monkey could not trigger the samples' payload because of the limitation in the Xposed framework (**Constants2** sample) or limitation in the used emulator (**QemuFingerprinting** sample).

Table 5
Ares vs. Curious-Monkey: Triggering the payload

Sample name	Ares	Curious-Monkey
AccelH	Detected	Detected
AdbEnable	Detected	Detected
AdbPortDetector	Detected	Detected
AtNight	Detected	Detected
BatteryCharging	Detected	Detected
BatteryFull	Detected	Detected
BatteryStatus	Detected	Detected
ConstantCalls1	Detected	Detected
ConstantCalls2	Detected	Detected
Constants1	Detected	Detected
Constants2	Detected	Not detected
ConstantsDLC	Not detected	Detected
DivById	Not detected	Detected
GetIpAddress	Detected	Detected
InstalledApps	Detected	Detected
LongAction	Not detected	Detected
PostDelayed	Detected	Detected

b- Comparing with Monkey

In the second comparison, we compare Curious-Monkey with Monkey, the most used event generator in both general purpose and specific dynamic malware analysis domains. To do the job we set Monkey to run 20000 times. Each sample is executed five times by each approach, and the best result is logged. We used 100 samples randomly selected from 10 families in the AMD dataset. Figure 2 shows the amount of code coverage that is obtained from both Monkey and Curious-Monkey.

As shown in Figure 2, Curious-Monkey could achieve better coverage than Monkey in most cases. This improvement can be noticed as in **Koler**, **Fusob**, and **Triada**.

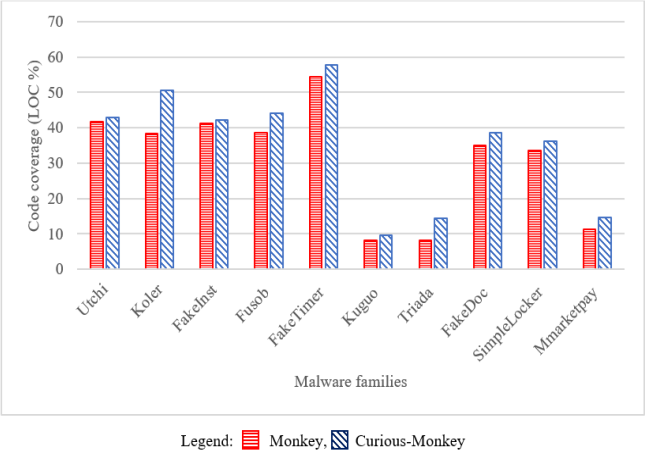

Fig. 2. Monkey vs Curious-Monkey: Code coverage

This is because in these samples generating system events and handling evasion techniques execute some parts of the code that cannot be reached by UI events. In other families, the improvement was little, as in **Uthci**, **FakeInst**, and **FakeTimer**. This is because the selected samples from these families mainly use UI events and had a small number of code lines. Therefore, the generated UI events are almost enough to execute as much as possible from these samples. Finally, in some families such as **Kuguo**, **Triada**, and **Mmarketpay**, both approaches achieved a little code coverage. This is because the samples in these families have complex activities that require complex interaction, such as filling some text fields with meaningful information then clicking the buttons, which cannot be achieved by Monkey. In addition to some guards that prevent executing the code and could not be handled by the proposed approach, such as triggering by attackers.

To measure the second metric (i.e., the number of sensitive API calls), Droidmon [22] is used. Note that the captured sensitive API calls reflect the malicious behavior of the sample under test. Table 6 represents the top 10 logged sensitive APIs called when using Curious-Monkey and the corresponding values in the case of using Monkey. Table 7 also represents the top 10 logged sensitive APIs, which are called when using Monkey, and the corresponding values in case of using Curious-Monkey.

Both tables show that more sensitive API methods could be extracted from Curious-Monkey, compared to Monkey using the same set of malware apps. For instance, the API method android.telephony.*TelephonyManager/getDeviceId* was logged from 20 malware samples when using Curious-Monkey in the analysis process and from 10 malware samples in case of using Monkey. This proves our initial hypothesis about the impact of system events and handling evasion techniques in triggering the hidden payloads (or part of it) in the malware code.

Note that the number of APKs, that we could log their corresponding sensitive APIs is 35, while in the case of Monkey, sensitive APIs from only 14 APKs are called. As a result, we can say that Curious-Monkey outperforms Mon-

Table 6
Top 10 logged sensitive API calls when using Curious-Monkey compared to those of Monkey

Sensitive API calls	Monkey	Curious-Monkey
android.telephony.TelephonyManager/getDeviceId	10	20
java.lang.reflect.Method/invoke	11	20
java.net.ProxySelectorImpl/select	8	17
android.telephony.TelephonyManager/getNetworkOperator	2	15
android.telephony.TelephonyManager/getSubscriberId	1	10
android.telephony.TelephonyManager/getLine1Number	0	10
dalvik.system.PathClassLoader	0	10
dalvik.system.BaseDexClassLoader/findLibrary	0	9
javax.crypto.Cipher/doFinal	1	9
dalvik.system.DexFile/openDexFile	0	9

Table 7
Top 10 logged sensitive API calls when using Monkey compared to those of Curious-Monkey

Sensitive API calls	Curious-Monkey	Monkey
android.telephony.TelephonyManager/getDeviceId	20	10
java.lang.reflect.Method/invoke	20	11
java.net.ProxySelectorImpl/select	17	8
javax.crypto.spec.SecretKeySpec	7	5
javax.crypto.Cipher/update	6	5
android.telephony.TelephonyManager/getNetworkOperatorName	5	4
java.net.URL/openConnection	5	4
android.telephony.TelephonyManager/getSimCountryIso	5	3
android.telephony.TelephonyManager/getNetworkCountryIso	5	3
dalvik.system.BaseDexClassLoader/findResource	4	3

key when we deal with dynamic malware analysis. Moreover, we should notice that triggering the payloads of 35 samples from 100 samples is relatively low. This is because many other factors help in triggering the malicious payloads, such as realistic UI events, and handling other outsmarting techniques that can be used to hide the malicious payloads. However, there is a trade-off between the speed of analysis and the ability to trigger the malicious payload. Since the proposed approach is random-based, it provides high speed in the analysis process with acceptable results in the case of triggering the malicious payload.

6. CONCLUSION

This paper presented a random-based approach, called Curious-Monkey, that can be used to generate events for dynamic malware analysis. Curious-Monkey integrates a random-based UI event generator tool, i.e., Monkey, with generating system events and handling evasion techniques. By doing so, we can leverage the advantage of Monkey to enhance the chance of triggering malicious payload in the code and to increase the obtained code coverage. The generated system events represent the most frequent events that are used by malware to trigger malicious payloads. Moreover, the handled evasion techniques represent the most used techniques by malware to hinder their malicious payload. We compared Curious-Monkey with both Monkey and Ares. The results showed that using system events and handling evasion techniques provide a considerable improvement in case of triggering malicious payload and the achieved code coverage. However, this is not sufficient to trigger malicious payloads because of some outsmarting techniques used by malware, such as triggering the payload by attacker instructions, or detecting the frequency of the generated events. We aim to handle these techniques as our future work.

References

[1] F. Wei, S. Roy, and X. Ou, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 3, pp. 1–32, 2018.

[2] L. Cai, Y. Li, and Z. Xiong, "Jowmdroid: Android malware detection based on feature weighting with joint optimization of weight-mapping and classifier parameters," *Computers & Security*, vol. 100, p. 102086, 2021.

[3] J. Alqatawna, A.-Z. Ala'M, M. A. Hassonah, H. Faris, *et al.*, "Android botnet detection using machine learning models based on a comprehensive static analysis approach," *Journal of Information Security and Applications*, vol. 58, p. 102735, 2021.

[4] C.-C. Hu, T.-H. Jeng, and Y.-M. Chen, "Dynamic android malware analysis with de-identification of personal identifiable information," in *2020 the 3rd International Conference on Computing and Big Data*, pp. 30–36, 2020.

[5] A. De Lorenzo, F. Martinelli, E. Medvet, F. Mercaldo, and A. Santone, "Visualizing the outcome of dynamic analysis of android malware with vizmal," *Journal of Information Security and Applications*, vol. 50, p. 102423, 2020.

[6] C. Jindal, C. Salls, H. Aghakhani, K. Long, C. Kruegel, and G. Vigna, "Neurlux: dynamic malware analysis without feature engineering," in *Proceedings of the 35th Annual Computer Security Applications Conference*, pp. 444–455, 2019.

[7] L. Singh and M. Hofmann, "Dynamic behavior analysis of android applications for malware detection," in *2017 International Conference on Intelligent Communication and Computational Techniques (ICCT)*, pp. 1–7, IEEE, 2017.

[8] R. B. Hadiprakoso, H. Kabetta, and I. K. S. Buana, "Hybrid-based malware analysis for effective and efficiency android malware detection," in *2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*, pp. 8–12, IEEE, 2020.

[9] Y.-C. Shyong, T.-H. Jeng, and Y.-M. Chen, "Combining static permissions and dynamic packet analysis to improve android

malware detection," in *2020 2nd International Conference on Computer Communication and the Internet (ICCCI)*, pp. 75–81, IEEE, 2020.

[10] H. Lockheimer, "Android and Security ." `http://googlemobile.blogspot.com/2012/02/android-and-security.html/`. [Online; accessed 30-May-2020].

[11] A. Developers, "Ui/application exerciser Monkey." `http://developer.android.com/tools/help/monkey.html/`, 2012. [Online; accessed 10-March-2020].

[12] S. Neuner, V. Van der Veen, M. Lindorfer, M. Huber, G. Merzdovnik, M. Mulazzani, and E. Weippl, "Enter sandbox: Android sandbox comparison," *arXiv preprint arXiv:1410.7749*, 2014.

[13] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 23–26, IEEE, 2017.

[14] X. Wang, Y. Yang, and S. Zhu, "Automated hybrid analysis of android malware through augmenting fuzzing with forced execution," *IEEE Transactions on Mobile Computing*, vol. 18, no. 12, pp. 2768–2782, 2018.

[15] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "Improving dynamic analysis of android apps using hybrid test input generation," in *2017 International Conference on Cyber Security And Protection Of Digital Services (Cyber Security)*, pp. 1–8, IEEE, 2017.

[16] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande, and V. V. T. Tong, "Grodddroid: a gorilla for triggering malicious behaviors," in *2015 10th international conference on malicious and unwanted software (MALWARE)*, pp. 119–127, IEEE, 2015.

[17] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel, "Making malory behave maliciously: Targeted fuzzing of android execution environments," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 300–311, IEEE, 2017.

[18] A. Developers, "Curious-Monkey." `https://github.com/hayyanHasan/Curious-Monkey/`. [Online; accessed 16-March-2021].

[19] L. Bello and M. Pistoia, "Ares: triggering payload of evasive android malware," in *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 2–12, IEEE, 2018.

[20] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 252–276, Springer, 2017.

[21] Y. Li, J. Jang, X. Hu, and X. Ou, "Android malware clustering through malicious payload mining," in *International symposium on research in attacks, intrusions, and defenses*, pp. 192–214, Springer, 2017.

[22] A. Developers, "Droidmon." `https://github.com/idanr1986/droidmon/`. [Online; accessed 30-May-2020].

[23] A. Pilgun, O. Gadyatskaya, S. Dashevskyi, Y. Zhauniarovich, and A. Kushniarou, "An effective android code coverage tool," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2189–2191, 2018.

[24] A. Developers, "Contagio Mobile Malware." `http://contagiomobile.deependresearch.org/index.html/`. [Online; accessed 5-March-2020].

[25] P. Carter, C. Mulliner, M. Lindorfer, W. Robertson, and E. Kirda, "Curiousdroid: automated user interface interaction for android application analysis sandboxes," in *International Conference on Financial Cryptography and Data Security*, pp. 231–249, Springer, 2016.

[26] H. Hasan, B. T. Ladani, and B. Zamani, "Megdroid: A model-driven event generation framework for dynamic android malware analysis," *Information and Software Technology*, p. 106569, 2021.

[27] M. Y. Wong and D. Lie, "Intellidroid: A targeted input generator for the dynamic analysis of android malware.," in *NDSS*, vol. 16, pp. 21–24, 2016.

[28] H. Hasan, B. T. Ladani, and B. Zamani, "Enhancing monkey to trigger malicious payloads in android malware," in *2020 17th International ISC Conference on Information Security and Cryptology (ISCISC)*, pp. 65–72, IEEE, 2020.

[29] G. Meng, "A semantic-based analysis of android malware for detection, generation, and trend analysis," *Ph. D. dissertation*, 2017.

[30] A. Developers, "Xposed Framwork." `https://github.com/rovo89/`. [Online; accessed 30-May-2020].

[31] A. Developers, "Logcat command-line tool." `https://developer.android.com/studio/command-line/logcat/`. [Online; accessed 30-May-2020].