MEGDroid: A Model-Driven Event Generation Framework for Dynamic Android Malware Analysis

Hayyan Hasan, Behrouz Tork Ladani, Bahman Zamani

MDSE Research Group, Faculty of Computer Engineering, University of Isfahan, Isfahan, Iran

Abstract

Context: The tremendous growth of Android malware in recent years is a strong motivation for the vast endeavor in detection and analysis of malware apps. A prominent approach for this purpose is dynamic analysis in which providing complex interactions with the samples under analysis is a need. Event generation tools are almost used to provide such interactions, but they have deficiencies for effective malware analysis. For example, anti-static and anti-dynamic analysis techniques employed by the malware prevent event generators to extract sufficient information for generating appropriate events. As a result, they fail to trigger malicious payloads or obtain high code coverage in most cases. 

Objective: In this paper, we aim to present a new framework to improve the event generation process for dynamic analysis of Android malware.

Method: We propose MEGDroid, a Model Driven Engineering (MDE) framework in which malware-related information is automatically extracted and represented as a domain-specific model. This model, then is used to generate appropriate events for malware analysis using model-to-model and model-to-code transformations. The proposed model-driven artifacts also provide required facilities to put the human in the loop for properly taking his/her knowledge into account.

Results: The proposed framework has been realized as an Eclipse plugin and we performed extensive practical analysis on a set of malware samples selected from the AMD dataset. The experimental results showed that MEGDroid considerably increases the number of triggered malicious payloads as well as the execution code coverage compared with Monkey and DroidBot, as two state of the art general-purpose and malware specific event generators respectively.

Conclusion: The proposed MDE approach, enhances the event generation process through both automatic event generation and analyzer user involvement who can efficiently direct the process to increase the effectiveness of the generated events considering small amount of information that is extractable from the malware code.

1. Introduction

Android is the most common operating system in mobile devices with a market share of 87% in 2020 [1]. As malware are major threat for Android, it becomes increasingly necessary to find ways to analyze Android malware in order to understand their behavior and to increase the ability to detect them. Dynamic analysis is a prominent approach for analyzing the behavior of Android apps. It includes running the code in a virtual environment (or in a real device in some cases) to understand its real behavior [2-9]. Event generation is an essential technique for analyzing the behavior of Android apps in general, because it represents the first step of the analysis process and the generated events are used to guide the dynamic analysis of the samples under test. This is, however, more critical for malware analysis, because malware generally need a combination of complex UI and system events to reach the malicious payloads and explore as much as possible from the malware code. In order to achieve such complex interaction, event generators need to extract information (that can be considered as sources for these events) from the malware, either statically or dynamically. However, there are some limitations for achieving this goal due to the anti-static and anti-dynamic analysis techniques that are usually used by malware to hide their information. This causes the generated events to be insufficient to explore the real behavior of the sample under test. Therefore, there is still an enormous demand for techniques that effectively generate events to exercise more code and activate more malicious payloads in Android malware.

In this paper, we propose a novel approach based on Model-Driven Engineering (MDE) [10-11] for automatically
generating events, specifically for dynamic malware analysis. In the proposed framework, called MEGDroid, we first use a Model-Driven Reverse Engineering (MDRE) approach to initially identify the sources of the potential raising events in the code. MEGDroid automatically extracts the available information about the event sources from the malware code and represents that information as a domain-specific model named Event Source Model (ESM). ESM represents the sources of the events in the malware code, such as the used views, the requested permissions, and the registered receivers, that are involved in generating events. ESM then is analyzed and transformed into another model, called Event Production Model (EPM), using model-to-model transformations. EPM represents the events that will be generated. Finally, MEGDroid uses model-to-code transformations to automatically generate the final working events from EPM. The proposed framework considers all necessary malware app components to extract the event sources from the code, and to generate both types of UI and system events. By UI events, we mean interacting with the activity components of the app under test, which must be identical to the human interaction as much as possible in the case of malware analysis. System events also represent any other events that the malware sample expects, such as environment conditions.

We believe that using a human-in-the-loop approach [12] can be helpful in case of complex interaction scenarios such as event generation for Android malware dynamic analysis. Indeed, the human knowledge should be incorporated into event generation process to make the malware dynamic analysis more effective. However, since manual event generation is always a time-consuming process and requires considerable effort, it has not been considered yet in previous works. One important advantage of using MDE approach in the proposed framework is that it can provide the required facilities for involving the analyzer (i.e., the user during the analysis process) to perform his/her dynamic malware analysis tasks as effectively as possible by adjusting and directing the event generation process. In fact, due to their high level of abstraction, the proposed model-driven artifacts provide proper representation for the generated events that enables the analyzer to easily modify the event generation process for reaching more achievable events considering the little extractable information. Moreover, since EPM is built automatically from ESM (i.e., EPM is not built from scratch by the analyzer), the analyzer involvement is restricted to just modifying EPM according to his/her knowledge about the sample under test. This greatly saves the time required for the analyzer involvement and makes it very efficient. MEGDroid can give the impression that the generated events are coming from human users rather than automatic tools which is very important in case of malware analysis that needs relevant and complex events rather than random events to trigger the malicious payloads.

To evaluate the proposed framework, we performed extensive experimental analysis using AMD malware dataset [13-14]. We chose 200 samples from 20 different malware families. The malware families are selected such that we can cover a set of diverse functionalities (from event generation perspective) including: a large/small number of activities, with/without activities, complex/simple views in each activity, having/lacking launchers, and having anti-reverse functionalities such as obfuscation, and dynamic load code. Moreover, thanks to the behavioral classification of malware in AMD dataset [13], we selected those families that have common malicious behaviors such as stealing device information, stealing personal information, and connecting to C&C servers. Moreover, all of these families use events to trigger their malicious payloads.

The proposed approach is evaluated and compared with Monkey [15] and DroidBot [16] that are two state of the art general-purpose and malware specific event generators respectively. Note that DroidBot is the only open-source tool that was available for us and includes prominent features and objectives that are comparable with MEGDroid. We consider code coverage, event generation performance, and the number of logged sensitive API calls as three important criteria in generating events for malware analysis. The experimental results show that MEGDroid provides better results than similar tools regarding the mentioned criteria. Comparing with other tools, MEGDroid generates a smaller number of events to reach its results, that shows the effectiveness and efficiency of the proposed approach.

The main contributions of this paper are as follows:

- A framework based on MDE approach to facilitate realizing the human-in-the-loop idea and using the human analyzer knowledge for efficiently directing the complex event generation process and hence increasing the effectiveness of the generated events.
- The Event Source Meta-model as a domain-specific modeling language that enables both modeling and extracting every possible event source from the malware code.
The Event Production Meta-model as a domain-specific modeling language that enables both modeling and generating different types of events as a response to the extracted sources.

Implementing the framework as an Eclipse plugin to show the feasibility and pertinence of the proposed approach. The plugin has been applied for practical analysis of 200 real-world Android malware as our experimental analysis.

The paper has been organized as follows: Section 2 presents the related work. Section 3 introduces a motivation example that we selected in order to demonstrate the research problem we worked on it. In Section 4 the proposed approach is presented in detail. In Section 5 the evaluation and the results of comparison with related works are presented. In Section 6 we discuss the limitations of the proposed approach. Finally, in Section 7 we conclude the paper.

2. Related work

Several tools are proposed to generate events for dynamic analysis of Android apps. We discuss them in two categories of general-purpose, and malware specific event generation tools. We first review the general-purpose tools and show that although these tools have many capabilities and some dynamic malware analysis frameworks use them, but they have several shortcomings that motivated some researches to develop malware specific event generators. Then we review the malware specific tools and discuss their capabilities and limitations. Finally, we conclude with the challenges that we are going to address using MEGDroid and our insight for solving the issues.

There are many general-purpose event generation tools for dynamic analysis of Android apps [15][17-23]. These tools use different approaches to generate events for Android dynamic analysis. The most used approaches are random based [15][17-19], and model-based [20-21] approaches. Monkey [15] is the most known general-purpose event generator for Android apps that is a random event generator introduced as a part of the Android developers’ toolkit. Monkey is easy to work with and provides high coverage for conventional apps. This tool is used to generate random UI events such as clicks, type, swipes, and screen touches. In general, the Android Debug Bridge (ADB) [24] tool is used to run Monkey and send the generated events to either the emulator or the real device. Although this tool is not a malware specific event generator one, many malware dynamic analysis frameworks [25-26] use Monkey as their underlying event generator. Dynodroid [17] is another tool that uses some improved random strategies to generate events for Android apps. This tool includes generating both UI and system events. Similarly, EHBDroid [18] generates UI and system events in a random manner.

Another interesting approach in generating events for Android dynamic analysis is the model-based approach [20-21]. In general, this approach first builds the GUI model of the apps and then generates events based on this model to explore the behavior of the apps. Adamant [20] is a tool that leverages MDE approach to generate both UI and system events for the analysis of Android apps. This tool includes models that involve valuable human knowledge about the apps under test for effective event generation process. The used models extended Interaction Flow Modeling Language (IFML) models to generate appropriate events. Adamant is similar to MEGDroid as it leverages human knowledge about the app under test to guide the event generation process. However, the used model is built from scratch which can be a time-consuming process. Moreover, the generated system events are related to the analysis of general apps and are not appropriate to trigger the malicious payloads in the code.

Although general-purpose tools show good code execution coverage when they are used in dynamic analysis of conventional Android apps, for the case of malware analysis they suffer from many drawbacks. For example, in the case of random based approach, the generated events are not relevant to each other which makes them not appropriate for simulating human behavior. Other approaches sought to make the generated events realistic as much as possible and provide a human-like impression. In order to make the generated events more relevant, they need to extract information from the code that is done via static analysis or instrumentation in most cases. This, in turn, brings in a drawback for the case of malware analysis because many malware authors use anti-static analysis or integrity checking techniques to prevent extracting information from the code. Therefore, general-purpose tools have many limitations when they used for malware analysis. This motivates the proposals for malware specific event generators.

The existing malware specific event generators, like general-purpose event generators, focused on providing human impressions in the generated events. However, to generate realistic events, they use dynamic instrumentation to extract the required sources of the generated events. DroidBot [16] is a model-based tool for generating appropriate UI events for
Android malware analysis. This tool is easy to use like Monkey but smarter than it when dealing with malware apps. DroidBot utilizes hooking to extract information from the code to generate appropriate UI events. In [27] the authors used a hybrid approach by integrating random-based approach (Monkey) with model-based approach (DroidBot) to improve the code coverage and trigger the malicious payloads in the code.

AppsPlayground [28], provides techniques for automatically executing Android apps. It includes generating UI and system events. System events are generated based on the permissions requested by the sample under test or the detected broadcast receivers. In the case of UI events, AppsPlayground extracts the sources of the UI events dynamically and provides the ability to determine the context from the UI elements to direct the execution of apps effectively. For example, it uses hints and other keywords from the UI elements to determine the text to be inserted or to determine the most appropriate button to be clicked. Curiousdroid [29] is another related tool that makes use of dynamic instrumentation, application layout description, and heuristic method to generate events in a human-like manner. Curiousdroid attempts to generate intelligent UI events by determining the context from the UI widgets using the same techniques in AppsPlayground.

Note that the aforementioned approaches focus on making UI events more realistic to simulate the human interaction as much as possible. Since these tools use dynamic instrumentation to get the sources of the generated events, there is no guarantee to detect all the event sources. This effects on the quality of the generated events. Moreover, the generated events are restricted to simple interaction, such as touch screen and click buttons, in most cases. Finally, most of these tools are used with dynamic analysis frameworks as their main UI event generators. For example, DroidBot is introduced with DroidBox [30] and Dynalog sandboxes [4], while Curiousdroid is introduced with Andrubis sandbox [25]. These sandboxes handle the generation process of the system events. However, the generated system events are generated based on the sources extracted from the code statically in most cases, which imposes some limitations because of the anti-static analysis techniques used by the malware.

GroddDroid [31] is another event generator that considers the anti-emulation capabilities of the malware. It forces the executing path to the malicious payload by modifying the malware code. However, this tool makes use of GUI runner, this runner is based on uiautomator [32] which is a python wrapper that is used for testing purpose. The current version of the GUI runner can consider buttons without other views such as checkboxes, and input texts, which makes it not sufficient to generate appropriate events.

As elaborated in the above paragraphs, most of the previous approaches are not effective enough to provide appropriate events to trigger the malicious payloads in the Android malware code, because they generate events randomly. Moreover, they are mostly not able to generate complex events such as clicking a specific button after filling text field with appropriate input. Finally, in most cases they do not have the ability to extract sufficient information to generate appropriate events.

Table 1 summarizes the characteristics of main aforementioned event generation tools including their specific limitations.

In this paper, we focus on the above-mentioned problems and propose a model-driven framework to improve the effectiveness of event generation process for dynamic analysis of Android malware. The proposed framework automatically extracts the malware related information and represents it as a domain-specific model which is used to generate appropriate UI and system events. The proposed model-driven artifacts provide the required facilities to involve the analyzer user in the event generation process for properly taking his/her knowledge about the sample under test into account.

3. Motivation example

To better demonstrate our research problem, in this section we introduce a real-world malware sample and discuss the designated challenges of events generation. The sample is an instance of a well-known Android malware family named Koler. Koler is a family of Android ransomware that locks the device until a ransom is paid. It shows a screen from some law agency which is selected according to the location obtained from the device and asks the user to pay for illegal use of the device. This screen will not be removed by pressing Home or Back buttons and repeats within a short time if dismissed. To unlock the device and remove this screen, an amount of money between 100$ to 300$ must be paid.
Table 1. A comparison among the main existing event generators

<table>
<thead>
<tr>
<th>Tool</th>
<th>Event generator type</th>
<th>Generated events</th>
<th>UI Event generating approach</th>
<th>Information extraction method</th>
<th>Analyzer involvement</th>
<th>Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UEG, UCE</td>
</tr>
<tr>
<td>Monkey [15]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Random</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Dynodroid [17]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Model-based</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>EHBDroid [18]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Static</td>
<td>✓</td>
<td>UEG, UCE</td>
</tr>
<tr>
<td>Adamant [20]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Dynamic</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>DroidBot [16]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>BMS</td>
</tr>
<tr>
<td>Alzaylaee et al. [27]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>UEG, UCE</td>
</tr>
<tr>
<td>AppsPlayground [28]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>UEG, UCE</td>
</tr>
<tr>
<td>Curiousdroid [29]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>USI</td>
</tr>
<tr>
<td>GroddDroid [31]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>MEGDroid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>UCE</td>
</tr>
</tbody>
</table>

* UEG: Unrelated Events Generation, BMS: need to Build Model from Scratch, UCE: Unable to generate Complex Events, USI: Unable to extract Sufficient Information.

Fig. 1, shows the payment page of Koler malware instance that requests inserting the card information to unlock the device. Other Android ransomware also may use similar payment pages. Note that filling this type of pages requires knowledge about the payment method to insert meaningful information (instead of just random information). Hence, automatic event generation tools will not be able to insert correct information.

Fig. 1. The payment page in Koler malware sample

In this sample, the used payment method is Walmart’s money card, so the automatic event generators need to insert some meaningful information related to the payment method to provide a human impression. However, when analyzing this sample using Monkey and DroidBot, both tools reach this page, but since these tools provide simple interaction, such as touch and press events, they are not able to insert meaningful payment information to cover the execution of more malware codes. This example shows that in some cases, the human knowledge is very important to insert appropriate events and hence to execute more lines in the malware code.

4. The Proposed Framework

MEGDroid uses an MDE approach for automatically generating events, specifically for dynamic malware analysis. In MDE, models play the most important role. Hence, it aims at abstract representations of the knowledge and activities that govern a particular application domain rather than the algorithmic concepts [10]. MEGDroid first extracts a specific domain model named Event Source Model (ESM) from the given code using the Model Driven Reverse Engineering (MDRE) approach. ESM is an abstract representation of sources of the potential raising events in the code such as the used views, requested permissions, and registered receivers that are involved in generating both UI and system events. ESM is then analyzed and transformed into another domain model named Event Production Model (EPM) using a model-to-model transformations techniques. EPM is an abstract representation of the working events that will be then generated automatically in concrete form using a model-to-code transformations. The reason for adopting two separate models is to make it easy for the analyzer to add or adjust the generated events without changing the sources of these events. Both ESM and EPM are specified using their corresponding meta-models.

In the following subsections, the MEGDroid framework architecture, the used meta-models (Event Source Meta-model
and Event Production Meta-model), the specification of the required transformations, and the analyzer involvement process are described.

4.1. MEGDroid Architecture

Fig. 2 represents the MEGDroid framework architecture. The framework consists of the following five steps that are performed in two levels: model level (where model is the main artifact) and code level (where code is the main artifact):

- **Step 1. Decompilation (code level):** The input to this step is the binary malware APK file and the output will be Java and XML files of the malware. The decompilation process is done using Jadx [33] which is a tool for generating source codes from Android DEX and APK files.

- **Step 2. Model discovery (code level):** The input of this step is the XML and Java files extracted from the first step, and the output will be XML and Java models. These models are extracted using MoDisco discoverer [34]. MEGDroid uses two types of discoverers: first, it discovers XML model from AndroidManifest.xml file using MoDisco XML discoverer, and then it uses MoDisco Java discoverer to generate the Java model from the Java files. These two models contain useful information that can be used in the next steps.

- **Step 3. Integration and transformation (model level):** In this step, the ATL transformation language [35] is used to integrate the two models obtained from the previous step into ESM. ESM conforms to its corresponding meta-model, i.e., the Event Source Meta-model, and includes all possible information about the sources of the events.

- **Step 4. Analysis and transformation (model level):** In this step, the ESM obtained in previous step is analyzed and transformed into the EPM using ATL transformation language. EPM includes all the events that will be generated as a response to the information extracted from the malware APK. EPM conforms to its corresponding meta-model, i.e., the Event Production Meta-model, and includes all possible events that will be generated. In general, malware make use of many techniques such as obfuscation and integrity check to hide their execution related information. Therefore, we may sometimes tackle with little extracted information. In such cases, the event generators may fail to produce human-like events. Our solution, in this case, is to provide the analyzer with a usable representation for the generated events using EPM to let him/her compensating the lack of information, based on his/her knowledge about the sample under test, by enriching or adjusting EPM which affect the generated events directly.

![Fig. 2. MEGDroid architecture](image-url)
• **Step 5. Events production (code level):** In this step, the working events will be generated from the EPM, automatically. This is achieved by using the Accelero code generator [36].

As shown in Fig. 2, in our framework we have also used an adapter component to automate the events transformation process. The adapter component acts as a bridge between MEGDroid and the test environment (i.e., the emulator or real device) to transfer the events generated by the tool to the test environment. This transformation is done using Android Device Bridge (ADB) which is a tool developed by Google to facilitate communicating with the emulators (or real devices). In addition to transfer events to the test environment, the adapter component loads the malware APK file into the test environment and setup the environmental conditions. For example, it checks the Internet connection or the airplane mode of the device. These settings may be sometimes very important because many malware samples need to connect to C&C servers or need to send and receive calls and SMSs. The adapter also maintains a log file of the whole testing process activities to gather the logged sensitive API calls.

### 4.2. Meta-models

In MDE terminology, a meta-model is a model that defines the abstract syntax of a modeling language [10]. A meta-model shows all the concepts in the language and their relationships. Since MEGDroid includes two models, ESM, and EPM, it uses two meta-models, correspondingly. In this section, we describe the corresponding meta-models in detail.

#### 4.2.1 Event Source Meta-model

This meta-model defines the model for representing the components of the code and the potential sources of the generated events. The generated events are either UI events or system events. To design this meta-model, and to understand the potential sources of each type of events, we used the Android programming technical information such as works of Mednieks et al. [37], Thornsby et al. [38], and the official Android website [39]. The designed Event Source Meta-model is shown in Fig. 3. In the following, the elements used in this meta-model are described:

- **SysModel:** This component is the root of the meta-model. It includes the malware PackageName, which is known as a unique identifier.

- **UsesPermission:** Android malware generally ask to access some device resources. They need to receive the corresponding permissions from the user to access the resources. Permissions are defined in the AndroidManifest.xml file under the <uses-permission> tag. This tag may contain many attributes, the Name of the requested permission is sufficient to generate appropriate events as a response to the requested permission.

- **Activity:** An Android activity is a component that appears to the user and he/she can interact with it. Some malware apps need to interact with the user, e.g., via clicking some buttons, to trigger their malicious payload. In order to simulate the user interaction, we need to extract ActivityName and all the views that are included in the malware code.

- **View:** An activity element may contain many views. We extract the ID, Text, and name attributes for each view to make the production of the UI events more realistic. The views are grouped into the following categories:
  - **TextView:** that represents the extracted text views.
  - **InputButton:** that represents buttons, toggle buttons, and image buttons.
  - **InputText:** that represents plain texts and password texts. There are also other attributes in an InputText element that we can use, such as Hint and InputType. This information is required to insert the appropriate string as a response to this view.
  - **InputChecks:** that represents radio buttons, and checkboxes.
  - **Menu:** that represents options, menus, and spinners.

- **BroadcastReceiver:** This is a component in the Android app that is used to listen to the events from different outlets and it is invoked when a certain action has occurred and it has been programmed to listen to it. In general, this component is the most used component in malware code to trigger the malicious payload. An example of using this component is that many malware samples require the device to be rebooted to start the malicious behavior, so the code
must contain a broadcast receiver which is invoked whenever the device is rebooted.

- **Service**: An Android service is a component that is used to perform operations on the background. Many Android malware insert malicious payloads in their service components [40] and in some cases generating system events or UI events may not be sufficient to launch the services. Therefore, we considered service component to add more flexibility to the analyzer for handling different components of the code. To handle the services, we need to extract the *Name* of services from the code.

- **IntentFilter**: Each component in an Android app (i.e., Activity, Broadcast Receiver, and Service) can have one or more intent filters that can be used to specify the kind of operation that the component can do. The *Name* attribute of this element is sufficient to generate a relevant response. In general, intent-filter can have the following three elements:
  o **Action**: this element can be used to determine the operation to be executed. It is worth to mention that each intent filter must contain at least one *Action* element. The *Name* of the *Action* is sufficient to generate an appropriate response.
  o **Data**: this element determines the data to be acted on, these *Data* can be referenced using *URI*.
  o **Category**: this element contains additional information about the kind of component that should handle the intent. The *Name* of the *Category* is sufficient to generate an appropriate response.

**4.2.2. Event Production Meta-model**

The schema of the Event Production Meta-model is illustrated in Fig. 4. This meta-model defines different types of events that can be generated. These events are divided into UI events and system events in addition to handling each detected component in the code. In order to design this meta-model, we have used the corresponding Android programming technical sources including the works mentioned in the previous section, as well as that of Xue et al. [41]. The elements used in the Event Production Meta-model are as follows:

- **GeneratedEvents**: This component is the root element of the Event Production Meta-model, which includes **UIEvents**, and **SystemEventsAndServices** elements.

- **UIEvents**: This element represents the UI events. It includes *PackageName* to represent the name of the package to be launched by MEGDroid.

- **Component2Run**: This element represents the activity to be launched. It includes *ActivityName* attribute to represent the name of the activity to be launched. MEGDroid generates the following **GeneratedUIEvents** events:
  o **Touch**: this means generating touch events. To generate the *Touch* events, the *startX*, *startY*, and the *NumberOfTouches* are needed.
  o **Drag**: this means generating swipe or scroll events. To generate the *Drag* events, the *startX*, *startY*, *endX*, *endY*, and the *NumberOfDrags* are needed. If we need to scroll the activity down for one time, the *NumberOfDrags* is set to 0, and the other attributes are set in such away the *StratY* is bigger than *StartX*.
  o **Type**: this means filling up the *InputText* element from the ESM. It includes the *EventName* attribute to represent the event to be generated. In fact, the inserted text is not randomly generated, but we use information extracted from the APK file to generate an appropriate string for the *InputText* element. This information represented in ESM and includes *ID*, *Text*, *Name*, *InputType*, and *Hint*. Moreover, there is an ability for the analyzer to insert an appropriate string for it according to his/her knowledge.
  o **Press**: this means generating press events for the *InputButton*, *InputChecks*, and *Menu* elements in the ESM. This element has two main attributes: *KeyName* to identify the name of the key to be pressed, and the *Type* to identify which element is going to be pressed.
  o **LongPress**: it includes performing Long press events at any point in the activity. To achieve the *LonPress* event we need to determine the *startX*, and the *StartY* in which the Long press will be applied.
- **SystemEventsAndServices**: This element is responsible for generating system events and handling the services. It includes `PackageName` to represent the name of the package to be launched by MEGDroid. **SystemEventsAndServices** has the following elements:
  - **ServiceBased**: this element handling the services extracted from the malware code. The name of the service is represented in the `Name` attribute, while the `EventsName` represents the event that will be generated.
  - **PermissionBased**: these events are generated after analyzing the permission requested by the malware. The `Type` attribute in this element represents the type of events that will be generated as a response to the requested permissions. MEGDroid provides the following **PermissionBased** events:
    - Phone State: means changing the device phone state. It includes changing the service state, data activity, data connectivity, and settings of the device,
    - Add contacts to the device,
    - Make calls to add call logs,
    - Add SMSs,
    - Add browser history and bookmarks.
    - Add calendar events,
    - Add Files,
  - And add accounts such as google account to the device.
  - **ReceiverBased**: this element represents the events that will be generated as a response to the broadcast receivers extracted from the malware code. The generated events include sending intents based on the intent filters used in the extracted broadcast receivers. These intents defined in the `EventsName` attribute, while the `PredifinedIntents` attribute represents a predefined intent that the analyzer can add manually.
  - **IntentFilter**: this element has `Action`, `Data`, and `Category` attributes that are required to generate events based on the registered receiver.

The order of the generated events is based on the information extracted from the malware code. MEGDroid first generates system events and force the service execution, then generates UI events. The order of the generated UI events is based on the type of the component extracted from the malware code and some other information represented in ESM such as `name`, `id`, and `label`. First, the **Type** events will be generated, while the last generated events are pressing the buttons. On the other hand, if there are two components that have the same type e.g., two buttons, we use the attributes such as `name`, `text`, and `ID` to determine which button will be clicked first. It is worth to mention that the analyzer can change the order of events or add some new events easily according to his/her knowledge about the sample under test.
4.3. Transformations

MEGDroid uses two types of transformations, model-to-model transformations which is achieved using ATL transformation language [35], and model-to-code transformations which is achieved using Acceleo code generator [36]. In the following, we illustrate these transformations in more detail.

4.3.1. Model-to-model transformations

We use model-to-model transformations in steps 3 and 4 of the framework illustrated in Fig. 2, where the Java and XML models are transformed and integrated into ESM, and then this model is transformed into EPM. We used ATL transformation language to achieve such transformations. The ATL code used in step 3 contains 17 helpers and 22 rules. As an instance, Listing 1 represents one of the written rules used in the transformations code that maps the Root element in the XML model to SysModel in ESM. From this list, we can see that lines (3 to 6) show the source (Root) and the destination (SysModel) elements. while lines (7 to 11) show how the PackageName attribute in the SysModel element is generated. Finally, lines (12 to 18) represent how the relationships in the SysModel element are generated.

```
1 -- Root to SYSModel transformation rule
2 rule Root2SYSModel {  
3       from  
4       s: XML!Root (s.name = 'manifest')
5   to  
6     t: EventsDesc!SysModel (  
7       PackageName < - if s.getAttribute('package').oclIsUndefined()  
8         then OclUndefined  
9         else s.getAttribute('package').value  
10      endif,  
11       activities < - $children->  
12       select(c | c.name = 'application').first().children - >select(c | c.name = 'activity'),  
13       service < - $children->  
14       select(c | c.name = 'application').first().children - >select(c | c.name = 'service') ,  
15       receiver < - $children->  
16       select(c | c.name = 'application').first().children -> select(c | c.name = 'receiver'),  
17       usesPermission < - $children -> select(c | c.name = 'uses-permission')  
18      )  
19   )
20 }  
```

Listing 1: ATL rule for mapping Root element in XML model to SysModel in ESM

ATL transformation also used in step 4. The used ATL code contains 23 helpers and 12 rules. The rules in this code are used to map the sources, represented in ESM, to events, represented in EPM. The helpers also are used to make the generated events more realistic. As an instance, Listing 2 represents one of the helpers used in the ATL code of step 4. This helper indicates the most used phone words, hence MEGDroid checks the attributes of the detected InputText in ESM i.e., (ID, Text, name, Hint, and InputType). If any of these attributes contains any word represented in the helper showed in Listing 2. The generated events as a response for this
detected **InputText** is typing a phone number. As another example, **Listing 3** shows two rules from the ATL code of step 4. These rules create events as a response to the registered receiver.

```plaintext
1 -- Phone words helper
2 helper def:PHONE WORDS: Sequence(OCLAny)=
3     Sequence {'phone', 'number', 'cell', 'home', 'fax',
4           'work', 'office', 'TYPE_CLASS_PHONE'}
5 );
```

**Listing 2:** ATL code of phone words helper

```plaintext
1 -- BroadcastReciver to ReciverBased transformation rule
2 rule creatReciver { from
3     s: EventsDesc!BroadcastReciver to
4         s: EventsDesc!BroadcastReciver
5         t:EventsGen!ReciverBased ( EventsName <- 'adb shell am broadcast -a',
6             intent_filter <- s.filters )
7 }
8 -- IntentFilter in ESM to IntentFilter transformation rule
9 rule createFilter { from
10     s:EventsDesc!IntentFilter ( not s.ReceiverName.oclIsUndefined() ) to
11     t:EventsGen!IntentFilter( Action <- s.action.first().Name ,
12         Category <- if not s.category.isEmpty() then
13              s.category.first().Name
14          else
15              OclUndefined
16          endif,
17         Data <- if s.data.oclIsUndefined() then
18             s.data.first().URI
19          else
20              OclUndefined
21          endif)
22 }
```

**Listing 3:** ATL rules for creating **ReciverBased** events as a response to the registered receiver

As can be seen from **Listing 3**, there are two rules, the first rule is responsible for generating the **ReciverBased** element (lines 2 to 10), and the second rule is responsible for generating the **IntentFilter** element that is necessary for the **ReciverBased** element (lines 12 to 29). The first rule generates **EventsName** attribute of the **ReciverBased** (line 7). While the second rule generates the **Action**, **Category**, and **Data** attributes of the **IntentFilter** element (lines 17 to 27).

### 4.3.2. Model-to-code transformations

Model-to-code transformations is used in step 5 of the framework represented in **Fig. 2**. This type of transformations is achieved using Acceleo code generator. Acceleo is an open-source, template-based code generator developed inside the Eclipse foundation. Using the template-based approach allows Acceleo to generate any text from the model, such as Java code, and text files. We created two Acceleo transformations. The first transformation is responsible for generating working system events and the actual instructions to handle the services. While the second transformation is responsible for generating working UI events and the actual instructions to handle the activities. The outputs of these transformations are two text files that are read by the adapter to send events to the emulator.

**Listings 4** shows a part of the Acceleo code that are used to generate system events. As can be seen, this code is used to generate the response of the registered receivers (i.e., generating the actual events from the **ReciverBased** element in the EPM). The response depends on the defined intent filters for the registered receivers. In other words, it includes sending intents with action, category, and data elements identical to those defined in the intent filter. However, in some cases not all the three elements (i.e., action, category, and data) are defined, so the response must consider the defined elements only. For example, if the intent filter has an action element, the **Action** attribute will be considered in the response (lines 3 and 4). If the intent filter has action and data, these two elements will be considered in the response (lines 6 and 7). If the intent filter has action, category, both **Action** and **Category** will be considered in the response (lines 9 and 10). Finally, if the three elements of the intent filter (i.e., action, category, and data) are defined, they will be considered in the response (lines 12 and 13).

```plaintext
1 [for (rec : ReciverBased | rec.reciver_based)]
2   [for (inte : IntentFilter | inte.intent_filter)]
3     [if (inte.Category.oclIsUndefined() and inte.Data.oclIsUndefined())]
4   [for. EventsName/ inte.Action/]
5     [if]
6     [if (inte.Category.oclIsUndefined() and not inte.Data.oclIsUndefined())]
7     [for. EventsName/ inte.Action/ -d inte.Data/]
8     [if]
9     [if (inte.Data.oclIsUndefined() and not inte.Category.oclIsUndefined())]
10    [for. EventsName/ inte.Action/ -c inte.Category/]
11   [if]
12    [if (not inte.Data.oclIsUndefined() and not inte.Category.oclIsUndefined())]
```

---

11
Listing 4: Part of the Acceleo code that is used to generate events as a response to the registered receiver

Listing 5 represents a part of Acceleo code that is responsible of generating Type events as a response to the detected InputText in the malware code. The response is an ADB form for inserting appropriate text (line 3), the appropriate text is generated during step 4 in the framework represented in Fig. 2, as illustrated before.

Listing 5: Part of the Acceleo code that is used to generate Type events

4.4. Analyzer involvement

In general, the ultimate goal of the event generation tools is to achieve as high code coverage as possible in as little as possible time. In the case of malware analysis, the event generators seek to provide appropriate events to trigger the malicious payloads in the code in a short time. In order to do that, the proposed event generation tools try to simulate the behavior of the mobile user as much as possible. In order to provide human-like interaction, they need to extract sufficient information from the code, which imposes some limitations because of the techniques used by malware to hide this information. The solution for this problem is by using manual event generation process i.e., let the user interact with the apps. However, this process is very time consuming and requires large efforts. Using MDE approach we can handle the aforementioned problem, and generate more complete and human-like events with less time and effort.

A basic benefit of the proposed approach is that it provides a high level of abstraction for both sources and generated events. This can provide the analyzer user with a preliminary vision about what events should be generated to achieve the best result. Moreover, it provides the ability to use the analyzer knowledge in general and about the sample under test specifically to effectively direct the event generation process. The proposed approach handle generating a large number of events automatically by transforming ESM into EPM using model-to-model transformations. In fact, we use reverse engineering as a static analysis method to extract as much information as possible from the malware code, this information can be used to generate appropriate events. Hence, the analyzer user does not need to build the EPM from scratch and his/her involvement is restricted to adding, deleting, or reordering some elements in the EPM which considerably saves time and effort. According to our experimental analysis, the average time needed for modifying EPM by the analyzer for the selected samples is 2 minutes.

Back to the motivation example illustrated in Section 3, we argue that analyzer knowledge about the payment method is required to provide meaningful information for generating events in case of the Koler malware sample. Fig. 5 shows a sample scenario of EPM that can be used to insert appropriate UI events for interacting with the page illustrated in Fig. 1.

As can be seen from Fig. 5, we first need to send press key tab event to select the first editable text illustrated in Fig. 1 (i.e. the CARD editable text), then the card information that includes meaningful16-digit number should be inserted. After that, a press key tab event is needed to move to the next editable text. After filling the editable texts, we need to insert a press key enter event to click the send button illustrated in Fig. 1.

It is worth to mention that analyzer involvement is not restricted to the UI events only, it includes modifying the system events by adding/ deleting/ or reordering the generated events based on his/her observation of the requested permissions, and registered receivers or according to his/her knowledge about the sample under test. Moreover, in some cases, the malware samples do not contain a main activity to be launched. Automatic event generators such as Monkey and DroidBot will fail to launch this type of samples. Using MEGDroid, the analyzer can determine the order of executing each detected activity and determine the activity to start with. In fact, MDE approach provides an easy and flexible method to
the analyzer to manipulate the generated events for better analysis results.

It should be noted that MEGDroid, like any other model-based approach, suffers from the possibility of having incomplete or incorrect input models that will cause incomplete or incorrect outputs. However, to deal with this problem we use two separate solutions. First, in the proposed Event Production Metamodel we tried to define all possible types of events in order to simulate human interaction, and since EPM conforms to this Event Production Metamodel, it implies that EPM can have every possible event to simulate the user interaction. Second, we put the human in the loop that is a key characteristic of our work. In our work, model driven approach provides a high-level abstraction that allows the analyzer to adjust and/or complete the model in order to generate complete/correct events. In fact, using MDE techniques allows the analyzer to adjust and complete the missed elements in EPM that could not be generated automatically.

Finally, since we make use of static analysis to extract the sources of the events, we may face some limitations because static analysis is unable to determine the mistakes in the generated events, or it could not determine whether or not the generated events are enough. Also, in some cases some important events can be neglected. However, due to using the human in the loop concept in the proposed approach, some of these mistakes can be detected and corrected by the analyzer user. Indeed, the analyzer can check and complete the EPM according to his/her knowledge which can almost solve the issues.

We will experimentally show the effectiveness of analyzer involvement in completing or correcting the event production model in the next section (under RQ4 experiment).

5. Experimental Evaluation and Comparison

To evaluate MEGDroid, we implemented the framework as an eclipse plugin tool. We used MoDisco to extract the model from the code, ATL to achieve model-to-model transformations, and Acceleo for model-to-code transformations. In this section we performed extensive experimental evaluations to address the following research questions:

RQ1) How effective is MEGDroid?
RQ2) How efficient is MEGDroid?
RQ3) How does MEGDroid compares with other Android event generator tools?
RQ4) What is the impact of analyzer involvement on the event generation process?

5.1 Experiments Environment Setup

To do experiments for evaluation, analysis of the impacts, and comparing the proposed approach with other related tools, we used Genymotion emulator [42] and installed Google Nexus 5 image with Android version 5.0 Lollipop and API level 21 on it. We used a system with a Core i7 processor - 3.70 GHz speed, and 32 GB of RAM memory. The emulator was connected to the Internet to allow the malware to connect to their C&C servers or to perform any network operation during the experiments.

Since we are dealing with malware, there is a big likelihood of existing evasion techniques (such as anti-emulation techniques) along the execution paths. Therefore, these techniques may hinder the execution of the payload in sandboxes or virtual environments. We have to take the anti-emulation techniques into consideration for the malware cases used in our experiments. For this purpose, we handle these techniques dynamically using the Xposed framework [43]. This Framework roots the test environment (in our case Genymotion emulator) and provides the ability to create some modules to hook the invoked methods during the execution. Hence, we wrote an Xposed module that handles the anti-emulation techniques used by the selected families of malware in our experiment and use it for the three tools. We provide the module with the common API methods that are used to get information about the test environment such as getDeviceId, and getSubscriberId. The module hooks these API methods, and sets the returned results of these invoked methods with values similar to the real device values. For example, if generating some events leads to an execution path that includes the getDeviceId method, the module will hook this method when it is executed and returns the value 351451208401216 instead of 000000000000000 (in case of the emulator). This way, these special anti-emulation cases along the execution paths are handled.

Finally, before running each sample, a new clone of the emulator is generated to ensure that the analyzed malware samples are not affected by each other.

We used AMD dataset [13-14] for the evaluation process. We selected 20 families of malware that use events to trigger malicious payloads, and have common malicious behaviors such as stealing device information, stealing personal information, and connecting with C&C servers. Moreover, the selected families cover a diverse set of functionalities (from the perspective of event generation) to show the challenges responded by the proposed tool. Some selected families contain
obfuscated samples, or samples with payloads dynamically downloaded which make static analysis, (and statically extracting information) difficult or even impossible. Also, some families include samples that have a large number of activities with different views in each activity, do not contain any activity, or do not contain main activity or launcher category. Finally, for the sake of fairness in the evaluation, some of the selected families contain small number of activities with simple views in each activity, which do not require any complex events to execute the sample under test. Note that, we randomly selected 10 samples from each family. Table 2 represents the selected families used in the experiments for answering the research questions 1, 2 and 3.

Table 2. The used malware families in the experiments for answering the research questions 1, 2 and 3

<table>
<thead>
<tr>
<th>Family</th>
<th>Code coverage</th>
<th>Number of generated events</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) BankBot</td>
<td>8) Fakelnst 15) GingerMaster</td>
<td></td>
</tr>
<tr>
<td>2) Bankun</td>
<td>9) Kuguo 16) Utchi</td>
<td></td>
</tr>
<tr>
<td>3) Dowgin</td>
<td>10) SlennBunk 17) SimpleLocker</td>
<td></td>
</tr>
<tr>
<td>4) Stealer</td>
<td>11) Fusob 18) Mecor</td>
<td></td>
</tr>
<tr>
<td>5) Svpeng</td>
<td>12) FakeDoc 19) Triada</td>
<td></td>
</tr>
<tr>
<td>6) Cova</td>
<td>13) Mmarketpay 20) FakeTimer</td>
<td></td>
</tr>
<tr>
<td>7) GoldDream</td>
<td>14) Koler</td>
<td></td>
</tr>
</tbody>
</table>

5.2. Measures

To address the effectiveness property in the aforementioned research questions, we adopt the metrics introduced in [20] with respect to the Android malware analysis process:

1- Code coverage: This metric is used by both general-purpose and malware specific event generators, illustrates the percentage of the malware line of code (LOC) that is executed through the generated events. We used ACVtool [44] to measure the code coverage in our experiments.

2- Number of sensitive API calls: This metric is used to measure the ability of the event generator to trigger the malicious payload (or part of it) in the code. Note that the selected families from AMD dataset use events to trigger the malicious behavior. Moreover, the selected families have common malicious behaviors as classified in [13]. For this purpose, we studied the malicious behaviors of the selected families and select the sensitive APIs that are related to these behaviors. Hence, when these APIs are invoked, we can say that the whole payload or part of it is activated. This metric is almost used by malware specific event generator tools to evaluate the effectiveness of their generated events [27][31]. We used Droidmon [45] to capture the number of sensitive APIs invoked by the malware.

Moreover, the efficiency of the event generation tools is generally measured by computing the number of generated events per time. However, in the case of malware analysis, malware can detect injecting events in a high frequency and hide the malicious payloads accordingly [46]. As a result, to measure the efficiency of the proposed approach, we defined another criterion, Code Coverage Performance (CCP), that is given using the following equation.

\[
CCP = \frac{\text{Code coverage}}{\text{Number of generated events}}
\]

Note that, we plotted the code coverage and the CCP by malware families because the samples in each family did not have so different results, but the results between the families were different. We ran factorial ANOVA test to support this hypothesis. The test was achieved using IBM SPSS version 16.0 software [47] for both code coverage and CCP metrics in all experiments. Prior to conducting the ANOVA test, we checked the samples in each family and it was confirmed that there were no distal points (extreme values) by using Boxplots tool in SPSS, so the results of samples in each family were close to each other. After that, we conducted two ANOVA tests. First, we conducted one-way ANOVA test to show the differences between families for both metrics (i.e., the average of code coverage and CCP) in each scenario individually. The P values that determine whether the hypothesis test results are statistically significant or not, were less than 0.05 and show that the test results are statically significant. Then, we ran two-way ANOVA test to see if the average of code coverage and CCP for the selected families differs between tools or differs in the case of analyzer involvement experiment. In the case of tools, the P-values (for both metrics) were less than 0.05 which implies that there are differences between families for all three tools. Moreover, in the case of with/without analyzer involvement, the P-values were less than 0.05 for both metrics, which implies that there are differences between families in the case of analyzer involvement scenarios.

5.3. Results

In this section, we answer the research questions and reports the result of experiments.

RQ1) Effectiveness of the proposed approach
To measure the effectiveness of MEGDroid, we used the code coverage and the number of sensitive API calls as metrics.

**Code coverage**

Fig. 6 shows the code coverage achieved by MEGDroid for the families represented in Table 2. The generated events cause to achieve a code coverage (measured by LOC %) varied between 9.16 % and 64.62% that is reasonable in case of malware analysis. The family with the highest code coverage is FakeTimer which has a relatively small number of code lines and simple functionality. Moreover, the number of used components in this family is restricted to two activities with small number of views, and two services, in most cases. Hence, it is slightly easy to obtain good code coverage for this family. On the other hand, Svpeng family registers the lowest code coverage. The selected samples from this family has large number of activities, complex functionalities, and take advantage of guards which directly affect the obtained code coverage.

As can be seen from Table 3, the top detected malicious behavior is Device Info Stealing, which can be considered as a very common behavior among all Android malware samples. It should be noted that using MEGDroid we could trigger malicious payload (or part of it) in 181 samples from the 200 samples used in the experiment which shows the effectiveness of MEGDroid to trigger malicious payloads in the code.

**RQ2) Efficiency of the proposed approach**

Each generated event in dynamic malware analysis starts a new potential path of analysis in the malware code that in turn imposes its corresponding amount of time and effort in the analysis phase. Moreover, especially for the case of modern malware, they almost try to outsmart the analyzer process, i.e. they try to detect the existence of automated event generation process to hide their malicious behavior accordingly [46]. Using high frequently generated events is a signature for malware to help it outsmarting the dynamic malware analyzer. Hence, providing a human impression in the analysis process through generating as lower number of events as possible, while reaching as higher code coverage as possible is an efficiency goal in the event generation process. One of our design objectives in MEGDroid has been to reach this goal and hence to evaluate our success, the CCP metric was already defined.

Fig. 7 shows the value of CCP obtained by MEGDroid to run the families illustrated in Table 2. As can be seen from Fig. 7, the CCP varied between 0.052 (for Svpeng) and 0.99 (for
FakeTimer) which shows the efficiency of MEGDroid. The number of generated events by MEGDroid for all samples is varied between 65 to 624 events including both UI events and system events. This number depends directly on the detected activities in the code. For example, in the case of FakeTimer family, there is only two activities detected with a little number of views in most samples. Hence, the number of required UI events is relatively small. On the other hand, in case of Svpeng family, the number of detected activities is very high (about 10 activities in most samples) which requires a large number of events. Moreover, Svpeng family has a complex functionality and uses many guards that MEGDroid (or the used Xposed module) could not handle. As the result, the obtained code coverage from this family is very low, and hence the value of CCP is low.

Since MEGDroid provides the facilities for analyzer user involvement, we recorded the time needed by the analyzer to adjust EPM. Moreover, since the EPM is not built from scratch, we used some software engineering students in our department who had been roughly familiar with Android apps development for this purpose. By observing the results, the average time used for modifying EPM by the analyzer was about 2 minutes, which can be considered relatively small. The most adjusted operations were related to adding UI events that is due to the higher knowledge of the analyzer users about the defined views in each activity in the sample under test. As a result, it can be said that MEGDroid provides an easy way to efficiently utilize the analyzer knowledge in the event generation process.

RQ3) How does MEGDroid compares with other Android event generator tools?

We compare MEGDroid with two state of the art event generators: Monkey and DroidBot. Monkey is a general-purpose event generator that provides UI events in a random manner, while DroidBot is a model-based event generator that provides UI events for Android malware analysis. Since these tools generate UI events without system events, for the sake of fairness in our evaluation, we equipped them with the system events mentioned in Dynalog [4], and compare them with MEGDroid. The reason of choosing Dynalog is that both Monkey and DroidBot have been used as UI event generators in this framework as illustrated in [27]. Moreover, we wrote Xposed module to dynamically handle anti-emulation techniques that are used by malware to hide the payloads, and use it for the three tools.

Comparing the code coverage

Fig. 8 represents the code coverage that is obtained from the aforementioned tools respectively. MEGDroid brought in better coverage than other tools in most cases. However, in some cases such as Dowgin, Monkey brought good coverage like MEGDroid and better than DroidBot. This is because Monkey generates more events compared to MEGDroid and DroidBot (we will discuss that in the next subsection), and the needed events are simples such as screen touches and buttons presses. In some families such as BankBot and Bankun, MEGDroid brought better coverage than both Monkey and DroidBot, because these families need complex UI events (such as insert bank information) that both tools cannot provide. On the other hand, in families such as Cova and Fusob, Monkey and DroidBot brought coverage almost equal to MEGDroid. This is because the required UI events are simple and can be achieved by both Monkey and DroidBot. In some families such as Svpeng and Kuguo, the code coverage is low because of existing some malware samples, in these two families, that use outsmarting techniques for protecting the malicious payloads and hence prevents executing the code. Note that these techniques could not be handled by the generated events or the used Xposed module. Timing techniques that delay executing the malicious payload, and waiting to a specific message from the attacker are two examples for the outsmarting techniques used by malware samples. Finally, we noticed that DroidBot failed in launching the app in many samples where it was unable to fetch the package of these samples to be launched, and it just generated random events without launching the sample under test.
Comparing the Event Generation Performance

From the previous subsection, we noticed that in some cases Monkey brought a very good code coverage despite its generated events are simple and not related. This is because Monkey generates more events compared to MEGDroid and DroidBot.

In our experiment, we ran Monkey 10 times with different number of events in each time, and we registered the number of events that obtained the highest coverage. Despite the high code coverage obtained by Monkey, we can say that MEGDroid and DroidBot provide better performance. The reason is that Monkey generates events randomly, while MEGDroid and DroidBot generate relevant events. However, to measure the performance of the tools we used CCP metric to measure the efficiency of the event generation tools.

Fig. 9 shows the derived value of the CCP of all three tools. As can be seen, MEGDroid provides better CCP compared to both Monkey and DroidBot. The main reason behind that is the advantage of using MDE approach. Making use of MDE approach in MEGDroid provides the ability to the analyzer to involve and use his/ her knowledge to adjust the generated events (especially UI events) and make them almost identical to the events that can be generated manually (i.e., by the user) but with less time and efforts. Moreover, in some families such as Cova, and Mecor, DroidBot brought almost the same CCP as obtained by MEGDroid. This is because DroidBot generates UI-guided events based on the transition model to simulate the user interaction as much as possible. Moreover, the required UI events are simple and can be easily simulated in this case. In the case of FakeTimer, Monkey brought CCP equals to DroidBot. This is because in this family DroidBot failed to launch most of the samples and generated random events that causes obtaining a very small code coverage. Finally, we noticed that CCP obtained by Monkey is very low because of the random based event generation technique that Monkey uses. In other words, there is no relation between the generated events in Monkey, and hence it requires generating a higher number of events to obtain higher code coverage.

Triggering more malicious payloads

Regarding the evaluation metrics, an important metric for assessing the effectiveness of event generators for dynamic...
malware analysis is their ability to trigger the malicious payloads in the code which can be measured by the number of sensitive APIs that reflect the malicious behavior. In this experiment, we ran the samples illustrated in Table 2, with all of the three tools and captured the sensitive APIs that were called by these tools. Note that the captured sensitive APIs provide an evidence on triggering the payloads in the code or part of it. Table 4 represents the top 10 sensitive API calls that were logged from the families represented in Table 2 using MEGDroid compared to the same sensitive API calls that were logged when Monkey and DroidBot were used respectively. While, Table 5, represents the top 10 API calls that were logged from the aforementioned families using DroidBot compared to the same sensitive API calls that were logged when Monkey and MEGDroid were used respectively. Finally, Table 6 represents the top 10 API calls that were logged from the aforementioned families using Monkey compared to the same sensitive API calls that were logged when DroidBot and MEGDroid were used respectively.

Table 4. Top 10 API calls logged from malware families using MEGDroid compared to Monkey and DroidBot

<table>
<thead>
<tr>
<th>Sensitive API method</th>
<th>MEGDroid</th>
<th>Monkey</th>
<th>DroidBot</th>
</tr>
</thead>
<tbody>
<tr>
<td>android.os.SystemProperties/get</td>
<td>158</td>
<td>35</td>
<td>74</td>
</tr>
<tr>
<td>java.net.URL/openConnection</td>
<td>121</td>
<td>25</td>
<td>53</td>
</tr>
<tr>
<td>android.telephony.TelephonyManager/getDeviceId</td>
<td>88</td>
<td>24</td>
<td>46</td>
</tr>
<tr>
<td>android.telephony.TelephonyManager/getLine1Number</td>
<td>66</td>
<td>12</td>
<td>35</td>
</tr>
<tr>
<td>dalvik.system.DexFile/openDexFile</td>
<td>61</td>
<td>11</td>
<td>40</td>
</tr>
<tr>
<td>dalvik.system.BaseDexClassLoader/findLibrary</td>
<td>59</td>
<td>9</td>
<td>37</td>
</tr>
<tr>
<td>android.telephony.TelephonyManager/getNetworkOperator</td>
<td>53</td>
<td>14</td>
<td>39</td>
</tr>
<tr>
<td>android.util.Base64/decode</td>
<td>47</td>
<td>6</td>
<td>22</td>
</tr>
<tr>
<td>android.provider.Settings.Secure/getString</td>
<td>44</td>
<td>8</td>
<td>40</td>
</tr>
<tr>
<td>android.telephony.TelephonyManager/getSubscriberId</td>
<td>41</td>
<td>10</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 5. Top 10 API calls logged from malware families using DroidBot compared to Monkey and MEGDroid

<table>
<thead>
<tr>
<th>Sensitive API method</th>
<th>DroidBot</th>
<th>Monkey</th>
<th>MEGDroid</th>
</tr>
</thead>
<tbody>
<tr>
<td>android.os.SystemProperties/get</td>
<td>74</td>
<td>35</td>
<td>158</td>
</tr>
<tr>
<td>java.net.URL/openConnection</td>
<td>53</td>
<td>25</td>
<td>121</td>
</tr>
<tr>
<td>android.telephony.TelephonyManager/getDeviceId</td>
<td>46</td>
<td>24</td>
<td>88</td>
</tr>
<tr>
<td>dalvik.system.DexFile/openDexFile</td>
<td>40</td>
<td>11</td>
<td>61</td>
</tr>
<tr>
<td>android.provider.Settings.Secure/getString</td>
<td>40</td>
<td>8</td>
<td>44</td>
</tr>
<tr>
<td>android.telephony.TelephonyManager/getNetworkOperator</td>
<td>39</td>
<td>14</td>
<td>53</td>
</tr>
<tr>
<td>dalvik.system.BaseDexClassLoader/findLibrary</td>
<td>37</td>
<td>9</td>
<td>59</td>
</tr>
<tr>
<td>android.telephony.TelephonyManager/getLine1Number</td>
<td>35</td>
<td>12</td>
<td>66</td>
</tr>
<tr>
<td>android.webkit.WebView/loadUrl</td>
<td>34</td>
<td>4</td>
<td>35</td>
</tr>
</tbody>
</table>

Table 6. Top 10 API calls logged from malware families using Monkey compared to DroidBot and MEGDroid

<table>
<thead>
<tr>
<th>Sensitive API method</th>
<th>Monkey</th>
<th>DroidBot</th>
<th>MEGDroid</th>
</tr>
</thead>
<tbody>
<tr>
<td>android.os.SystemProperties/get</td>
<td>35</td>
<td>74</td>
<td>158</td>
</tr>
<tr>
<td>java.net.URL/openConnection</td>
<td>25</td>
<td>53</td>
<td>121</td>
</tr>
<tr>
<td>android.telephony.TelephonyManager/getDeviceId</td>
<td>24</td>
<td>46</td>
<td>88</td>
</tr>
<tr>
<td>android.telephony.TelephonyManager/getNetworkOperator</td>
<td>14</td>
<td>39</td>
<td>53</td>
</tr>
<tr>
<td>java.net.ProxySelectorImpl/select</td>
<td>13</td>
<td>17</td>
<td>40</td>
</tr>
<tr>
<td>android.telephony.TelephonyManager/getLine1Number</td>
<td>12</td>
<td>35</td>
<td>66</td>
</tr>
<tr>
<td>dalvik.system.DexFile/openDexFile</td>
<td>11</td>
<td>40</td>
<td>61</td>
</tr>
<tr>
<td>android.telephony.TelephonyManager/getSubscriberId</td>
<td>10</td>
<td>24</td>
<td>41</td>
</tr>
<tr>
<td>dalvik.system.BaseDexClassLoader/findLibrary</td>
<td>9</td>
<td>37</td>
<td>59</td>
</tr>
<tr>
<td>android.telephony.TelephonyManager/getNetworkOperatorName</td>
<td>7</td>
<td>21</td>
<td>32</td>
</tr>
</tbody>
</table>

As can be seen from these 3 tables, more features can be extracted from MEGDroid compared to Monkey and DroidBot. The main reason behind these results is that MEGDroid provides more complex interactions because it uses analyzer knowledge to provide appropriate UI and system events. Moreover, by comparing the results represented in the three aforementioned tables, we can notice that DroidBot brought better results compared to Monkey, and this shows the impact of generating related events on triggering malicious payload. The number of extracted features using Monkey is low compared to both MEGDroid and Monkey. This approves our claim about Monkey that this tool is not appropriate for malware dynamic analysis because it does not generate related UI events. Finally, it is worth mentioning that the API method android.os.SystemProperties/get was the most called method by all three tools. It was called in 158 samples when using MEGDroid, in 74 samples when using DroidBot, and in 35 samples when using Monkey. Note that most of the sensitive APIs had been called more than one time in a single sample, but we considered each call once for each sample.

Table 7, represents a comparison among the three tools to trigger the most common malicious behaviors for the selected families. Note that to compute the result for each behavior, we use the number of sensitive API methods related to each behavior as illustrated in the discussion of the first research question section.
Table 7. Most common malicious behaviors: Monkey vs DroidBot vs MEGDroid

<table>
<thead>
<tr>
<th>Malicious behavior</th>
<th>Monkey</th>
<th>DroidBot</th>
<th>MEGDroid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Info Stealing</td>
<td>126</td>
<td>319</td>
<td>568</td>
</tr>
<tr>
<td>Personal Info Stealing</td>
<td>13</td>
<td>27</td>
<td>62</td>
</tr>
<tr>
<td>Communicate with C&amp;C Via SMS</td>
<td>3</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Communicate with C&amp;C Via Internet</td>
<td>51</td>
<td>172</td>
<td>295</td>
</tr>
<tr>
<td>Evade Analysis via Encryption</td>
<td>33</td>
<td>93</td>
<td>146</td>
</tr>
<tr>
<td>Evade Analysis via Dynamic Load</td>
<td>34</td>
<td>131</td>
<td>181</td>
</tr>
</tbody>
</table>

As can be seen from Table 7, the most malicious behavior that we got from the aforementioned malware families, is the Device Info Stealing behavior by the three tools, which is largely used by Android malware to collect information about the device in which the sample is running. Moreover, we can notice that MEGDroid obtained the best result, then DroidBot, and after that, Monkey. Therefore, we can say that DroidBot can be more appropriate than Monkey to trigger the malicious behavior from the code, but it suffers from some limitations in case of complex interactions that we greatly enhance in MEGDroid.

**RQ4) What is the impact of analyzer involvement in the event generation process**

Since MEGDroid, is a model-based event generator, the completeness and the correctness of the model directly affects the generated events. In other words, if the model is incomplete or incorrect, the generated events may not be complete/correct. Hence, we take advantage of the analyzer knowledge and experience to deal with this problem. In this experiment we answer this question and measure the impact of the analyzer user on the effectiveness of the events generation process.

Because EPM (which is used to generate actual events) is not built from scratch manually, and the analyzer involvement is restricted to modify EPM by adding, deleting, or reordering the elements in EPM according to his/her knowledge, the time consumed by the analyzer to modify the EPM is relatively small. However, to measure the impact of analyzer involvement in the event generation process, we ran MEGDroid with and without analyzer involvement and compare the results regarding code coverage, event generation performance, and number of called sensitive APIs which reflect the effectiveness and the efficiency of the analyzer involvement in the event generation process. Table 8 represents the families (with 10 samples for each family) that are used in this experiment.

Table 8. The families of malware that are used in the analyzer user involvement experiment

<table>
<thead>
<tr>
<th>Family</th>
<th>1) GoldDream</th>
<th>4) FakeTimer</th>
<th>7) Bankun</th>
</tr>
</thead>
<tbody>
<tr>
<td>2) Kuguo</td>
<td>5) Koler</td>
<td>8) Svpeng</td>
<td></td>
</tr>
<tr>
<td>3) Fusob</td>
<td>6) Utchi</td>
<td>9) SlemBunk</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 10 represents the code coverage obtained from the families represented in Table 8 in two scenarios i.e., with and without analyzer involvement. As can be seen from Fig. 10, in some situations the analyzer involvement can deeply affect the obtained code coverage as in Koler and Bankun families. After analyzing these families, we noticed that, in the Koler family, there were some GUI elements that were dynamically constructed. Hence, since MEGDroid uses static analysis to extract event’s sources from the code, it could not automatically generate events for such GUI elements, and we use the analyzer knowledge to add events for these GUI elements. For the case of Bankun families, the samples include fake bank apps that require many meaningful information such as bank accounts, and MEGDroid could not provide this information without the analyzer involvement. Finally, in some cases, such as Fusob and Svpeng, the analyzer involvement relatively affected the obtained code coverage. This is because MEGDroid could automatically detect the sources of the events and generates appropriate events as a response to these sources without the need to a heavy analyzer involvement in the case of Fusob family. While in the case of Svpeng family, the obtained code coverage in both scenarios were small because of the aforementioned outsmarting techniques.

Table 9 shows the malicious behaviors detected when the families in Table 8 are executed with and without the analyzer involvement. The results show that the knowledge of the
analyzer about the samples under test considerably affects the number of triggered sensitive APIs (that reflects the malicious behavior of the samples). This reflects the effectiveness of the analyzer knowledge involvement in the event generation process especially in case of malware analysis.

Table 9. Most common malicious behavior: with and without the analyzer user involvement

<table>
<thead>
<tr>
<th>Malicious behavior</th>
<th>Without analyzer involvement</th>
<th>With analyzer involvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Info Stealing</td>
<td>124</td>
<td>283</td>
</tr>
<tr>
<td>Personal Info Stealing</td>
<td>7</td>
<td>42</td>
</tr>
<tr>
<td>Communicate with C&amp;C Via SMS</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Communicate with C&amp;C Via Internet</td>
<td>52</td>
<td>136</td>
</tr>
<tr>
<td>Evade Analysis via Encryption</td>
<td>31</td>
<td>98</td>
</tr>
<tr>
<td>Evade Analysis via Dynamic Load</td>
<td>53</td>
<td>106</td>
</tr>
</tbody>
</table>

In order to show the efficiency of the analyzer involvement in the generation process we used the CCP metric, and ran the samples of the families that have been illustrated in Table 8 using MEGDroid in two scenarios, i.e. with and without analyzer involvement. Fig. 11 shows the resulted CCP in both scenarios. As can be seen from Fig. 11, in most families such as Koler, analyzer involvement can have a slightly good effect on the generated events performance. However, in some families such as Fusob and FakeTimer, the CCP computed when the analyzer was not involved is better than the case of analyzer involvement. By analyzing these families, we found that in the case of Fusob family, when the analyzer involved, the average number of generated events was 135 and the achieved coverage was 44.48%, while without analyzer involvement the average number was 125, and the achieved coverage was 41.85%. Hence the added events improve the achieved coverage which is more important. This situation applies to FakeTimer family. Finally, it should be noted that in some samples, the analyzer involvement includes reordering some elements in EPM, which did not affect the number of the generated events.

5.4. Threats to validity

Since Monkey generates random events, there is no guaranty that we got its best results. However, we ran Monkey 10 times, with different numbers of events each time and we selected the best result from the 10 results that we got. Moreover, our experiment on analyzing the impacts of analyzer involvement is somehow subjective. This means that the experiences of the analyzer and his/her familiarity with the sample under test can strongly affect the results. However, we tried to use students with moderate or low expertise and familiarities with the subjects to do the job. However, if MEGDroid is tended to be used by domain experts such as expert Android programmers or analyzers in a malware analysis lab, for example, the achieved results can be considered as an underestimation. Finally, we tried to capture the sensitive APIs that reflect the malicious behavior of the sample under test but there is a chance that the captured sensitive APIs are not used in the payload, for example in the case of Cova family we captured the dalvik.system.DexFile/openDexFile API which means that there is a dynamic load behavior, but according to the description that we got from AMD dataset, Cova family does not have dynamic load as malicious behavior. This implies having false positive error in triggering the malicious payloads.

6. Limitations

Since MEGDroid depends on static analysis for extracting information from the code to generate appropriate events, this imposes some limitations as the following:

1- We may not be able to extract any information if the sample under test is highly and completely obfuscated or encrypted. In this case, we completely depend on the analyzer knowledge about the sample under test to almost generate appropriate events for this sample. However, the analyzer knowledge/information still may not be enough especially when the samples have many complex activities. As our future work, we aim to use dynamic analysis (in addition to static analysis) to extract more information from the code.

2- Using static analysis prevents MEGDroid from monitoring the reaction of the sent events. As an example, consider a case that MEGDroid sends two events where the first one is for filling a text field and
the second one is for clicking a button after filling up the text. If the first event for any reason failed to fill the text field, MEGDroid will not be able to decide filling the text field again. For this purpose, we aim to extend MEGDroid to use dynamic analysis to monitor the sent events during the execution time.

Finally, since the Android malware can be triggered by one or more of the three inputs: environment conditions, victim activities, or attacker activities. The current version of MEGDroid handles the first and second inputs. i.e., it can generate UI and system events without considering any other inputs from the attacker. For example, MEGDroid can generate SMS receive events, but without filling up the received SMS with appropriate content that may be treated as instructions to trigger the payload. We aim to extend the proposed meta-models to handle such inputs as future work.

7. Conclusion

In this paper, we introduced MEGDroid, as a novel model-driven framework, to generate events for Android malware dynamic analysis. This framework includes two meta-models, one for defining the sources of the events, and the other for defining the generated events. MEGDroid provides the ability to generate both system and UI events in addition to the analyzer involvement option for modifying the generated events to become more realistic and human-like. The proposed framework has been realized as a tool which is implemented as an Eclipse plugin. In order to evaluate the proposed approach and tool, we used 200 samples selected form 20 different families in AMD dataset. The results of the evaluation showed that MEGDroid provides better coverage despite the low number of generated events. Moreover, it provides a remarkable enhancement in the case of logged sensitive API methods called by the malware. Moreover, it has succeeded to run some specific cases that other tools failed to run.

This work can be improved in several directions. For instance, generating UI and system events for triggering malicious parts of the code are not sometimes sufficient because of other activities (such as attacker activities) that can be used to protect malicious parts of the code. Hence, the proposed approach can be extended to handle this challenge as future work. Moreover, the framework can be extended with dynamic analysis for both extracting adequate information and monitoring the sent events which can reduce the efforts and time required by the analyzer to produce appropriate events and could considerably enhance the strength of the tool. Finally, since MEGDroid is an event generation tool, we can tailor it to be used as a security fuzzer, i.e., to generate test inputs for security purposes such as finding vulnerabilities in Android apps.

References


