Towards Empirically Assessing Behavior Stimulation Approaches for Android Malware

Aleieldin Salem, Michael Hesse, Jona Neumeier, and Alexander Pretschner

Technische Universität München
Garching bei München, Germany
Email: {salem, hessem, neumeiej, pretschn}@in.tum.de

***Abstract*—Android malware authors have increasingly relied on techniques to hinder dynamic analysis of their apps by hiding their malicious payloads or by scheduling their execution based on complex conditions. Consequently, researchers devise different approaches to bypass such conditions and stimulate the malicious behaviors embedded within the Android malware. Despite the availability of different behavior stimulation approaches and dynamic analysis tools that implement them, they are seldom empirically evaluated to assess their applicability and effectiveness. In this paper, we survey the literature to identify different behavior stimulation approaches and assess the performance of three tools implementing them against four datasets of synthetic and real-world malware. Using the obtained results, we highlight significant limitations of such analysis tools, including their instability and their inability to stimulate scheduled behaviors even in automatically generated synthetic malware. Those limitations enable simple approaches based on the random manipulation of an app's User Interface (UI) to outperform more sophisticated behavior stimulation approaches. We aspire that our results instigate the adoption of more rigorous evaluation methods that ensure the stability of newly-devised analysis tools across different platforms and their effectiveness against real-world Android malware.**

Keywords–Android Security; Application Analysis; Malware Detection.

I. INTRODUCTION

Android malware authors utilize different techniques to hinder static analysis of their apps, such as anti-debugging techniques [1], code obfuscation and encryption [2], dynamic code loading [3], and triggering and scheduling [4] [5]. However, more recently, malware authors have increasingly relied on evasion techniques to hinder dynamic analysis as well [1]. For example, Wei et al. found that the majority of malicious apps they gathered and analyzed utilized *schedulers* to delay the execution of their payloads [5]. Consequently, researchers have devised different approaches to identify suspicious segments within Android apps and stimulate (i.e., execute) them, such as in [6]–[10]. We refer to these approaches as *behavior stimulation* approaches. Stimulating suspicious behaviors helps detection methods understand the true intentions of an app, and classify it correctly as malicious or benign [11].

In describing their stimulation approaches, and the dynamic analysis tools that implement them, researchers tend to focus on distancing their work from the previous work by, for example, enumerating the new features offered by their tools or the limitations of prior work their new tools tackle. Furthermore, approaches are usually evaluated using either a few samples [7] [9] [10] or mainly using synthetic malware datasets [6], to allow for a more in-depth description of the concepts upon which the approaches are built.

Unfortunately, this renders it difficult for other researchers to assess the applicability and effectiveness of the current behavior stimulation approaches against Android malware found in the wild. In particular, the answers to the following questions are unknown to researchers: (**Q1**) How well can the current behavior stimulation tools stimulate scheduled malicious behaviors embedded in (synthetic) Android malware? (**Q2**) How difficult is it to trigger malicious behaviors dwelling in Android malware found in the wild (e.g., app marketplaces)? And (**Q3**) What are the limitations that face some of the current analysis tools and their behavior stimulation approaches?

To answer such questions, we conducted preliminary experiments performed on (1) synthetic malicious apps that implement schedulers, and (2) random samples drawn from real-world Android malware datasets (i.e., *Piggybacking* [12] and *AMD* [5]). In those experiments, we compared the performance of three tools that represent different behavior stimulation approaches according to three criteria, viz. the time taken to analyze an app, whether a tool undermines the stability of an app (e.g., causes it to crash), and whether a tool managed to unveil the malicious code segments embedded within an app.

The results we obtained from our preliminary experiments suggest the following. Firstly, we found that a noticeable number of the behavior stimulation tools we surveyed are (a) poorly documented, which makes it difficult to set them up and configure them, and (b) did not deliver the functionalities described in their papers. Secondly, none of the tools we used during our experiments managed to stimulate malicious behaviors guarded by primitive schedulers (i.e., time-based triggers), embedded in synthetic malware generated by the repackaging tool, `Repackman` [4]. Thirdly, our results imply that Android malware authors implement their instances in a manner that exhibits some of their malicious behaviors without schedulers, especially malicious apps belonging to the less subtle types of `Adware` and `Riskware`. This enables primitive tools (e.g., the ones that interact with an app's UI), to outperform more sophisticated counterparts in terms of the aforementioned three criteria.

In summary, our contributions are:

- We survey the literature to identify the existing and available Android malware dynamic analysis tools that implement behavior stimulation approaches and categorize them according to such approaches (Section II).
- The conducted experiments revealed that the tools used during evaluation–including those mainly designed to bypass schedulers–were unable to stimulate malicious behaviors protected by simple time-based

 ISBN: 978-1-61208-746-7

triggers. We found that maintaining the stability of test apps is more important than the complexity of a tool's stimulation approach as it increases its chances to stimulate malicious behaviors in malware (Sections III and IV).

- We share with the research community the results of our experiments to verify, reproduce, and improve upon our findings.

II. Stimulation Approaches

We surveyed the literature in pursuit of dynamic analysis tools that (a) implement any behavior stimulation approach, and (b) are designed to analyze Android malware or, at least, accommodate for malicious apps. Tools such as `TriggerScope` [13], for instance, which identifies logic-based triggers embedded within an Android app, but does not include modules to stimulate behaviors within the app were ruled out. Using such criteria, we so far managed to identify 11 dynamic analysis tools with behavior stimulation modules, as seen in Table I. Those tools implemented three approaches to behavior stimulation that we refer to as *random UI manipulation*, *forcing execution*, and *environment adaptation*.

In addition to investigating the stimulation approach adopted by different analysis tools, we studied the techniques they use to target suspicious code segments within an app (*Code Targeting*), the mechanisms they utilize to execute the identified targets (*Code Triggering*), whether they require any modifications to the apps under test or the systems on which they are analyzed (*Invasiveness*), the programming language level they operate on (*Operation Level*), and whether they provide any documentation or source code to the research community (*Availability+Maintainability*). In the following sections, we briefly explain those strategies and techniques and how different tools utilize them.

A. Random UI Manipulation

The random manipulation of an app's user interface is the most primitive of stimulation approaches. Tools adopting this approach usually do not implement any strategies to target code segments within apps. Instead, the majority of such tools randomly interact with the graphical user interface elements of apps (e.g., `Button` or `TextField`), and their background components (e.g., `Service` or `BroadcastReceiver`), to instigate (suspicious) runtime behaviors. Consequently, random UI manipulation tools usually do not implement any automatic strategies to trigger specific segments of code. Such responsibility is delegated to the user in the form of scripts that define sequences of interactions with app components (e.g., start `Activity` A, then tap `Button` B, then broadcast `Intent` I) [16] [17].

The lack of code targeting and triggering strategies implies that random UI manipulation tools are, by and large, non-invasive. That is, apart from injecting logging statements into an app, such tools do not require any modifications to the app under test or the test environment to function. Furthermore, random UI manipulation tools tend to solely operate on apps' (graphical) components, without the need to explore or analyze the apps' codebases.

B. Forcing Execution

In Android apps, some code segments are implemented to execute only if some conditions are satisfied. For example, updates usually require devices to be connected to the internet via WiFi and the devices to be plugged in for charging. Forcing execution tools are designed to bypass any conditions that prevent the code of interest from executing, effectively forcing it to execute. We identified two main methods to force the execution of code segments. The tools `GroddDroid` [10] and `Harvester` [18] *replace* any conditional statements leading to the target code with unconditional ones, whereas `Droid-AntiRM` [19] and `ARES` [6] *alter* the boolean expressions of such conditional statements to values that lead to the execution of the target code. The majority of forcing execution tools maintain lists that define the Android Application Programming Interface (API) calls they should pursue in an app's code and attempt to execute. The API methods in those lists are known to be often utilized by malware (e.g., `sentTextMessage`) [7] [10].

Forcing execution tools usually operate on a low-level representation of an app's code (e.g., `DEX` bytecode), and utilize different techniques including slicing [18] and control-/data-flow analysis [10] [19], to find paths between entry points in the code (e.g., the app's main activity), and the target API calls. Modifying the apps' code implies app-level invasiveness. Moreover, some tools, such as `ARES`, require the modification of the test environment as well to generate trace logs that might reveal previously unforeseen execution paths.

After modification, the paths to target code should be unobstructed with any (boolean) conditions, and the target code should execute after simple, random interaction of the apps UI (e.g., using tools like `Monkey`). Some forcing execution tools embed a controller activity into the modified app, which calls the functions along the path leading to the target code [18].

C. Environment Adaptation

Environment adaptation attempts to trigger the target code without modifying an app's control flow or structure. To do that, tools adopting this approach alter the environment surrounding the app to steer its execution towards the targeted code. For example, if the target code needs access to the device's Global Positioning System (GPS) module to execute, the analysis tool would switch on the location services and grant the app any necessary permissions.

Environment adaption analysis tools rely on user-defined target code usually in the format of API calls to identify target code. Once identified, a path from an app's entry point to the target code is calculated, and variables along this path are included in constraints generated using symbolic execution [7]–[9]. The symbolic variables in such constraints depict values read from files, statuses returned after querying system modules, or variables inside the app's code.

During runtime, environment adaptation tools make sure that the symbolic variables in those constraints have values that steer an app towards the target code. Instead of altering the environment itself to influence the symbolic variables, current environment adaptation tools intercept the values returned from the system or queried resource, and replace them with the values required to execute the target code.

 ISBN: 978-1-61208-746-7

TABLE I. A SUMMARY OF THE IDENTIFIED ANDROID MALWARE DYNAMIC ANALYSIS TOOLS THAT IMPLEMENT A BEHAVIOR STIMULATION APPROACH. THE TOOLS WE UTILIZED DURING OUR EXPERIMENTS ARE HIGHLIGHTED IN RED.

Tool	Stimluation Approach			Code Targeting			Code Triggering		Invasiveness			Operation Level			Availability+ Maintainability			
	Random UI	Force Execn	Env Adaptn	None	Auto	Manual	Random UI	Guided	No	App	System	User Interface	Native Code	DEX Bytecode	User Manual	Source Code	Executable(s)	Last Commit
SmartDroid [14]	✓			✓			✓		✓			✓						
CopperDroid [15]	✓				✓		✓	✓			✓		✓	✓				
Droidbot [16]	✓			✓			✓	✓	✓			✓			✓	✓	✓	Apr'19
Droidmate-2 [17]	✓			✓			✓	✓		✓		✓			✓	✓	✓	Jun'19
GroddDroid [10]		✓				✓	✓	✓		✓				✓	✓	✓	✓	
Harvester [18]		✓				✓		✓		✓				✓	✓			
Droid-AntiRM [19]		✓				✓	✓			✓				✓				
ARES [6]		✓			✓		✓			✓	✓			✓	✓	✓	✓	Apr'18
IntelliDroid [9]			✓			✓		✓			✓			✓	✓	✓	✓	Dec'16
FuzzDroid [7]			✓			✓		✓		✓		✓		✓		✓	✓	Feb'17
Malton [8]			✓		✓	✓		✓	✓				✓					

To calculate the aforementioned constraints, the analysis tools usually operate on a low-level representation of the apps' code, such as DEX bytecode [7] [9] or on native code executed on Android Runtime (ART) layer [8]. In terms of invasiveness, Malton does not modify the apps or their testing environments, IntelliDroid needs a modified version of the Android operating system to include a service that feeds the app during execution with the values required to satisfy the constraints, and FuzzDroid injects logging statements into the apps for a similar purpose along with tracking the execution paths.

III. EXPERIMENTS

a) ***Tools****:* Out of the 11 tools we identified so far, only six offered their source code or executables to the research community, which we attempted to install, configure, and test. Unfortunately, half of the remaining tools (i.e., three tools), either (a) were incomplete and needed a further extension before being used, or (b) did not deliver the functionality described in their respective papers. For example, the tool FuzzDroid needed to be extended to accommodate for different types of sensitive API calls apart from Short Message Service (SMS)-related ones. Furthermore, after obtaining the source code of the tool ARES, following the instructions available on its website to compile a customized version of the Android kernel, setting up and running it against the *EvaDroid* dataset, we could not reproduce the results reported in its paper [6], primarily because the tool did not manage to trigger the payloads in such apps. Consequently, we ran our experiments using the three remaining tools, namely Droidbot, GroddDroid, and IntelliDroid, which fortunately respectively represent the stimulation approaches of *random UI manipulation*, *forcing execution*, and *environment adaptation*.

b) ***Datasets****:* We ran the aforementioned three tools against four datasets of synthetic and real Android malicious apps. The first dataset we considered is *EvaDroid* [6], which comprises 24 manually-developed, synthetic malicious apps that implement different types of schedulers (e.g., time-based, virtualization fingerprinters, battery status checkers, etc.). The second dataset, referred to as *Repackman*, comprises 30 synthetic malicious apps automatically generated using the Repackman tool [4] by grafting trigger-protected malicious payloads into benign apps from the Google Play store. The third dataset is a random sample of 30 malicious apps drawn from the *Piggybacking* dataset, which includes real-world Android repackaged malware [12]. The distribution of malware types in this dataset is Adware (63%), Riskware (17%), Trojan (16.1%), and Spyware (3%). Lastly, we drew a random sample of 130 malicious apps out of 24,553 from the *AMD* dataset including different families and types of Android malware (e.g., Adware (57.7%), Trojan (27.78%), and Ransomware (8.74%)) [5].

A. Results

Table II summarizes the results obtained from our experiments. For each dataset, we calculated the **a**verage **t**ime taken (in seconds) by each tool to analyze an app (AT), the percentage of **a**pps that were successfully **a**nalyzed (AA), and whether the malicious **p**ayloads embedded within the apps were **t**riggered (PT) and found in the logs of the successfully analyzed apps. The time taken by Droidbot to analyze apps seems constant across different datasets because we allowed the tool to run for five minutes per app.

We defined an analysis to be successful if a tool that monitors the API calls issued by an app during runtime, Droidmon [20], managed to generate a log for an app. Droidmon monitors a defined list of API calls that (a) interact with sensitive system resources (e.g., user contacts), or (b) are known to be widely-adopted by malicious apps (e.g., sending and receiving SMS messages). We made sure to synchronize the lists of API calls targeted by GroddDroid and IntelliDroid and monitored by Droidmon itself. This means that stimulation tools, for example, GroddDroid, would attempt to target and execute the same API calls that are monitored and logged by Droidmon. Such synchronization is the only modification we made to the analysis tools. To automate the process of analysis, we wrote scripts that iterate over the apps in a dataset, launches the analysis tool, and downloads any logs generated by Droidmon from the virtual device on which the analysis was performed. We manually examined the generated Droidmon logs of each analyzed app to inspect whether any of its malicious payload(s) have exhib-

 ISBN: 978-1-61208-746-7

TABLE II. A SUMMARY OF THE RESULTS OBTAINED FROM OUR EXPERIMENTS.

Dataset	EvaDroid			Repackman			Piggybacking			AMD		
	AT	AA	PT	AT	AA	PT	AT	AA	PT	AT	AA	PT
Droidbot	305.6	100%	17%	304.45	100%	**0%**	305.89	96.67%	**86.20%**	306.34	76.15%	**54.54%**
GroddDroid	104.45	100%	37%	1434.2	40%	**0%**	2629.22	66.33%	68.42%	209.26	57.69%	46.67%
IntelliDroid	N/A	100%	33%	N/A	43%	**0%**	N/A	46.67%	71.42%	N/A	79.23%	42.67%

ited. For the synthetic malware datasets, the inspection was straightforward, especially since the malicious payloads were merely logging messages or `Toast` messages that indicate the execution of the targeted code (e.g., *Evil Payload Triggered!!*). We made sure that the analysis tools and `Droidmon` do indeed target and monitor such logging statements, respectively. As for apps in the *Piggybacking* and *AMD* datasets, we relied on the information available on `VirusTotal` [21] or provided by the dataset authors about the apps' behaviors.

IV. Discussion

In this section, we attempt to answer the research questions (**Q1**), (**Q2**), and (**Q3**) that we postulated in Section I.

Q1

How well can the current behavior stimulation tools stimulate scheduled malicious behaviors embedded in (synthetic) Android malware?

On the *EvaDroid* dataset, despite managing to outperform their simpler counterpart, `Droidbot`, the more sophisticated analysis tools `GroddDroid` and `IntelliDroid` performed mediocrely on such dataset, given the simplicity of its apps and the schedulers they utilize. Moreover, we noticed that some of the tools managed to analyze and trigger the payloads in apps that other tools did not manage to either successfully analyze or to reveal their payloads. For example, `GroddDroid` managed to uniquely trigger the payloads in the apps *accelH* and *network1*, whereas `IntelliDroid` triggered the payloads in *adbPortDetector*, *installedApps*, *qemuFingerprinting*, and *uptime*. Collectively, the three tools successfully analyzed and triggered the payloads in 13 (54%) out of 24 apps in the *EvaDroid* dataset, which continues to be a less than expected percentage.

The performance of all tools worsened on the *Repackman* dataset. While this can be expected from simple approaches as `Droidbot`'s, this result is indeed not expected from `GroddDroid` and `IntelliDroid`. One possible reason of failing to trigger such payloads is the inability of the `GroddDroid` and `IntelliDroid` to successfully analyze around 60% of the apps in the dataset, which might be a result of runtime errors rather than technical shortcomings. Upon futher investigation, we found that apps tested using `GroddDroid` did not generate any `Droidmon` logs as they encountered various types of runtime exceptions, mostly related to calling methods in classes that are yet to be loaded (e.g., `java.lang.NullPointerException` and `android.os.DeadObjectException`). A possible reason behind this behavior could be `GroddDroid`'s technique of skipping over specific code segments and conditions in order to execute the targeted code. As for `IntelliDroid`, unlike `GroddDroid`, we did not find any evidence of crashes in the system logs we downloaded from the virtual devices. In other words, failure to target and/or trigger any payloads in the apps might be due to some deficiencies in the tool's approach. We consider the failure of both tools to trigger such payloads as a significant source of concern, given the ability to generate hundreds of malicious apps using such automated method and the simplicity of the triggers injected into those apps.

The tools' performances on the *AMD* dataset are indeed more balanced, yet raise other concerns. The majority of malicious apps in this dataset makes use of schedulers that delay the execution of the apps' malicious payloads [5]. So, we expected `Droidbot` to be outperformed by the other tools in terms of triggering scheduled malicious payloads. However, as implied by the (**PT**), **Droidbot** slightly outperformed **GroddDroid** and `IntelliDroid`. Similar to the *EvaDroid* dataset, we found that the three tools complemented one another in terms of apps they uniquely managed to trigger their payloads. Between the three tools, the payloads of 122 (93.84%) out of 130 apps were successfully triggered. We investigated the apps that tools uniquely analyzed and triggered in pursuit of patterns that might indicate the strengths of some of the tools.

On the one hand, `Droidbot` managed to uniquely trigger the payloads in three apps that belong to different malware types (i.e., `Ransomware`, `Trojan`, and `Adware`). The other tools could not identify any code to target within such apps. On the other hand, `GroddDroid` and `IntelliDroid` managed to trigger malicious payloads in 19 apps that `Droidbot` did not manage to trigger. We found the payloads in ten of those apps were uniquely triggered by `IntelliDroid`, whereas six were uniquely triggered by `GroddDroid`.

In pursuit of any differences between the apps successfully analyzed by each tool, we consulted `VirusTotal` and retrieved the labels given by different scanners to each app, the last time an app was modified (i.e., roughly its development date), and the Android Software Development Kit (SDK) versions an app supports according to its `AndroidManifest.xml` file. Firstly, the VirusTotal labels did not reveal any patterns vis-à-vis the malware families or types that each tool excels at analyzing. In other words, we could not find any evidence that suggests, for example, that `GroddDroid` can uniquely trigger payloads embedded in `Trojans` or the `DroidKungFu` malware family. The tools also shared the average year in which a malicious app was last modified and presumably developed viz., 2013, and the average minimum SDK supported by the apps they uniquely triggered their payloads (i.e., API level 6).

The results we observed imply, we argue, that the success of a tool to trigger the payloads in any given malicious app hinges on the app itself (e.g., its functionalities, utilized permissions, used libraries, etc.). Furthermore, the

 ISBN: 978-1-61208-746-7

poor performance of older tools such as GroddDroid and IntelliDroid on apps in the *Repackman* dataset, which are newer than those in other datasets, implies that such tools do not generalize to newer apps with newer technologies (e.g., runtime permissions), or are meant to run on newer versions of Android. Consequently, as discussed earlier, the analysis of any given app should be carried out collectively using different analysis tools to increase the likelihood of successful analysis and payload triggering.

Q2

How difficult is it to trigger malicious behaviors dwelling in Android malware found in the wild (e.g., app marketplaces)?

To answer (**Q2**), we use the performance of Droidbot on the *Piggybacking* and *AMD* datasets as an indication of the difficulty to trigger the malicious payloads in an app. If Droidbot's simple random UI manipulation stimulation approach stimulates any malicious behaviors in an app, we infer that complex schedulers did not protect such malicious behaviors and, hence, were easy to stimulate. The decent performance of Droidbot's simple random UI manipulation implies that authors of Android malware in the *Piggybacking* dataset did not graft the legitimate apps they repackaged with sophisticated schedulers. We found that the majority of the malicious apps in the *Piggybacking* dataset comprise Adware, which usually focuses on the monetary gain rather than stealth and sophistication [12]. That is to say, the authors of Adware would rather trigger their malicious or *potentially unwanted* payloads as soon as possible to maximize their profit than hide the true intentions of their apps, especially since displaying more advertisements or implicitly rerouting their revenues does not bother device users or interrupt their usage as much as other more notorious breeds of malware (e.g., Ransomware). In this context, albeit unexpected, the performance of Droidbot is not necessarily surprising. Droidbot's performance is not replicated in case of the *AMD* dataset, which implies the use of more sophisticated schedulers and triggers. Nevertheless, given the simplicity of its approach, the tool managed to trigger the malicious payloads embedded within more than 50% of the dataset's apps. Similar to the *Piggybacking* dataset, the majority of malware types in *AMD* is Adware, which may have facilitated the tool's task.

So, we argue that using stimulation approaches as simple as random UI manipulation, a subset of the malicious behaviors found in some real-world Android malware types (e.g., Adware) or indications of their presence (e.g., fingerprinting the device), might be revealed. Otherwise, the existence of (complex) schedulers in apps noticeably hinders the performance of all tools.

Q3

What are the limitations that face some of the current analysis tools and their behavior stimulation approaches?

In addressing (**Q3**), we identified the following limitations facing the analysis tools we examined so far. Firstly, apart from Droidbot, the utilized tools were either too slow in targeting code and devising strategies to trigger it or required manual operation, such as IntelliDroid (hence the **N/A** in Table II). Secondly, the approaches adopted by GroddDroid and IntelliDroid, seemed to have undermined the stability of the apps, which hindered their successful analysis. Being a pre-requisite for payload triggering, we believe that this has negatively affected the ability of such tools to trigger payloads. Lastly, we noticed that all tools could only analyze apps compatible with the Android versions that the tools target. That is to say, the tools seem to be limited to the environments within which they were implemented and evaluated. With the lack of maintainability, the analysis tools we examined cease to cope with the frequently changing structures and behaviors of Android (malicious) apps, rendering them obsolete within a few years.

V. Limitations and Future Work

a) ***Dataset Size****:* As discussed in Section III, one of the main criteria in evaluating the performance of a behavior stimulation approach is its ability to trigger malicious behaviors embedded within the apps. To make sure that payloads have indeed triggered, especially in real-world malware, we manually inspected the Droidmon logs generated by the dynamic analysis tool. Using the entire corpus of the *Piggybacking* and *AMD* datasets and the logs generated from their apps by each of the **three** analysis tools means manually analyzing around 77,859 logs. So, we randomly selected apps from those datasets for our experiments, in order to have a sample that, we argue, represents the entire population of apps, their trends, triggers, and payloads.

b) ***Subset of Tools****:* The second limitation is the utilization of a subset of dynamic analysis tools we identified in Section II: out of 11 analysis tools, we ran our experiments using three tools. In order not to mistakenly claim that a particular tool is incapable of stimulating malicious behaviors during our preliminary experiments, we only considered tools that we could set up correctly, and that exhibit the behaviors described in their corresponding papers.

c) ***Future Work****:* Our future work is planned to address the aforementioned two limitations. Firstly, we wish to increase the sizes of the datasets we use during our experiments. The challenge, however, is to devise a method to semi-automatically investigate the logs generated for different apps, and decide upon whether the payloads residing in the apps have triggered. Secondly, we wish to continue surveying the literature for more dynamic analysis tools in pursuit of different approaches to behavior stimulation. Along with the tools we already identified, we wish to re-run our experiments on larger datasets of Android malware. As for tools we did not manage to set up or execute, we wish to investigate the reasons behind their unexpected behaviors after consulting their developers.

VI. Related Work

In [22], Sadeghi et al. perform a large scale study on the approaches and techniques used to assess the security of Android apps in general. Among the plethora of tools discussed in this study, dynamic analysis tools that implement behavior stimulation approaches (e.g., GroddDroid), are discussed. However, the study does not consider behavior stimulation as a technique to distinguish between tools and, hence, does not discuss it. Tam et al., in [1], do consider behavior stimulation approaches in surveying the literature for static and dynamic tools and discuss those specifically built to analyze Android

 ISBN: 978-1-61208-746-7

malware (e.g., `CopperDroid`). Richter [23] surveys different Android malware analysis tools and attempts to compare them concerning their weaknesses and limitations. In other words, without conducting any experiments, Richter studies the features and approaches offered and adopted by different tools (e.g., `Harvester`), and speculates the challenges that might face them. Lastly, Hoffmann et al. survey the literature for different analysis tools used to analyze Android (malicious) apps [2]. They focus, however, on the resilience of such analysis tools against obfuscation.

There are two main differences between our work and the aforementioned related work. Firstly, we focus on the approaches adopted by some dynamic analysis tools to increase the chance of stimulating malicious behaviors in Android apps. The majority of research efforts that survey dynamic analysis tools tend to ignore such approaches. Those that do consider it, such as [1], do not delve into comparing them. Secondly, to the best of our knowledge, there are no approaches that attempt to empirically compare the performance of different analysis tools against Android malware, even using small or sample datasets, which we do in this paper.

VII. Conclusion

Despite the existence of different behavior stimulation approaches, the research community does not possess any empirical studies on any scale that assess their applicability or effectiveness against (synthetic) Android malware. To address this gap, we surveyed the literature, identified three behavior stimulation approaches adopted by dynamic analysis tools, and assessed the performance of three tools representing them against four datasets of synthetic and real-world Android malware.

The results of our preliminary experiments suggest that, despite competing with more sophisticated behavior stimulation approaches, a simple approach based on the random manipulation of an app's UI can help reveal (subsets of) the malicious behaviors it contains. This proved to be the case with certain families of malware (e.g., `Adware` and `Ransomware`), whose authors usually do not implement complex schedulers to protect their malicious payloads. More importantly, our experiments revealed that all the utilized tools did not manage to trigger any time-scheduled payloads in synthetic malware generated by the automatic repackaging tool `Repackman`. Lastly, without maintaining a tool's code and adapting it to newer versions of Android, dynamic analysis tools seem to be suited to analyze a particular set of malicious apps viz., ones that were implemented for the same Android API version.

References

[1] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," ACM Computing Surveys (CSUR), vol. 49, no. 4, 2017, p. 76.

[2] J. Hoffmann, T. Rytilahti, D. Maiorca, M. Winandy, G. Giacinto, and T. Holz, "Evaluating analysis tools for android apps: Status quo and robustness against obfuscation," in Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. ACM, 2016, pp. 139–141.

[3] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci, "StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications," in Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, ser. CODASPY '15. ACM, 2015, pp. 37–48.

[4] A. Salem, F. F. Paulus, and A. Pretschner, "Repackman: A tool for automatic repackaging of android apps," in Proceedings of the 1st International Workshop on Advances in Mobile App Analysis. ACM, 2018, pp. 25–28.

[5] 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. Springer, 2017, pp. 252–276.

[6] L. Bello and M. Pistoia, "Ares: triggering payload of evasive android malware," in Proceedings of the 5th International Conference on Mobile Software Engineering and Systems. ACM, 2018, pp. 2–12.

[7] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel, "Making malory behave maliciously: Targeted fuzzing of android execution environments," in Proceedings of the 39th International Conference on Software Engineering. IEEE Press, 2017, pp. 300–311.

[8] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, "Malton: Towards on-device non-invasive mobile malware analysis for art," 2017, pp. 289–306.

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

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

[11] A. Salem, T. Schmidt, and A. Pretschner, "Idea: Automatic localization of malicious behaviors in android malware with hidden markov models," in International Symposium on Engineering Secure Software and Systems. Springer, 2018, pp. 108–115.

[12] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro, "Understanding android app piggybacking: A systematic study of malicious code grafting," IEEE Transactions on Information Forensics and Security, vol. 12, no. 6, 2017, pp. 1269–1284.

[13] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in Security and Privacy (SP), 2016 IEEE Symposium on. IEEE, 2016, pp. 377–396.

[14] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications," in Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices. ACM, 2012, pp. 93–104.

[15] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," in Proc. of the Symposium on Network and Distributed System Security (NDSS), 2015.

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

[17] N. P. Borges Jr, J. Hotzkow, and A. Zeller, "Droidmate-2: a platform for android test generation," in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, 2018, pp. 916–919.

[18] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques." in NDSS, 2016.

[19] X. Wang, S. Zhu, D. Zhou, and Y. Yang, "Droid-antirm: Taming control flow anti-analysis to support automated dynamic analysis of android malware," in Proceedings of the 33rd Annual Computer Security Applications Conference. ACM, 2017, pp. 350–361.

[20] Droidmon. Droidmon - dalvik monitoring framework for cuckoodroid. [Online]. Available: https://goo.gl/HPtdtG

[21] VirusTotal. Virustotal. [Online]. Available: https://goo.gl/s7GSrn

[22] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software," IEEE Transactions on Software Engineering, vol. 43, no. 6, 2017, pp. 492–530.

[23] L. Richter, "Common weaknesses of android malware analysis frameworks," Ayeks. de, 2015, pp. 1–10.

 ISBN: 978-1-61208-746-7