

Monkey Gamer: Automatic Profiling of Android Games

Javier Marián Santos, Simin Nadjm-Tehrani, Aruna Prem Bianzino
 Department of Computer and Information Science, Linköping University, Sweden
 Email: simin.nadjm-tehrani@liu.se

Abstract—Creation of smartphone applications has undergone a massive explosion in recent years and there is an urgent need for evaluation of their resource efficiency, trustworthiness and reliability. A large proportion of these apps are going to be within the gaming area. In this paper we classify game apps on the basis of their development process, their I/O process and their interaction level. We present Monkey Gamer, a software to automatically play a large class of Android games and collect execution traces, based on a state machine to partially describe the game structure and interactions. A significant similarity is shown when comparing the results obtained by the Monkey Gamer and by human players, for three of the most popular Android games. We evaluate the performance of the Monkey Gamer by comparing the traces it generates with traces created when humans play the games, and find significant similarity in the trace sets.

I. INTRODUCTION

The introduction of the app-driven software deployment for mobile devices is redrawing the application software development map for a significant proportion of interactive software. Many players, including software developing companies and independent developers, are given the opportunity to enter the app market and to distribute them in the virtual shops for a widespread dissemination.

This new dynamics motivates a new look at the application development cycle and efficient investigations on a mass market scale. Heterogeneity in the range of available apps gives rise to the need for tools to easily and automatically test their efficiency, trustworthiness and reliability - either with a common denominator, a generic tool that can treat aspects of many apps, or with a tool that is tailor-made to a significant subset of the apps appearing. Clearly, application profiling, creating logs of system status, system calls, user inputs, access to sensitive data, network usage, energy-consuming features, etc, can be of great help for creation of future system software and hardware producers, as well as for research.

Games represent a big share of the available apps. It has been estimated that 20% of the applications available in the iPhone app store are games, and about 80% of application downloads concern games [1]. Other more conservative estimations still claim that games are responsible for 36% of the downloads in iTunes and for 22% of the downloads in Play Store [2]¹. Game apps constitute a challenging case from a testing and profiling point of view, as games require a

higher level of interaction with the user – potentially within a simultaneous group of users – compared to other apps. Games usually present a more complex structure including different screens and different activities.

In this paper, we take two important steps towards automatically profiling and analysing game behaviours: (i) we introduce a game classification, which represents a basis for a better understanding of the game app population, and (ii) we develop Monkey Gamer, a software to automatically play Android games belonging to many of these classes, emulating their behaviour and producing play traces similar to human-player traces.

Key elements in automatically playing games are the identification of the interaction points in the screen and the kind of input that the game is taking. Our game classification highlights these aspects, in order to understand which population of games can be profiled using Monkey Gamer, and how to design the tool to be able to play these games. We estimate that Monkey Gamer is able to profile about 45% of the top 100 downloaded free games on the Google Play Store, as for the Swedish market [3].

The goal of Monkey Gamer is to support profiling of an Android game, without the use of expensive human players and human time consumption. Game profiling by Monkey Gamer results in a sequence of inputs and the corresponding execution traces, in terms of system status, system calls, access to sensitive data, network usage, and game state. The input sequence is not pre-generated, but constructed on-the-fly, on the basis of the current game and system state. The Monkey Gamer should behave as similarly as possible to a human player, in order to keep the obtained profiling meaningful.

We validate Monkey Gamer considering three popular games, accounting for millions of downloads. The selected games represent different levels of game complexity, different sets in the game classification, different underlying game development procedures, and different input types (tap and swipe). Monkey Gamer performance is validated in terms of similarity to real human player traces, showing high resemblance, and provides a deeper exploration of game states.

Previous approaches to automatic profiling Android applications have been targeting just a specific game [8], or just a specific aspect of games, like the game energy states [7], game network connections [11], game security [9] or privacy, i.e., access to user sensitive data by games [10]. A number of works test generic Android applications, without specifically targeting games, but they either relay on completely random

¹Original data from the mobile app search company Chomp, now dismissed.

input generation [14], application-specific input generation [15], a properly tuned mix of random input and input paths generated from a given application-specific seed [11], or a prior knowledge of all the interactive elements present in each game state [12]. Finally, there are approaches to test generic Android applications from a holistic point of view, but using human players instead of automated input [13]. Our work is the first attempt at applying a generic and sophisticated profiling technique to a large class of Android applications, a major subset of the games available on the market.

The rest of the paper is organised as follows: Sec. II presents our classification of Android games. Sec. III describes Monkey Gamer, while Sec. IV validates it considering real and popular games. Finally, Sec. V concludes the paper and presents future work directions.

II. ANDROID GAME CLASSIFICATION

The interaction required by Android games while playing them may vary a lot, both from the physical and the technical point of view. A deep understanding of this variability is a fundamental starting point for automatic game profiling. On the one hand, it helps to understand which games can be automatically profiled and which can not. On the other hand, it characterises the relevance of such analysis in terms of game population coverage. At the same time, such game classification can be of help for other kinds of studies concerning user/app interaction and app profiling in general.

In the remainder of this section we discuss the different kinds of user/game interactions, as well as the different choices available to developers to define the game Graphical User Interface (GUI).

A. User Input

Some games can be described as a discrete state machine, where the game enters a state and waits for a user input before entering the following state. A familiar example of this class of games may be the game of chess. We refer to this class of games as “time-independent”. Other games change state in a continuous fashion, partially independent from the user input, and respond to a specific input, depending on the instantaneous state. A familiar example of this class of games may be any car racing or platform game, where the state of the game changes even when the player is not interacting with the game. We refer to this class of games as “time-dependent”.

From the point of view of automatic profiling of games, the time-dependent games represent a huge challenge as the state space of a time-dependent game is infinite, and the Monkey Gamer would have to produce input in a continuous fashion. For this reason, we decided to narrow our scope to the profiling of time-independent games. Still 49 of the 100 most downloaded free games on the Google Play Store on the Swedish market were time-independent games (at the time of our sampling²), making this subset interesting enough.

In this work we consider user/game interactions through a touch screen. Other inputs may be given by users through

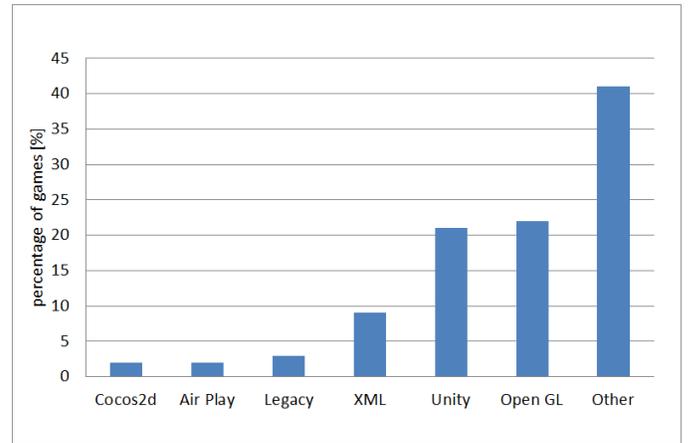


Fig. 1: Distribution of games by GUI implementation technology.

other sensors, such as accelerometers, or microphones, but when considering the time-independent game population a very common mode of input generation is through the touch screen interface.

B. GUI Implementation Technology

Users interact with games through the touch screen by touching specific *interactive* objects. Game developers can use different technologies to implement the GUI. The GUI can be defined using an xml file – a common practice for non-game applications. As an alternative, game developers can use specific game engines to define the GUI of a game. Game engines provide game developers with specific development tools to manage game graphics, physics engines, animations, artificial intelligence, etc. These frameworks make the use of game engines quite popular. At the same time, different game engines are available depending on the developer needs. The result of this process is a high heterogeneity in the game development tools used for games.

The distribution of the GUI implementation technologies among the 100 most downloaded free games on the sampled Google Play Store is reported in Fig. 1. This analysis shows that an approach to automatic profiling games that targets a specific development technology would be much less relevant than a generic one, due to the high heterogeneity of development technologies.

III. THE MONKEY GAMER

Monkey Gamer is a program executed on a computer that controls a real (Android) device. It plays a game on the Android device, generating automatic input and collecting traces about the game state, the system state, system calls and network connections generated by the game, etc.

A. Architecture of the profiler

The main blocks comprising the Monkey Gamer are presented in Fig. 2

²The game population analysis was performed on November 28th 2012.

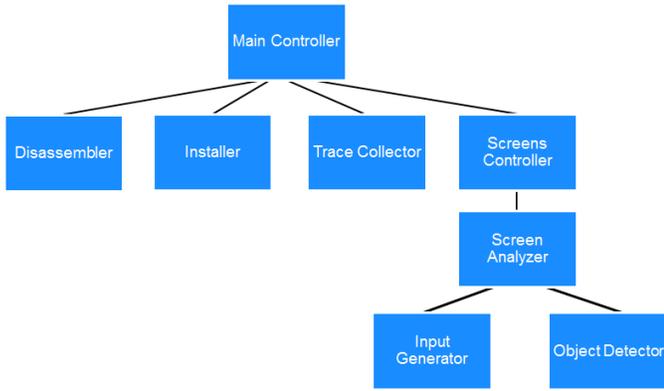


Fig. 2: Structure of Monkey Gamer as function blocks.

The *Main Controller* block takes as input the location of the .apk file of the game to be analysed and is responsible for coordinating the other modules and collecting the execution trace.

The *Disassembler* block is in charge of disassembling the .apk file of the analysed game, while the *Installer* block installs a clean instance of the game in the physical Android device.

The *Trace Collector* block is responsible for collecting system calls and network traffic.

The *Screens Collector* module governs the state machine associated with the game and is the key module of Monkey Gamer. Iterations with the game are executed in sequence in order to simulate a played game. Each interaction corresponds to a sequence of operations: (i) wait for the screen to stabilise (screen switch may include animations, a proper switching time t_s is selected), (ii) take a screenshot and check the system state, (iii) determine the game state, (iv) retrieve the information about the interactive object on the screen and (v) generate the following input.

In particular, a specific module, i.e., the *Screen Analyser*, is responsible for step (iv) described above. The *Screen Analyser* returns the interactive elements of the current game screen, being buttons, images, or text fields. Interactive elements may be clearly identifiable, i.e., if the GUI is defined through an xml file, every object on the screen is uniquely identified in the xml file, and interactive objects are identified by specific parameters (e.g., clickable = true). On the other hand, if the GUI is defined using a game engine, it is not possible to access a file describing the different objects and their properties since game engines are not standard. In the latter case, interactive objects must be identified from the screenshot itself, and this operation is relegated to the specialised *Object Detector* module.

Finally, the *Input Generator* module is responsible for generating the subsequent input to the game. This step includes the selection of the object to interact with, the kind of interaction (type text, press a hard key, touch or swipe), and eventual interaction parameters (e.g., text to enter, or direction of the swipe gesture). Each screen is associated with the list of the detected interactive objects it contains and the number of times the screen has been visited before. The



Fig. 3: Different screens of the *Tic Tac Toe Free* game, corresponding to different game states.

object to be interacted with is selected at each step considering every interactive object and every possible permutation of the interactive objects list. For example, if the considered screen contains three different interactive objects, the sequence of objects with which to interact at each visit of the screen will be $\{1, 2, 3; 1, 3, 2; 2, 1, 3; 2, 3, 1; 3, 1, 2; 3, 2, 1\}$, where 1, 2, and 3 are the object identifiers.

B. Game States

Game profiling strongly relies on the definition of game states, as games are modelled as a state machine, but no agreed standard definitions for “game state” exists. States for games have been defined in a highly coarse scale [7], or for very specific games [8]. Our methodology needs to rely on a definition of game state which is generic enough to be applicable to all the games in the selected class, while detailed enough to differentiate different game stages as different game states. Thus, we define the state of a game as a point within the game, characterised by a specific screen output and a specific set of interactive elements. Examples of different game screenshots, corresponding to different game states, for the *Tic Tac Toe Free* game are depicted in Fig. 3. Each state is characterised by a different screen layout and a different set of interactive elements. Note that similar screenshots, still characterised by the same layout and the same set of interactive elements, are mapped to the same game state. An example of different screenshots mapped to the same game state, for the *Angry Birds* game, is reported in Fig. 4.

From an implementation point of view, the *activity* class is defined for mobile applications as a task container, but the use of game engines makes this class not uniquely corresponding to our definition of game state, since different activities may correspond to different states, but a single activity can also hold different states. That justifies our choice which maps states in the game state machine to the screenshots. At every stage of the game profiling, Monkey Gamer needs to know whether it is visiting a new state of the game or an already known one. This is needed both to know where to map the state in the game state machine, and to generate the next input to the game.



Fig. 4: Different screenshots of the *Angry Birds* game mapped to the same game state.

The screen comparison is implemented by means of both a pixel-by-pixel comparison and a histogram comparison. The pixel-by-pixel comparison guarantees excellent results in case of static screens (i.e., screens without animations). For each pixel in a screen, its values of RGB brightness are compared with the ones of the corresponding pixel in the reference screen and the difference is contrasted against a proper threshold σ_p . The comparison returns the percentage of similar pixels.

The histogram comparison returns a value of similarity which is global for the picture, representing a better comparison method for dynamic screens (i.e., screens including animation). In this comparison, for each screen a histogram is produced representing the composition of the distributions of the brightness values for the three RGB components of the pixels composing the screen. The comparison between two histograms returns the percentage of similar elements in the histograms of the two compared screens, again under a properly selected threshold σ_h . The final screen comparison is a linear composition of the two similarity indexes returned by the pixel-by-pixel comparison and by the histogram comparison³. It is compared to a global threshold σ_g to decide whether the

³The similarity index returned by the pixel-by-pixel comparison goes from 0, i.e., no similarity, to 1, i.e., exactly the same picture. The same applies to the histogram comparison. The global index is the sum of the two comparison indexes. Hence, the global similarity index goes from 0, i.e., no similarity at all, to 2, i.e., exactly the same picture.

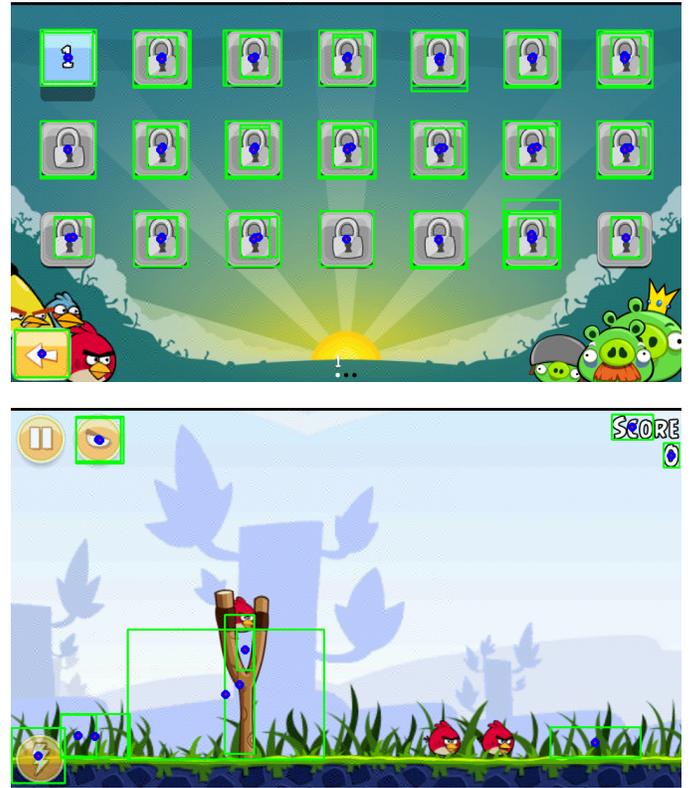


Fig. 5: Examples of object detection in the *Angry Birds* game.

screens belong to the same game state or not.

C. Detailed implementation

Some of the previously described tasks have been solved using available tools. In particular, the *Main Controller* is a monkeyrunner script [4], while for the *Trace Collector* module, system calls are collected using the `strace` tool, and network traffic is monitored using the `tcpdump` tool. The screen comparisons performed by the *Screen Controller* module have been implemented using the PIL libraries.

The other described tasks could not be solved using already available tools, and custom solutions have been developed. In particular, object detection in games the GUI of which was defined using game engines has been implemented using the Hough Transformation [5].

Interactive objects are usually sharp edges, contours, or shapes, being buttons or touchable elements on the screen. The Hough Transformation is particularly good in detection of this kind of objects in an image, like a screenshot. Open source libraries for implementing the Hough Transformation are available [6]. An example of object detection results for two different screenshots of the *Angry Birds* game is presented in Fig. 5, using the (green) highlighted boxes. As we can see, most of the interactive objects are correctly detected, especially for regular shaped objects (e.g., the square buttons in the top screen), but some interactive objects are not detected (e.g., the

Symbol	Value	Description
σ_o	500 square pixels	Minimum size for interactive detected objects to be considered.
t_s	2 s	Waiting time for screen switching.
T	14-21 hours	Simulation duration.
σ_p	10	Pixel-by-pixel comparison threshold for the difference in luminosity, for each RGB component.
σ_h	100 pixels	Histogram comparison threshold for the difference in luminosity distribution, for each RGB component.
σ_g	1.2	Global threshold for screen comparison.

TABLE I: Simulation parameters.

pause button in the top left corner of the bottom screen) and some interactive objects are detected while they are not present (e.g., a grass block in the bottom right corner of the right side screen).

The right tradeoff between false positives and false negatives in interactive object detections may be tuned by a suitable selection of the parameters of the Hough Transformation and by the definition of proper heuristics (e.g., do not consider “too small” objects, having a size of less than σ_o square pixels). Finally, some times a part of the screen is defined using an xml file, while another part is defined using a game engine. In this case the two parts can be easily identified in the xml file, and a hybrid object detection is performed by the *Object Detector* module.

IV. PROTOTYPE VALIDATION

We validate the Monkey Gamer in terms of similarity to the game profiling obtained by real human players. This section details the evaluation setting and methodology, and describes the obtained results.

A. Validation Scenario

The testing environment consists of a Dell Latitude E6320 laptop, Intel Core i7-2620M CPU at 2.70GHz x 4, 8GB memory, running Ubuntu Linux 12.04 64-bit, an HTC Sensation Z710e smartphone, 1.2GHz Qualcomm MSM8660 Dual-Core, running Android Gingerbread 2.3.5 (developers build), and an LG-P990 smartphone, ARMv7 Processor rev0, running Android Jelly Bean 4.2.2 (CyanogenMod). The phones were connected to the Internet via a general purpose WiFi network (eduroam). Results are reported in the rest of the paper used the HTC phone, while tests have been executed also on the LG phone to test the Monkey Gamer compatibility in terms of hardware and OS version.

The evaluation was carried out with three Android games, representing a heterogeneous selection with respect to the game population described in Sec. II. In particular, the three selected games are *Tic Tac Toe Free* (TTTF), *Bubble Worlds* (BW), and *Angry Birds* (AB). TTTF represents a simple game, with a reduced and countable number of states. Its GUI is defined using an xml file and the player/game interaction is limited to taps. BW represents a more complex game, resulting in an uncountable number of final results. Its GUI

is defined using the OpenGL game engine (the most popular game engine, according to our game population analysis). The player/game interactions in BW are still limited to tap gestures. Finally, AB is a really popular game, representative of a higher complexity. It accounts for higher variety among screens, an uncountable number of final results, and requires both tap and swipe gestures to be played. The AB’s GUI is defined using the Cocos2d game engine. All the selected games are among the top 100 downloaded free games on the Google Play Store, as mentioned before, and account for millions of downloads each.

The obtained execution traces have been compared to traces obtained while human players were playing the same games in the same setting. Three person have been asked to play each game, providing a total of nine different human players. Each human playing session was over a period lasting between 15 and 30 minutes.

B. Parameter Selection

Simulation parameters were empirically selected on the basis of the worst case among the observed ones, e.g., longest screen transaction time for the t_s parameter, or smallest interactive object for the σ_o parameter. The simulation durations were chosen in order to obtain the same number of inputs generated by the human players while playing a specific game, resulting in 21 hours for TTTF, 16 hours for BW, and 14 hours for AB⁴. Parameter values are summarised in Tab. I.

C. Validation Results

For each analysed game, we have three human execution traces and one Monkey Gamer trace. Each trace includes the sequence of the game system calls, the TCP traffic generated and received by the game, the player input sequence and a screen shot of each game step (i.e., after each input).

a) Network Connections: From the point of view of the network connections, we compare the different traces looking at the IP addresses to which the game connects. Games connect to servers when some specific states are reached in the game. This usually means that a result is registered into a global server for ranking purposes, or new contents are required and downloaded.

Tab. II reports the results for the network connection comparison for the three analysed games, where Human is the average number of IP addresses to which the game connects during the human play testing, MG is the number of IP addresses to which the game connects during the Monkey Gamer testing, Common is the average number of common IP addresses to which the game connects both during the human play testing and during the Monkey Gamer testing, and Coverage is the average percentage of IP addresses of the human player trace covered by the Monkey Gamer trace.

⁴Note that simulation times are much longer than the human players playing time, as Monkey Gamer is waiting after each input for the longest possible animation time, i.e., 2s, while the human players have much shorter and adaptive response time. Furthermore, the Monkey Gamer needs to wait for computations to be completed for each screen analysis step.

Game	Human	MG	Common	Coverage
TTTF	17.67	79	7.67	46.67%
BW	4.67	57	2.67	73.13%
AB	12.00	36	5.00	35.20%
Average	11.45	57.33	5.11	51.67%

TABLE II: Network connections: number of IP addresses to which the game connects.

Game	Human	MG	Common	Coverage
TTTF	6.00	9	6.00	100.00%
BW	8.33	11	8.00	95.83%
AB	18.33	22	9.33	50.39%
Average	10.89	14.00	7.78	82.07%

TABLE III: Game states visited during the tests.

Analysing the traces we observed a high variability for the IP addresses to which the game connects, even among the different human traces for a single game. At the same time, we observe that the Monkey Gamer covers on average more than half of the network connections generated by a human player, while exploring more than five times more potential network connections.

b) Game States: Game states are identified by the screen shown by the game. We compare here the game states met by the game during the Monkey Gamer testing and during the Human play testing.

Tab. III reports the results for the game state validation for the three analysed games, where Human is the number of game states visited by the game during the human play testing, MG is the number of game states visited by the game during the Monkey Gamer testing, Common is the average number of common states between the human traces and the Monkey Gamer trace, and Coverage is the average percentage of states from the human traces covered by the Monkey Gamer trace.

The Monkey Gamer covers a very high percentage of the states covered by the human players, while offering a higher exploration of the game-state space.

c) System Calls: System calls represent the interface between an app and the operating system. They are the way apps require services to the kernel. This may include access to data, creation and execution of processes, communication with kernel services (e.g., scheduler), etc. While comparing the human and the monkey gamer traces for system calls we are interested in “patterns”, i.e., sequences of system calls, since the presence of a similar pattern in the different traces means that the same macro-operation has been performed by the game in the different tests. We define system call patterns as sequences of at least two consecutive system calls. As the repetition of a macro-operation may result in a slightly different sequence of system calls, we consider two consecutive patterns as a single pattern, even if the two patterns are separated by a single different system call. For instance, for the system calls A , B , C , and D , the two sequences $\{A, A, B, C, A\}$ and $\{A, A, D, C, A\}$ are considered as the same pattern.

In order to objectively quantify the similarity between two

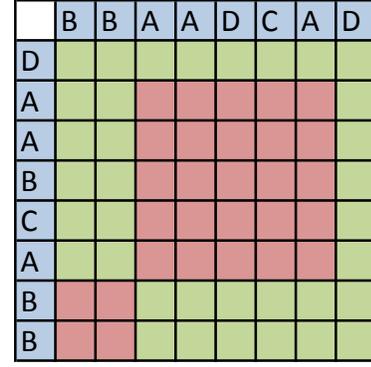


Fig. 6: Graphical representation of the similarity between two system call sequences: the area in red represents a similarity between the two sequences.

Chunk	Human	MG	Common	Similarity
TTTF-1	912.00	812	534.33	43.17%
TTTF-2	572.00	458	154.67	69.70%
TTTF-3	297.33	293	110.33	76.00%
BW-1	261.33	292	37.00	71.23%
BW-2	92.33	201	29.00	85.83%
BW-3	32.00	86	18.33	88.53%
AB-1	398.67	395	90.33	68.57%
AB-2	218.67	142	28.33	79.53%
AB-3	63.67	115	22.67	88.87%
Average	316.44	310.44	113.89	74.60%

TABLE IV: Validation results for the system calls pattern comparison.

different traces from the point of view of the system call sequences, we use the technique described in the work by Cornelissen and Moonen [16]. This technique produces a graphical representation of the similarity between two sequences⁵. An example is reported in Fig. 6, where the two compared sequences are reported on the x and y axis, respectively. The (dark) red area represents the intersection between common patterns, while the (light) green area represents the intersection between non-common parts of the sequences. We measure the similarity between two sequences as the percentage of the total area covered by intersection between the common patterns (45.31% in the shown example).

Collected traces contain hundreds of thousands of system calls. For each trace, we analyse three different chunks of 10^4 system calls, corresponding to an early, a middle, and a late game stage. Each trace chunk from the Monkey Gamer simulation is hence compared to the corresponding ones from the human play traces. Results are reported in Tab. IV, where Chunk is the ID of a given chunk, composed by the game acronym and by a sequence number (i.e., 1 for the early chunk, 2 for the middle chunk, and 3 for the late chunk), Human is the average number of distinct system calls made by the

⁵The technique is originally used to highlight self similarities in a single sequence by comparing a sequence to itself [16]. Here we use the technique to draw the same conclusions when comparing two different sequences.

game for the considered chunk while played by humans, MG is the number of distinct system calls made by the game for the considered chunk during the Monkey Gamer play test, Common is the number of system calls made by the game for the considered chunk during the human play testing covered in the corresponding chunk of the Monkey Gamer play testing trace, and Similarity is the similarity percentage computed as described above considering the system call patterns.

Despite the very high number of distinct system calls in the traces, Monkey Gamer is able to produce a test trace with high similarity to the one produced by the human players. Even if the overall rate of covered individual system calls is sometimes low, the similarity between traces in terms of patterns is high, meaning that the macro-activities performed by the game are generally the same in the Monkey Gamer traces compared to the human traces, and that the common system calls are used more often than the non-common ones. Furthermore, we can see that later stages of the game are characterised by higher similarity. Later stages present a lower amount of distinct system calls, among which, apparently, the common patterns dominate.

D. Validation Discussion

Considering the results obtained analysing the Monkey Gamer behaviour while testing the three selected games, and contrasting it with the human play testing, we can conclude that Monkey Gamer offers a good approximation of a human player while profiling games. This applies from the point of view of the number of network connections, the game states, and the patterns of system calls.

Apart from a good approximation of a human player, the Monkey Gamer also offers a higher coverage of network connections, game states, and system calls potentially performed by the game, offering a higher value for game testing. These are valuable inputs to several types of studies e.g., security analysis. For example, while profiling the AB game with the Monkey Gamer, a request for Superuser permissions was detected, which was not detected during the human play testing, and which may represent a security threat. A screenshot of the suspicious permission request is depicted in Fig. 7.

V. CONCLUSIONS

Mobile applications have a high and increasing penetration in everyday life, and gamification is being used to expand the reach of software applications in new settings. Games represent a considerable share of current application download and usage. Automatic profiling of game applications is needed for research and development purposes, and to check the efficiency, trustworthiness and reliability of the games. Towards these goals, this paper describes our effort in automatic profiling for Android games. Games are profiled to reach a wide variety of the system states, and to obtain records of the system calls and network traffic generated by the game. The game profiling process relies on the automatic and adaptive input generation, taking into account the interactive objects detected at each game state.

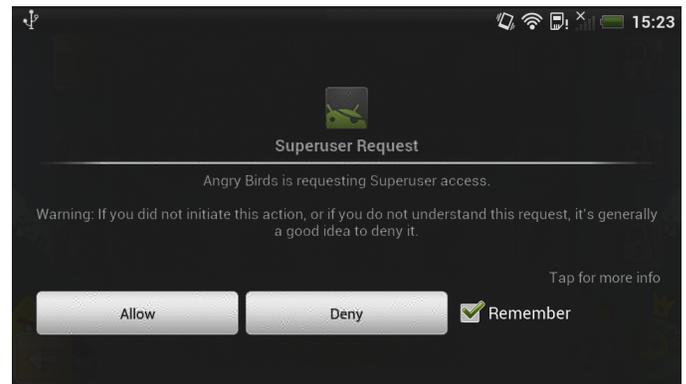


Fig. 7: Superuser permission request by *Angry Birds*.

Simulation results show that the obtained profiling for three of the most popular Android games is highly similar to the one obtained while using real human players, and guarantees a higher degree of exploration for potential game states, network traffic and system call generation.

This work began with a pre-study of the games market, and thereby offers a first step in the analysis of the game population, cataloguing games from the point of view of the user input process and of the game development process.

The analysis and results presented in this paper should be intended as a first tool validation, while, for a systematic evaluation of bugs and games reliability, more testing is needed. A further tool validation should include a comparison with other automatic tools, even if narrowed down to the specific game aspect, or to the specific game, targeted by the tool which is object to comparison. This aspect is left as future work.

Future work will also aim at extending automatic game profiling to include time-dependent games, as well as other kinds of inputs, like multitouch gestures (e.g., pinch, rotate, etc) or phone sensor inputs (e.g., gyroscope).

ACKNOWLEDGMENTS

This work has been supported by the Swedish national Graduate school in computer science (CUGS). The Authors thank Dr. Marcello Calisti for the insightful suggestion on the use of the Hough transformation for object detection.

REFERENCES

- [1] Manweiler, J., Agarwal, S., Zhang, M., Roy Choudhury, R., and Bahl, P. "Switchboard: a matchmaking system for multiplayer mobile games." *Proceedings of the 9th international conference on Mobile systems, applications, and services (MobiSys 2011)*. ACM, 2011.
- [2] Perez S., Tech Crunch, <http://techcrunch.com>. Cited 30 May 2013.
- [3] Play Store (previously Android Market), <http://play.google.com/store>. Cited 30 May 2013.
- [4] MonkeyRunner, http://developer.android.com/tools/help/monkeyrunner_concepts.html. Cited 30 May 2013.
- [5] Duda, R.O., and Hart, P.E., "Use of the Hough transformation to detect lines and curves in pictures." *Communications of the ACM* 15.1 (1972): 11-15.

- [6] Open Source Computer Vision Library, <http://www.opencv.org>. Cited 30 May 2013.
- [7] Dietrich, B., and Samarjit, C., "Power management using game state detection on android smartphones." *Proceeding of the 11th annual international conference on Mobile systems, applications, and services (MobiSys 2013)*, ACM, 2013.
- [8] Patro, A., Rayanchu, S., Griepentrog, M., Ma, Y., and Banerjee, S., "The anatomy of a large mobile massively multiplayer online game." *ACM SIGCOMM Computer Communication Review*, 42.4 (2012): 479-484.
- [9] Gilbert, P., Chun, B. G., Cox, L. P., and Jung, J., "Vision: Automated Security Validation of Mobile Apps at App Markets." *In Proceedings of the second international workshop on Mobile cloud computing and services*, ACM, 2011.
- [10] Gilbert, P., Chun, B. G., Cox, L., and Jung, J., "Automating Privacy Testing of Smartphone Applications." *Technical Report CS-2011-02*, Duke University, 2011.
- [11] Dai, S., Tongaonkar, A., Wang, X., Nucci, A., and Song, D., "Networkprofiler: Towards automatic fingerprinting of android apps." *In Proceedings of the 32nd IEEE International Conference on Computer Communications (INFOCOM)*, 2013, Chicago.
- [12] Anand, S., Naik, M., Harrold, M. J., and Yang, H., "Automated Concolic Testing of Smartphone Apps." *In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012.
- [13] Wei, X., Gomez, L., Neamtiu, I., and Faloutsos, M., "Profiledroid: Multi-Layer Profiling of Android Applications." *In Proceedings of the 18th annual international conference on Mobile computing and networking*, ACM, 2012.
- [14] Hu, C., and Iulian N., "Automating GUI Testing for Android Applications." *Proceedings of the 6th International Workshop on Automation of Software Test*, ACM, 2011.
- [15] Takala, T., Katara, M., and Harty, J., "Experiences of System-Level Model-Based GUI Testing of an Android Application." *Software Testing, Verification and Validation (ICST), Fourth International Conference on*, IEEE, 2011.
- [16] Cornelissen, B., and Moonen, L., "Visualizing Similarities in Execution Traces." *Proceedings of the 3rd Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, SERG, 2007.