Evading Android Runtime Analysis Through Detecting Programmed Interactions

Wenrui Diao
The Chinese University of Hong Kong
dw013@ie.cuhk.edu.hk

Xiangyu Liu
The Chinese University of Hong Kong
lx012@ie.cuhk.edu.hk

Zhou Li
ACM Member
lzcarl@gmail.com

Kehuan Zhang
The Chinese University of Hong Kong
khzhang@ie.cuhk.edu.hk

ABSTRACT

Dynamic analysis technique has been widely used in Android malware detection. Previous works on evading dynamic analysis focus on discovering the fingerprints of emulators. However, such method has been challenged since the introduction of real devices in recent works. In this paper, we propose a new approach to evade automated runtime analysis through detecting programmed interactions. This approach, in essence, tries to tell the identity of the current app controller (human user or automated exploration tool), by finding intrinsic differences between human user and machine tester in interaction patterns. The effectiveness of our approach has been demonstrated through evaluation against 11 real-world online dynamic analysis services.

Keywords

Android malware; dynamic analysis; programmed interaction

1. INTRODUCTION

With the evolution of mobile computing technology, smartphone has experienced enormous growth in consumer market, among which Android devices have taken the lion's share. Unfortunately, Android's open ecosystem also turns itself into a playground for malware. According to a recent report [9], on average, 8,240 new Android malware samples were discovered in a single day.

To combat the massive volume of Android malware newly emerged, automated detection techniques (static and dynamic) were proposed and have become the mainstream solutions. Dynamic analysis frameworks monitor the behaviors of the app samples executed in a controlled environment under different stimuli. Compared with static analysis, dynamic analysis does not have to understand the complicated logic in malicious code and is immune to code obfuscation and packing. Moreover, less noticeable runtime malicious behaviors could be discovered.

The traditional dynamic analysis platforms were largely built upon emulators to enable fast and economic malware analysis.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

WiSec'16, July 18–20, 2016, Darmstadt, Germany.

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4270-4/16/07...$15.00

DOI: http://dx.doi.org/10.1145/2939918.2939926

To evade dynamic analysis, a broad spectrum of *anti-emulation* techniques have been proposed [21, 28, 14, 17] and adopted by malware authors. In general, these techniques were designed to fingerprint the runtime environment and look for artifacts that can tell physical device and emulator apart. Though effective at first, countermeasures have been developed by the security community to diminish the efficacy of anti-emulation. Recently, researchers proposed to use physical devices [19] and morph artifacts unique to emulators [12, 11, 13]. These methods wrecked the base of anti-emulation techniques, but we believe the arms race between dynamic analysis and evasion has not yet ended.

Automated Exploration. Different from the traditional desktop malware, Android malware are event-driven, meaning that malicious behaviors are usually triggered after certain combinations of user actions or system events. Therefore, the simple install-then-execute analysis model is not effective to trigger malware's runtime behaviors. To solve this issue, automated exploration techniques are integrated into dynamic analysis frameworks, including event injection, UI element identification, etc. The ultimate goal of them is to achieve good coverage of app's execution paths in a limited period.

New Evading Techniques. In this paper, we propose a new approach to evade Android automated runtime analysis through detecting programmed interactions. The core idea of this approach is to determine the identity of the party operating the app (a human user or an automated exploration tool) by monitoring the interaction patterns. To malware analysis, the goal of interaction is different from that of a real user. For efficiency, exploration tool injects simulated user events and avoids accessing the underlying devices. Such simulated events and hardware generated ones are inconsistent in most cases. Also, to achieve high coverage of execution paths, exploration tool tends to trigger all valid controls, among which some are not supposed to be triggered by human. We leverage these insights and built an evasive component `PIDetector`, which monitors the event stream and identifies the events unlikely coming from a real user. The malicious payload will be held from execution if a dynamic analyzer is identified.

Compared with the previous anti-emulation techniques, our approach exploits the gap between human and machine in runtime behaviors, instead of relying on features regarding execution environment. One prominent advantage of our approach is its robustness against any testing platform, even one composed of physical devices.

We implemented a proof-of-concept app and submitted it to 11 online dynamic analysis services screening samples submitted

from all sorts of sources. The preliminary results have already demonstrated the effectiveness of our approach: nearly all (available) surveyed services exhibit at least one pre-defined pattern of programmed interactions. As a recommendation, the design of the current dynamic analysis platforms should be revisited to defend against such new type of evasion.

Contributions. We summarize this paper's contributions as below:

- *New Technique and Attack Surface.* We propose a new approach to evade Android runtime analysis: programmed interaction detection, which provides a new venue for evading dynamic analysis other than existing anti-emulation works.

- *Implementation and Evaluation.* We implemented a proof-of-concept app and tested it on several real-world Android dynamic analysis platforms. The experimental results demonstrate our approach is highly effective.

2. RELATED WORK

Most Android dynamic analysis frameworks are built upon emulators [20], which is easier to be deployed and more economical, as the cost of purchasing mobile devices is exempted. Besides, the app behaviors on emulators are easier to be monitored and controlled. Such frameworks, however, are not robust against evasive malware, and anti-emulation techniques have been widely discussed. In this section, we review these techniques and describe the countermeasures proposed by security community.

2.1 Anti-Emulation

Nearly all previous anti-emulation techniques [21, 28, 14, 17] exploit the unique features of the virtualized environment and refrain from executing the core malicious payload (e.g., sending SMS to premium number) when the host is found as an emulator. The features that differentiate emulators from real mobile devices and are leveraged for anti-emulation are listed below:

Firmware Features. The mobile devices manufactured by vendors are assembled from distinctive firmware, which embeds unique ID or information reflecting the hardware specification. On the contrary, emulators tend to use fixed dummy values to fill firmware features. For example, `null` and `android-test` are fed to firmware-query APIs like `Build.SERIAL` and `Build.HOST` by emulators.

Device Features. A lot of peripheral devices, especially sensors, have been integrated into mobile devices, like accelerometer and gyroscope. Not all the sensors are supported by emulators, which can be exploited for emulator identification. For the sensors simulated by emulators, the data stream produced differs significantly (usually constant) from what is generated from real devices (randomly distributed) [28].

Performance Features. Performance, particularly processing speed, is a disadvantage for emulators. Though modern desktop PC has more processing power, such improvement is overwhelmed by penalty from instruction translation. As shown in [28], adversary could measure CPU and graphical performance, and then determine the existence of emulator.

It also turns out that there exists a huge number of heuristics can be employed for emulator detection. Jing et al. [14] proposed a framework which can automatically detect the discrepancies between Android emulators and real devices, and more than 10,000 heuristics have been discovered. Fixing these discrepancies on emulators needs tremendous efforts by all means.

2.2 Countermeasures

The anti-emulation techniques surveyed above are quite effective but not impeccable. They all look for observable artifacts produced from *virtualization*, which turns out to be the Achilles' heel. We describe two types of countermeasures for obscuring running platform below:

Using Physical Devices. Building analysis platform on physical devices could thwart anti-emulation behaviors naturally. Vidas et al. [29] proposed a hybrid system named A5, which combines both virtual and physical pools of Android devices. More recently, Mutti et al. [19] proposed BareDroid, which runs bare-metal analysis on Android apps. The system is built solely upon off-the-shelf Android devices and applies several novel techniques like fast restoration to reduce the performance cost. The evaluation results of these works prove that malware are not able to discern the analysis platform with users' devices.

Changing Artifacts. Another direction is to change the observable artifacts to masquerade the emulators as real devices. Hu et al. [13], Dietzel [11] and Gajrani et al. [12] followed this trail. They customized the emulator framework and hooked runtime APIs (in both Java and Linux layer) to feed fake values to the probing functions of malware. The malicious behaviors could be revealed when the checks for real devices are all passed.

3. BACKGROUND AND MOTIVATION

From the perspective of the adversary, pursuing the direction of fingerprinting execution environment would lead to a dead-end in the trend that more and more analysis platforms are driven by real devices or tailored emulators. In this work, we explore a new direction: instead of sensing *what* environment runs the app, we inspect the behaviors of dynamic analyzer and focus on *how* it interacts with the app. We first briefly overview the current dynamic analysis techniques and then introduce the concept of *programmed interaction* to motivate our research.

3.1 Dynamic Analysis

Different from static analysis tools, which scrutinize the source code or binary code of the program to identify the malicious payload, dynamic analysis frameworks execute the program to capture the malicious behaviors in the runtime. In particular, the execution environment for dynamic analysis is instrumented, and various system or user inputs (e.g., clicking UI buttons) are injected to trigger all sorts of app's behaviors. If certain malicious I/O patterns or behaviors are identified (e.g., sending SMS to premium numbers), the app is considered as malware. Though static analysis avoids the cost of running app and is usually more efficient, it could be thwarted when obfuscation or packing techniques are employed. As shown in the work by Rastogi et al. [23], common malware transformation techniques could make malicious apps evade popular static analysis tools at high success rate. On the other hand, dynamic analysis is robust against code-level evading techniques and is suitable for processing apps with complicated program logics. A corpus of frameworks have been developed and proved to be effective, including DroidScope [31], AppsPlayground [22], CopperDroid [26], etc. Google also developed its dynamic analysis framework, Bouncer [16], to check every app submitted to Google Play.

3.1.1 Input Generation and Automated Exploration

Since app's runtime behaviors often depend on the inputs from the user or system, the effectiveness of the dynamic analysis framework highly depends on the strategy of input generation.

Comparing to the traditional PC malware, which tend to take malicious actions (e.g., controlling the system) once executed, mobile malware tend to delay the malicious actions till a sequence of events are observed (e.g., hijacking the legitimate app and stealing the received messages). Therefore, the testing platform should be able to generate the input in a context-aware manner and explore the execution paths automatically. Below, we describe two widely adopted strategies in automated path exploration:

Fuzzing-based Exploration. Fuzzing is a black-box testing technique in which the system under test is stressed with invalid, unexpected or random inputs transmitted from external interfaces to identify the bugs in programs [25]. On the Android platform, Google provides an official fuzzer Monkey [8], which generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events and injects them into the framework through Android Debug Bridge (ADB). Several dynamic analysis frameworks have incorporated Monkey as the exploration engine, such as VetDroid [32] and Andrubis [15].

Model-based Exploration. On the contrary, model-based testing aims at injecting events aligning with a specific pattern or model which could be derived by analyzing the app's code or UI. The test cases generated are usually more effective and efficient in discovering malicious activities. To support this testing mode, Google has developed an exploration tool named MonkeyRunner [5] which allows testing platform to interact with an app in pre-defined event sequences. MonkeyRunner has been adopted by several testing platforms including Mobile-Sandbox [24], CopperDroid [26], etc.

In the course of automated UI interactions, a large number of invalid actions could be triggered if the properties of UI structure is disregarded. As a solution, Google developed UI Automator [7], which inspects the layout hierarchy and device status to decide the meaningful UI actions. Besides, AppsPlayground [22] leveraged a number of heuristics to customize inputs for certain UI controls (e.g., login box). CuriousDroid [10] decomposes the on-screen layout and creates context-based model on-the-fly. SmartDroid [33] uses a hybrid model which extracts call graphs through static analysis and initiates actions leading to sensitive APIs.

3.2 Motivation: Programmed Interaction

The main design goal of the above frameworks is to explore *all* potential paths leading to malicious behaviors *efficiently*. As such, the input events they generated are usually predictable, fired at regular and short interval, and massive for good coverage, which significantly differ from what are produced by human users. Hence, leveraging this insight, we design a new mechanism to capture such *programmed interactions* and distinguish human users from testing platforms. We envision our approach could be implemented as a component (we call it `PIDetector`), embedded within Android malware and monitoring the system events of its interests. Before the execution of malicious payload, the collected event sequence will be analyzed by `PIDetector`, and the execution only proceeds when the event sequence is determined to be produced by human user.

Compared with anti-emulation techniques, our approach offers another layer of protection to malware even analyzed on bare-metal platforms. It is also robust against the upgrades which alter the observable artifacts by analysis frameworks. At the high level, our approach can be considered as a variant of CAPTCHA [30] – humans can pass, but computer programs can't pass. In fact, the state-of-art text or image based CAPTCHA schemes may achieve the same or even better accuracy in distinguishing human and computer. However, asking user to solve CAPTCHA before using the app would drive away many users and reduce the infection rate. In contrast, such issues are not embodied in our approach.

3.3 Assumptions

Our approach intends to evade the detection by dynamic analysis. Evading static analysis is out of the scope of our work. In fact, such task could be fulfilled by off-the-shelf obfuscators and packers.

We also assume the dynamic analysis platforms interact with the testing app through events injection, and the execution logic of the app cannot be forcefully altered, i.e., bypassing `PIDetector` and directly invoking malicious payloads. This strategy is in theory possible but requires precise analysis on app's code to identify the critical branches, which is quite challenging and again vulnerable to obfuscation and packing techniques. This setting is also adopted by all previous works on evading dynamic analysis [21, 28, 14, 17].

4. ATTACK VECTORS

In this section, we elaborate several attack vectors that can be leveraged to detect programmed interactions. Overall, the qualified attack vectors should fulfill the three requirements below:

- *Reverse Turing Test* – humans can pass, but current exploration tools can't pass.
- *Passive* – hard to be discovered by end-users.
- *Lightweight* – easy to be built and deployed.

Given these constraints, we design two classes of attack vectors targeting the vulnerabilities underlying event injections and UI element identification in dynamic analysis. To notice, some testing platforms built upon Monkey can be trivially identified through invoking the `isUserAMonkey()` API [3] and inspecting the returned value. We do not include it into the attack vectors as the returned value can be easily manipulated (e.g., it can be bypassed by UI Automator through calling `setRunAsMonkey(false)` [18]). We elaborate each attack vector in the following subsections.

4.1 Detecting Simulated Event Injections

We found the data attached to two types of user events, MotionEvent [6] for touchscreen tapping and KeyEvent [4] for key pressing, can be leveraged for detection. It turns out the both individual event and event sequence reveal distinguishable patterns.

4.1.1 Single Event

When a user operates a mobile device, the events are initiated by the onboard hardware and the information regarding the hardware is attached. To the opposite, the events injected by dynamic testing tools, like Monkey, are passed from external interfaces and most of the parameters are filled with dummy values. Specifically, while the core parameters (e.g., coordinates of input location) are filled with real values, the auxiliary parameters (e.g., keyboard type) are not filled similarly.

Table 1 and Table 2 list differences between the values generated from real-world usage and Monkey testing for MotionEvent and KeyEvent. Clearly, Monkey fills the values in a distinctive pattern that can be identified. For example, the ToolType parameter of KeyEvent generated by Monkey is always `TOOL_TYPE_UNKNOWN`, which cannot be used if this event is produced by hardware.

4.1.2 Event Sequence

To reach the high coverage of app behaviors in limited time, dynamic analyzers tend to inject events at high frequency which

Table 1: MotionEvent: real vs. simulated (by Monkey)

Parameter	Real	Simulated
ToolType	1: TOOL_TYPE_FINGER	0: TOOL_TYPE_UNKNOWN
DeviceId	[non-zero value]	0
Device	valid	null

Remarks: 1) DeviceId: zero indicates that the event does not come from a physical device and maps to the default keymap.

Table 2: KeyEvent: real vs. simulated (by Monkey)

Parameter	Real	Simulated
ScanCode	[non-fixed value]	0
DeviceId	[non-fixed value]	-1
Device.Name	[non-fixed value]	Virtual
Device.Generation	[non-fixed value]	2
Device.Descriptor	[non-fixed value]	af4d26ea4cdc857cc0f1ed1ed51996db77be1e4d
Device.KeyboardType	1: non-alphabetic	2: alphabetic
Device.Source	[non-fixed value]	0x301: keyboard dpad

Remarks: 1) ScanCode: the hardware key id of the key event; 2) Generation: the number is incremented whenever the device is reconfigured and therefore not constant; 3) Descriptor: the unique identifier for the input device; 4) KeyboadType: the value is "non-alphabetic" as the nowadays smartphone models do not integrate hardware keyboards.

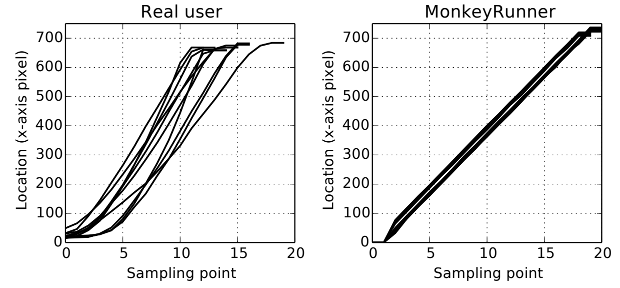


Figure 1: Swiping trajectory: real user vs. exploration tool

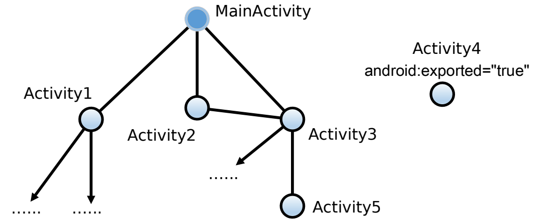


Figure 2: Example of isolated Activity

cannot be performed by human users. Therefore, by measuring the frequency of the events the dynamic analyzers could be identified. Also, the distribution of events along time series is also unique for dynamic analyzers, and we show how this observation could be leveraged for our purposes. As one example, the key presses are usually issued at changing speed when a user types text in EditText while the interval is fixed for dynamic analyzers. IME partially causes this: an IME will show up when a user taps EditText and due to the variance of the distances between characters on IME, the interval between key presses fluctuates.

From the aspect of MotionEvent series, Android provides standard APIs for an app to recognize touch gestures inputted by user. At the same time, a series of screen touching events (MotionEvent) can be observed, and the events are issued much more regular if from dynamic analyzers. As an example, we asked a participant to swipe the touch screen on Samsung Galaxy S III from far left to far right and directed MonkeyRunner for the same action. The test was ran 10 times and we draw the tap locations in x-axis (`float x` field of MotionEvent) against 20 sample points at the same interval in Figure 1. The trajectories of the swipes from the user are rather dispersed, and displacements at the start and end of the action are smaller. In contrast, Monkey's swipes are highly similar and are moved at constant speed. Such difference could be modeled through time series similarity measure related algorithms.

4.2 Implanting UI Traps

To increase the chance of triggering malicious activities, especially the ones associated with user behaviors, dynamic analyzers have to explore and interact with as many UI elements as possible. Such design, however, leads to a dilemma that can be exploited: the adversary could implant UI traps that are inaccessible to human users and unable to be distinguished by dynamic analyzers. Below we elaborate the designs of two such attack vectors:

4.2.1 Isolated Activity

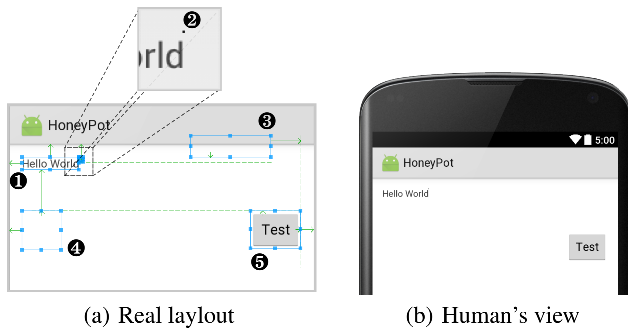
An Android app defines the UI interface and routines for event processing in *Activity* component, which is also declared in the Manifest file. An app usually contains one main Activity and subsequent Activities that can be transitioned to, as shown in Figure 2. In addition, developers could export an Activity that can be launched by other apps (Activity4 in Figure 2), through setting `android:exported="true"`. Common dynamic analyzers tend to parse the Manifest file and visit Activities in both cases while the users follow the defined interaction logic to visit Activities. This motivates us to create an *isolated Activity* which could not be reached through interaction as a trap: if an unused and exported Activity is invoked, the party behind should be dynamic analyzer. Such trap is hard to be detected ahead, as the interaction logic is defined in app's code and can be obfuscated.

4.2.2 Invisible UI Elements

We demonstrate another attack vector here which manipulates the visual effects of UI elements. A human user normally taps the UI elements she can see to input information. On the other hand, dynamic analyzers could invoke system APIs (such as `parseViewHierarchy` under `DeviceBridge` class) to identify *valid* elements and simulate the interaction. Therefore, we could leave a valid but *"invisible"* control in Activity as a trap, and use it to detect the dynamic analyzer falling in.

In fact, Android UI library provides a set of highly customizable UI controls for developers, including `Button`, `TextView`, `Switch`, etc. and they can be easily configured to be invisible. One such control is `ImageButton` that displays image as a button and accepts clicks. If the image is filled with the same color as the nearby pixels of background, it would not be discovered by the human user. Moreover, we could change the layout and location of the controls for the same purpose. For instances, we could set the control to be tiny (e.g., 1 pixel) or fall out of the screen.

Figure 3 explains this idea. From human's view, only a button (Test) and a sentence (Hello World) can be identified. Actually, 5 UI controls are embedded in the layout: ❶ text view, ❷ image button, ❸ switch, ❹ image button, and ❺ button. They are all valid to dynamic analyzers. For example, we ran Google UI Automator to parse the UI layout and it recognizes all controls and extracts the information listed below:



(a) Real laylout (b) Human's view

Figure 3: Invisible UI elements: real layout vs. human's view

```
(0) FrameLayout[0,0][720,1280]
  (0) View[0,50][720,146]
  (1) TextView:Hello World[32,178][178,216]
  (2) ImageButton[176,178][178,180]
  (3) Switch:ON[408,124][614,180]
  (4) ImageButton[32,318][132,418]
  (5) Button:Test[560,318][688,414]
```

All these controls will be exercised by dynamic analyzers, but only ❶ and ❺ could be noticed by human. Therefore, events fired on other controls could indicate the existence of dynamic analyzers. To escape from this type of traps, dynamic analyzers have to accurately determine the visibility of UI controls (to human eyes), which remains an open problem.

5. EVALUATION

In this section, we present the experimental results on the effectiveness of PIDetector. The testing methodology is introduced first, and the results are discussed after.

Testing Methodology. We developed a testing app integrating PIDetector as an internal component. Our goal is to examine whether PIDetector can correctly detect dynamic analyzers through one or more attack vectors. Therefore, the testing app only has basic UI and functionalities, and we did not include malicious payload into the testing app to reduce the impact to the operationalized platforms. When the app is loaded, PIDetector starts to collect raw logs regarding fired events (details are provided below). The raw logs were sent to an HTTP server set up on Amazon Elastic Beanstalk [1] and analyzed through the models implemented by us. We decide to exfiltrate the event data to gain a better understanding of the behaviors of testing platforms. All analysis can be done locally when adopted by adversary's real app.

Raw Logs. The collected raw logs include the following information: the parameters of captured MotionEvent and KeyEvent objects, invoked Activities, visited UI elements. Every returned log is padded with a unique ID to distinguish different testing platforms and times of running. Only the first 100 logs are transmitted to the server to obtain enough data and avoid excessive network connections, which might be considered as anomaly activities by testing platforms. We also collected the configuration information of every tested service, such as Android ID, IMEI, Build.SERIAL and Build.MANUFACTURER, to see if countermeasures against anti-emulation are deployed.

Testing Platforms. We tested 10 dynamic analysis services built for malware analysis, among which four come from the academia while the others come from the security companies. In addition, we upload our testing app to Google Play to test its official dynamic analyzer, Google Bouncer. These 11 services are listed in Table 3. The experiments were conducted in January and March 2016.

We were able to obtain valid raw logs from 7 services, and the final results are summarized in Table 3. Among the remaining ones, A5 and CopperDroid refused to analyze our app, since the processing queue has been fully occupied. No raw logs or informative messages are returned for Payload Security and Malwr, and we speculate the causes are: 1) they only launch static analysis on our app; 2) The outbound network connections from app are blocked.

Finding 1. *Nearly all (available) analysis services are vulnerable to at least one attack vector.* Most of them could be identified by analyzing single event parameter, except TraceDroid for lacking enough parameters and Google Bouncer for filling valid values. For example, the Input Device parameter of the KeyEvent from SandDroid was always "-1". Isolated Activity feature is also quite effective, and half of these online services fell into this trap. On the other hand, only Tecent Habo hit invisible elements, and no service was found to generate continuous event sequence (e.g., swipe). We suspect that these interactions are missed because complex UI analysis and interactions are not performed.

Finding 2. *Emulator camouflaging or physical device has been deployed by online analysis systems.* For example, we found the platform configuration of Google Bouncer is quite like physical device – Google Nexus 5 or 6, as showing below:

```
Version: 6.0.1              SDK_INT: 23
MODEL: Nexus 6              BRAND: google
BOARD: shamu                DEVICE: shamu
HARDWARE: shamu             SERIAL: ZX1G22HMB3
ID: MMB29K                  PRODUCT: shamu
DISPLAY: MMB29K             MANUFACTURER: motorola
HOST: wped2.hot.corp.google.com
BOOTLOADER: moto-apq8084-71.15
FINGERPRINT: google/shamu/shamu:6.0.1/MMB29K
    /2419427:user/release-keys
```

To notice, emulator camouflage has been used for other purposes on Android platform. BlueStacks [2], a popular emulator designed for running Android games on Windows and Mac platforms, camouflages itself as certain models of Samsung devices to evade emulator detection performed by apps. Hence, we believe our techniques for programmed interaction detection is meaningful even in the short term to attackers.

6. DISCUSSION

Limitations. As countermeasures, the developers of dynamic analyzers could change the UI interaction pattern and make the testing process closer to human beings. For example, the dummy parameter values of the injected MotionEvent and KeyEvent could be changed to use real data. On the other hand, how to hide against the more complicated attack vectors we devised (e.g., event sequence) is unclear. Though user's interactions on App UI can be recorded and replayed, challenges have to be addressed on how to automatically adjust the recorded actions to different apps.

7. CONCLUSION

In this work, we propose a new approach to evade Android runtime analysis. This approach focuses on detecting programmed interactions to determine whether an app is under analysis, instead of relying on the traditional emulator detection. The preliminary experimental results have demonstrated the effectiveness of our methods. We believe the evasive techniques leveraging subtleties of human-computer interaction should be seriously considered by

Table 3: Experimental results for online dynamic analysis services

Service Name	URL	Simulated Events			UI Traps	
		MotionEvent Paramters	KeyEvent Parameters	Event Sequence	Isolated Activity	Invisible UI Elements
NVISO ApkScan	https://apkscan.nviso.be	$\surd$	$\surd$	$-$	$-$	$-$
SandDroid	http://sanddroid.xjtu.edu.cn	$\surd$	$\surd$	$-$	$\surd$	$-$
TraceDroid [27]	http://tracedroid.few.vu.nl	$\times$	$\times$	$-$	$\surd$	$-$
Anubis [15]	http://anubis.iseclab.org	$\times$	$\surd$	$-$	$\surd$	$-$
Tecent Habo	https://habo.qq.com/	$\surd$	$\surd$	$-$	$-$	$\surd$
VisualThreat	https://www.visualthreat.com	$\surd$	$\surd$	$-$	$-$	$-$
Google Bouncer	N/A – no public link	$\times$	$-$	$-$	$-$	?
A5 [29]	http://dogo.ece.cmu.edu/a5/	The upload process always reported error.				
CopperDroid [26]	http://copperdroid.isg.rhul.ac.uk	Too many submitted samples were queued.				
Malwr	https://malwr.com	No raw log was returned.				
Payload Security	https://www.hybrid-analysis.com	No raw log was returned.				

Remarks: 1) "$\surd$": Judged as programmed interaction. 2) "$\times$": Judged as human interaction. 3) "$-$": Not triggered or found. 4) "?": Google Bouncer clicked all buttons on the main Activity but ignored the image button which was camouflaged as a normal button by us. We speculate Bouncer only triggers the UI controls with the `Button` property by design. Since this is indirect evidence, so we label it as "?".

security community and call for further research on closing the gap between machine and human in runtime behaviors.

8. ACKNOWLEDGEMENTS

We thank anonymous reviewers for their insightful comments. This work was partially supported by NSFC (Grant No. 61572415), and the General Research Funds (Project No. CUHK 4055047 and 24207815) established under the University Grant Committee of the Hong Kong Special Administrative Region, China.

9. REFERENCES

[1] AWS Elastic Beanstalk. http://aws.amazon.com/elasticbeanstalk/.

[2] BlueStacks. http://www.bluestacks.com/.

[3] isUserAMonkey(). http://developer.android.com/reference/android/app/ActivityManager.html#isUserAMonkey().

[4] KeyEvent. http://developer.android.com/reference/android/view/KeyEvent.html.

[5] MonkeyRunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[6] MotionEvent. https://developer.android.com/reference/android/view/MotionEvent.html.

[7] Testing Support Library. https://developer.android.com/tools/testing-support-library/index.html.

[8] UI/Application Exerciser Monkey. http://developer.android.com/tools/help/monkey.html.

[9] G DATA Mobile Malware Report - Threat Report: Q4/2015. https://secure.gd/dl-us-mmwr201504, 2016.

[10] P. Carter, C. Mulliner, M. Lindorfer, W. Robertson, and E. Kirda. CuriousDroid: Automated User Interface Interaction for Android Application Analysis Sandboxes. In *Financial Cryptography and Data Security - 20th International Conference, FC 2016, Revised Selected Papers*, 2016.

[11] C. Dietzel. Porting and Improving an Android Sandbox for Automated Assessment of Malware. Master's thesis, Hochschule Darmstadt, 2014.

[12] J. Gajrani, J. Sarswat, M. Tripathi, V. Laxmi, M. S. Gaur, and M. Conti. A Robust Dynamic Analysis System Preventing SandBox Detection by Android. In *Proceedings of the 8th International Conference on Security of Information and Networks (SIN)*, 2015.

[13] W. Hu and Z. Xiao. Guess Where I am: Detection and Prevention of Emulator Evading on Android. *XFocus Information Security Conference (XCon)*, 2014.

[14] Y. Jing, Z. Zhao, G. Ahn, and H. Hu. Morpheus: Automatically Generating Heuristics to Detect Android Emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*, 2014.

[15] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. ANDRUBIS-1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proceedings of the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[16] H. Lockheimer. Android and Security. http://googlemobile.blogspot.com/2012/02/android-and-security.html, 2012.

[17] D. Maier, M. Protsenko, and T. Müller. A Game of Droid and Mouse: The Threat of Split-Personality Malware on Android. *Computers & Security*, 54:2–15, 2015.

[18] A. Momtaz. Allow for setting test type as a monkey. https://android.googlesource.com/platform/frameworks/base/+/8f6f1f4%5E!/, 2013.

[19] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna. Baredroid: Large-scale Analysis of Android Apps on Real Devices. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, 2015.

[20] S. Neuner, V. van der Veen, M. Lindorfer, M. Huber, G. Merzdovnik, M. Mulazzani, and E. R. Weippl. Enter Sandbox: Android Sandbox Comparison. In *Proceedings of the 2014 IEEE Mobile Security Technologies Workshop (MoST)*, 2014.

[21] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware. In *Proceedings of the Seventh European Workshop on System Security (EuroSec)*, 2014.

[22] V. Rastogi, Y. Chen, and W. Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[23] V. Rastogi, Y. Chen, and X. Jiang. Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks. *IEEE Transactions on Information Forensics and Security (TIFS)*, 9(1):99–108, 2014.

[24] M. Spreitzenbarth, F. C. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. Mobile-Sandbox: Having a Deeper Look into Android Applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC)*, 2013.

[25] A. Takanen, J. DeMott, and C. Miller. Fuzzing Overview. In *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008.

[26] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[27] V. van der Veen. Dynamic Analysis of Android Malware. Master's thesis, VU University Amsterdam, 2013.

[28] T. Vidas and N. Christin. Evading Android Runtime Analysis via Sandbox Detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014.

[29] T. Vidas, J. Tan, J. Nahata, C. L. Tan, N. Christin, and P. Tague. A5: Automated Analysis of Adversarial Android Applications. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM)*, 2014.

[30] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford. CAPTCHA: Using Hard AI Problems for Security. In *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Proceedings*, 2003.

[31] L. Yan and H. Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[32] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[33] 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 2012 ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2012.