An Anti-Fuzzing Approach for Android Apps

Chris Chao-Chun Cheng, Li Lin, Chen Shi, and Yong Guan

Iowa State University, Ames, Iowa, USA
guan@iastate.edu

Abstract. Extracting evidence pertaining to mobile apps is a key task in mobile device forensics. Since mobile apps can generate more than 19,000 files on a single device, it is time consuming and error prone to manually inspect all the files. Fuzzing tools that programmatically produce interactions with mobile apps are helpful when paired with sandbox environments to study their runtime forensic behavior and summarize patterns of evidentiary data in forensic investigations. However, the ability of fuzzing tools to improve the efficiency of mobile app forensic analyses has not been investigated.

This chapter describes AFuzzShield, an Android app shield that protects apps from being exercised by fuzzers. By analyzing the runtime information of mobile app interaction traces, AFuzzShield prevents real-world apps from being exercised by fuzzers and minimizes the overhead on human usage. A statistical model is employed to distinguish between fuzzer and human patterns; this eliminates the need to perform graphical user interface injections and ensures compatibility with apps with touchable/clickable graphical user interfaces. AFuzzShield verifies mobile app program coverage in situations where apps engage anti-fuzzing technologies. Specifically, it was applied to apps in AndroTest, a popular benchmark app dataset for testing fuzzers. The experimental results demonstrate that applying AFuzzShield significantly impacts mobile app program coverage in terms of reduced evidentiary data patterns.

Keywords: Android Apps · Evidentiary Data · Forensic Analysis · Anti-Fuzzing

1 Introduction

As of March 2023, around 2.67 million Android apps were available to the public [31]. Whereas a manual examination by a forensic practitioner can cover most usage scenarios of an individual app to understand the evidentiary data generated at runtime, the manual approach does not scale to handle the thousands of files generated by the numerous apps residing on a typical mobile device [5]. Automated fuzzing tools that programmatically generate interactions with Android apps provide an alternative. In fact, Android app fuzzers have been applied in security/privacy leakage analyses [12, 33] as well as in forensic analyses [38].

Android app fuzzers are categorized according to how they generate interaction events, namely, random-based [13, 22, 40], model-based [1, 3, 6, 16, 39] and

© IFIP International Federation for Information Processing 2023
Published by Springer Nature Switzerland AG 2023
G. Peterson and S. Shenoi (Eds.): DigitalForensics 2023, IFIP AICT 687, pp. 37–53, 2023.
https://doi.org/10.1007/978-3-031-42991-0_3

systematic [2, 3, 5, 23, 34]. Their common goal is to improve app code coverage in a limited time frame. However, while these fuzzers provide key insights on driving and running apps efficiently, they can be leveraged to discover app vulnerabilities and launch the corresponding attacks. To combat abuses, researchers have proposed anti-analysis techniques for Android devices [10, 17, 24, 26, 35]. The general idea is to perform software checks on certain system parameters to detect the sandbox environments in which fuzzers and dynamic analyzers are usually deployed. But the anti-analysis techniques can be disabled by automatically removing the conditional statements that check system parameters to detect sandbox environments [28].

At this time, no study has systematically analyzed how anti-fuzzing techniques impact the mobile app program coverage achieved by fuzzers. As a result, it is not possible to assess the reliability of using fuzzers to generate mobile app evidentiary data patterns in sandbox environments. Diao et al. [10] detect programming patterns in order to block the use of fuzzing tools. However, their approach, which is implemented and evaluated only for a proof-of-concept app, may hinder app functions by overlapping with the original graphical user interface (GUI) layouts, reducing its value as a real-world anti-fuzzing solution. Fuzzification [18], an anti-fuzzing tool, is preferred by developers. It profiles the frequencies of program paths visited by fuzzers and injects timing delays in less-frequently-used program paths to slow down the program while minimizing the impacts to normal use. However, due to Fuzzification's distinct environment, user behaviors and Linux command-line interface, it is unable to analyze the impacts of adopting Android app anti-fuzzing approaches to hinder forensic analyses.

An Android app anti-fuzzing technique must meet two requirements. First, it should introduce minimal overhead to real-world app users without inducing app malfunctions. Second, it should hinder fuzzers from exercising apps when they are triggered. Due to the lack of tools that meet these requirements, this research has developed AFuzzShield, an app shield that protects apps from being exercised by fuzzers. AFuzzShield is employed to analyze the reliability of Android app fuzzers that could be leveraged by forensic practitioners to generate evidentiary data patterns of mobile apps. AFuzzShield dynamically collects usage patterns such as clicks and swipes, and determines if the interaction events are triggered by human users or fuzzers. By injecting timing delays into app programs when fuzzer patterns are identified, AFuzzShield hinders fuzzer testing and prevents fuzzers from exercising apps. AFuzzShield also has the ability to profile fuzzer usage based entirely on runtime information. Since sandbox detection is unreliable and overwriting a GUI may negatively impact the original app functions, runtime pattern identification is more stable and less likely to crash the app.

This chapter presents several programming patterns for identifying existing fuzzers and demonstrates how the patterns are exploited in anti-fuzzing solutions. Experimental results involving more than 68 real-world apps in AndroTest [41] demonstrate that evidentiary data extraction from mobile apps using fuzzers is not reliable because fuzzing can be detected and mitigated.

2 Android App Fuzzing

Fuzzing is a software testing technique that automatically generates and injects inputs into software, helping detect vulnerabilities in tested programs more efficiently than human testers. Fuzzing tools that programmatically produce interactions with mobile apps such as tapping and swiping are useful when paired with sandbox environments to study their runtime forensic behavior and summarize patterns of evidentiary data. Android app fuzzers such as TaintDroid [12] and others [33, 38] have been used very effectively for dynamic program analysis, including generating evidentiary data and determining app vulnerabilities.

Android app fuzzers are categorized according to how they generate user interaction events. Random-based fuzzers [13, 22, 40] cover as many program branches as possible by randomly generating input events that trigger the app functions being called. Monkey [13], a fully-automated random-based fuzzer, randomly generates user interaction events such as swipes and clicks according to the configured probabilities, pushes them into the event queue and executes them as required.

Model-based fuzzers, unlike random-based fuzzers, require knowledge of runtime user interaction events that prevents them from generating redundant input events. During app runtime, a model-based fuzzer constructs GUI models by parsing the runtime information into a user interaction hierarchy that enables the fuzzer to employ a path finding algorithm to produce maximum code coverage with minimal inputs.

Systematic fuzzers typically instrument apps under test using program analysis techniques. By thoroughly analyzing the input events required to cover portions of an app program, a systematic fuzzer gains complete knowledge about the appropriate events needed for app testing. For example, after instrumenting an app, when certain activity transitions require a user to click a button twice, a systematic fuzzer can do so without attempting other useless combinations of inputs.

3 Methodology

AFuzzShield is a pattern-based anti-fuzzing solution. Unlike other approaches that extract system characteristics to detect sandbox environments [10, 17, 24, 26, 35] or modify the original GUI layouts of apps [10], AFuzzShield identifies fuzzers using their programming patterns. This section describes the AFuzzShield design framework, discusses the pattern differences between fuzzers and human users, and presents the AFuzzShield anti-fuzzing solution.

3.1 AFuzzShield Overview

Figure 1 presents an overview of AFuzzShield. AFuzzShield is designed as a third-party library for use in app development, where users instrument AFuzzShield

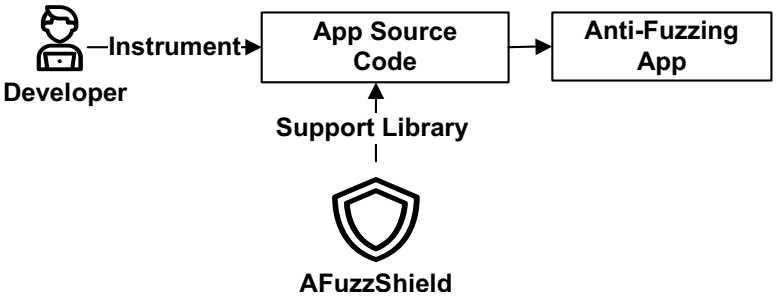

Fig. 1. AFuzzShield overview.

APIs in their implemented GUI callback methods. During app execution, AFuzzShield monitors the interaction traces and dynamically-adjusted pattern-based timing delays that correlate with the likelihood of fuzzer usage. This reduces the app code coverage using the same amount of time as when the app is exercised by fuzzers.

3.2 App Fuzzer Patterns

This section discusses the patterns of Monkey and other fuzzers.

Monkey Monkey [13] is a popular random-based, open-source fuzzer with fingerprinting features. Investigation of Monkey's source code revealed that its click and swipe event implementations are adequate to identify Monkey. Since clicks and swipes are fundamental GUI object operations, simply excluding them in Monkey's options results in significant coverage loss during software testing. Therefore, AFuzzShield can exploit the click and swipe patterns of Monkey to detect the use of fuzzers.

Three utility methods enable Monkey to implement randomness on top of uniform distributions:

- **randomPoint:** This method returns coordinates (x, y) where x is picked randomly from zero to the device width and y is picked randomly from zero to the device height.
- **randomVector:** This method returns a set of random values in the range –25 to 25 pixels.
- **randomWalk:** This method, upon being given a set of coordinates (x, y), randomly modifies their values in the range –25 to 25, but casts the results within the boundaries of the device display.

The three utility methods yield the following patterns for click and swipe events:

- **Click**: A click event comprises two consecutive touch events. The first touch event receives coordinates from `randomPoint` and the second touch event receives coordinates from `randomWalk` applied to the first set of coordinates.

– **Swipe**: A swipe event is extended from a click event by randomly adding one to nine movements between the first and second touch events of the click event. Each movement is the result of calling `randomWalk` with the previous set of coordinates, i.e., it moves randomly –25 to 25 pixels along the x and y axes, respectively.

The stability of the click and swipe events in fuzz testing enables AFuzzShield to use them as programming patterns that identify Monkey.

Other Fuzzers A stable feature of Android fuzzing tools is the set of coordinates corresponding to their generated click events. Specifically, investigations revealed that all existing open-source fuzzers [1–3, 6, 7, 11, 14, 16, 20, 22, 32] click the centers of GUI objects regardless of their exploration strategies.

Consider, for example, A3E [3], which relies on Robotium [29] to interact with an app being tested. When A3E picks an exploration strategy, it instructs Robotium to click the target GUI object. Robotium then computes the center of the target object on the screen and clicks the target. When computing the centers of odd lengths, some fuzzers like Robotium that do not round values yield different center coordinates from fuzzers that round values. To address the different implementations when one side of a GUI object has an even length, AFuzzShield picks the nearest point with an integer value as the center.

It is surprising that fuzzers have such explicit artificial patterns. The likely reason is that they click the centers of objects to avoid clicking the wrong targets when the object sizes are small. One exception is APE [14], which enables users to choose between Monkey's strategy or clicking the centers of objects using its designed strategy. However, this exception does not impact AFuzzShield because it considers both patterns.

3.3 Real-World Human Patterns

This research employed the Rico [9] dataset to identify the differences between fuzzer and human patterns in user interactions. The Rico dataset contains more than 9,300 app GUI layouts and human interaction traces generated by 11 real users. The human interaction traces include 52,456 effective click event coordinates and 13,377 swipe event traces.

Human patterns were learned by mapping click and swipe event coordinates to the corresponding topmost GUI object and normalizing the coordinates by transforming them linearly to a standard 2×2 square. Interestingly, the heatmap in Figure 2 reveals that the hottest area with the most click events is not near the center of a button. Clearly, human patterns do not follow common normal or uniform distributions. In fact, Figure 3 shows that, even when the two-dimensional plot is reduced to a one-dimensional plot for each coordinate, the normalized human clicking data does not have a clear bell shape. This means that very special human click patterns must be considered to combat fuzzing tools.

The swipe events generated by Monkey contained no more than nine consecutive movements and the scale of movement in any direction was bound by

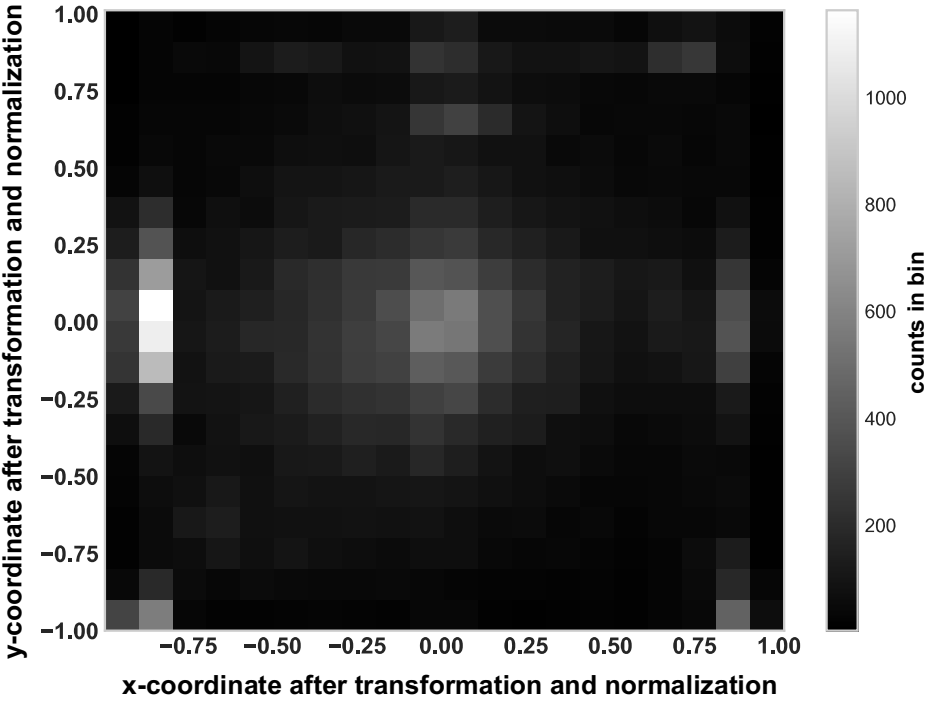

Fig. 2. Heatmap distributions of normalized human click coordinates on buttons.

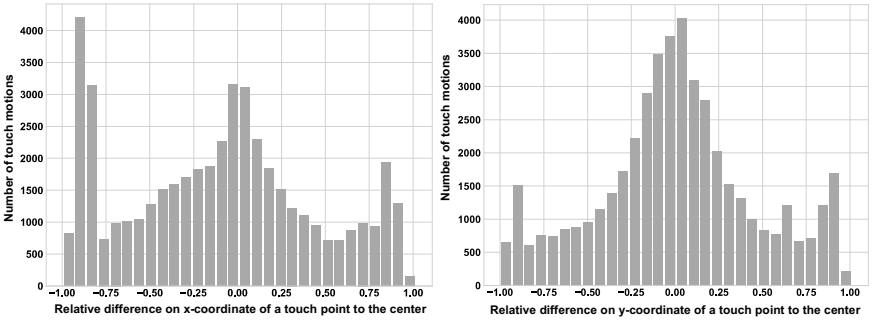

Fig. 3. Normalized click coordinate histograms (left: x-axis, right: y-axis).

25 pixels. However, only 56.67% of the Rico dataset traces were determined to have less than ten movements. Additionally, among the 373,635 movements in 13,377 real-world swipe events, just 55,187 (14.77%) along the x-axis or y-axis were determined to have scales larger than 25 pixels. Since the Rico dataset

contains such large numbers of outliers from Monkey's model, it is reasonable to designate outlier swipe events as being produced by human users.

3.4 Differentiating Fuzzer and Human Patterns

All the model-based fuzzers investigated clicked the exact centers of buttons in the interaction traces, which were explicitly different from human interaction patterns. In the Rico dataset, only eight out of thousands of human click events touched the button centers. Even when the ranges were extended to two pixels from the centers, the amount of human click events was less than 300. In fact, the probability of a human user touching the center of a GUI object is less than $300/46{,}694 = 0.6\%$ and the probability of a human user touching the center n times is $(\frac{0.6}{100})^n$, which is extremely small when $n \geq 3$. Therefore, if the center of a button is clicked consecutively more than three times, it is safe to assume that the events were created by a fuzzer instead of a human user.

Based on the Monkey patterns discussed above, human patterns can be distinguished from Monkey patterns using the Chi-square test to test the independence of click regions from click event data. The Chi-square test is selected because the Rico data indicates a human preference or bias towards click regions whereas Monkey views all click regions the same.

After experiments and analysis, it was decided to divide an entire normalized button GUI into three regions with equivalent rectangle areas and compare the difference between the expected frequency to the observed frequency in order to set up a hypothesis for testing. Specifically, the null hypothesis H_0 is that the events are produced by the Monkey fuzzer in that the click data has a uniform distribution on the two-dimensional plane. The alternative hypothesis H_1 is that the data does not have a uniform distribution.

For all the GUIs in a single app, if there are n touches, the theoretical frequency for a partition is $\frac{n}{3}$. Let k_1, k_2 and k_3 be the numbers of touches in the three regions. Then, under the null hypothesis H_0, if the statistic:

$$V = \frac{3}{n}\sum_{i=1}^{3}(k_i - \frac{n}{3})^2$$

is viewed as a random variable v, then v follows a Chi-square distribution with a degree of freedom of two (i.e., $v \sim \chi_2^2$). The corresponding p-value for this hypothesis test is computed as:

$$p = Prob(v \geq V)$$

where v has a χ_2^2 distribution. It is well known that p-values weight evidence against the null hypothesis. Therefore, the smaller the p-value, the stronger the evidence to reject H_0.

4 Evaluation

AFuzzShield was evaluated in a virtual machine environment created by AndroTest [41], a widely-adapted sandbox with 68 open-source real-world apps

for experimenting with Android app fuzzers [5, 11, 20, 25, 32]. Attempts made to debug and fix AndroTest's scripts for PUMA [16], A3E [3], Dynodroid [22], ACTEve [2] and GUIRipper [1] (PADAG) were successful. However, it was not possible to fix and run the script for SwiftHand [6].

The deployment of AFuzzShield required the manual identification and instrumentation of 632 user interaction callback methods in millions of lines of app code, after which the output Android Package Kit (APK) file was built. The impacts of real-world human user experiences were evaluated using the public Rico dataset [9]. Simulations were employed to evaluate AFuzzShield performance against human users, who received minimal impacts. Monkey and PADAG were executed on the original apps as well as on the apps with AFuzzShield deployed. Each fuzzer was executed for an hour and the line coverage results were collected using EMMA [30] every five minutes. Note that AndroTest employs a one-hour experiment time and five-minute data collection intervals. The same parameters were used to consistently reproduce the experimental environment.

4.1 Real-World App Performance

This section discusses the performance of AFuzzShield on real-world apps with human users and with Monkey and other fuzzers.

Human User Evaluation While it would be best to employ human user click coordinates on the evaluated apps, difficulty in obtaining institutional review board approval for human subjects during the research window forced the use of data from the Rico dataset. Therefore, the statistics discussed above were applied to random samples from the Rico dataset of sizes 18, 24, 30, 60 and 90 with 100 times random experiments without replacements, following which the p-values were computed.

Table 1 summarizes the Chi-square test results for the Rico data where, by convention, the significance level threshold for rejecting H_0 is set to $p \leq 0.05$. Note that identification using the p-values of the Chi-square test with more than 60 click events yields a low false negative rate (FNR), i.e., the error rate of misclassifying human users as Monkey. However, in practice, many more click events are required to achieve sufficient power for a significance level of 0.05. Therefore, to enhance classifier performance in real-world scenarios, especially those with lower numbers of clicks, a much larger p-value, such as $p < 0.1$, would indicate that the events are more likely to have been created by a fuzzing program.

Although the larger number of click events improves the false negative rate, it deteriorates fuzzer performance as more program paths are covered. Based on the evaluation results, in order to balance user experience while protecting apps from being exercised by fuzzers, a three-stage timing delay was injected into each app source function similar to the Fuzzification approach [18] of hindering fuzzers. Specifically, the runtime number of click event thresholds N_0, N_1, N_2 of 24, 30, 60, respectively, and P_0, P_1, P_2 of 0.05, 0.05, 0.1, respectively, were

Table 1. Performance of the Chi-square test as a classifier of human clicks.

Humans Clicks	FNR (Reject if $p < 0.05$)	FNR (Reject if $p < 0.1$)	Max p-Value
18	58%	40%	0.1573
24	38%	26%	0.1024
30	25%	20%	0.7400
60	2%	0%	0.1572
90	0%	0%	0.0160

applied in the three-stage timing delays. When N_i event data was collected, the computed p-value was compared against P_i to determine whether the injected timing delay should be updated. In the evaluation, timing delays of $T_0 = 0.2\,\text{s}$, $T_1 = 1\,\text{s}$ and $T_2 = T_{max} = 5\,\text{s}$ were employed for the three stages.

Existing Fuzzer Evaluation The experiments for evaluating AFuzzShield performance with existing fuzzers was designed based on their runtime patterns. The evaluation covered a random-based fuzzer (Monkey) and model-based fuzzers (PADAG). In addition to PADAG, open-source code of the other fuzzers was examined to confirm that they shared the same programming patterns observed in PADAG.

- **Monkey Evaluation:** A preliminary test was performed for one hour to obtain background knowledge about how much runtime information AFuzzShield obtained when deploying Monkey on an app. Because AFuzzShield relies only on app runtime information to evade fuzzer performance and the statistical model requires the number of click events to be higher than 60, apps that could not satisfy the requirement were filtered from the Monkey evaluation. For the remaining apps, line coverages upon running Monkey on them with and without AFuzzShield were compared in the evaluation.
  Table 2 shows the aforementioned preliminary results and results with AFuzzShield for 20 real-world apps. Note that Inst. denotes the number of instrumented user interfaces, Recv. denotes the number of events received per hour, LC_0 denotes the line coverage without AFuzzShield, LC_A denotes the line coverage with AFuzzShield, Delta denotes the absolute change of line coverage and R. Delta denotes the relative reduced line coverage.
  The results reveal that AFuzzShield effectively evades the performance of Monkey for 14 of the 20 real-world apps (highlighted in the table) and the best case demonstrates a reduction of line coverage from 59.85% to 43.41%. Although three apps have zero evasion and three apps have slightly elevated line coverages, AFuzzShield still protects most of the apps from being exercised by Monkey.
  Upon parsing the evaluation results, it was discovered that the more complex the app design, such as the number of user interface buttons, the more likely

Table 2. Monkey evaluation results.

App	Inst.	Recv.	LC_0	LC_A	Delta	R. Delta
A2DP Volume	17	158	42.44%	39.92%	**2.52%**	**5.94%**
AnyMemo	82	133	26.39%	22.56%	**3.83%**	**14.51%**
Baterrydog	5	107	62.47%	62.47%	0.00%	0.00%
BookCatalogue	68	147	33.39%	27.69%	**5.70%**	**17.07%**
Battery Circle	6	176	72.99%	73.92%	–0.93%	–1.27%
Alarm Clock	9	140	71.14%	66.97%	**4.16%**	**5.85%**
aCal	44	77	21.84%	16.15%	**5.69%**	**26.05%**
Yahtzee	8	64	59.97%	52.23%	**7.74%**	**12.91%**
CountdownTimer	4	198	71.14%	75.85%	–4.71%	–6.62%
Dialer2	15	546	37.71%	34.86%	**2.86%**	**7.58%**
MunchLife	4	215	72.45%	72.45%	0.00%	0.00%
MyExpenses	19	188	47.96%	40.52%	**7.44%**	**15.51%**
LearnMusicNotes	8	537	59.85%	43.41%	**16.44%**	**27.47%**
passwordmanager	17	123	9.00%	8.22%	**0.78%**	**8.67%**
RandomMusicPlay	6	347	78.19%	77.72%	**0.48%**	**0.61%**
SoundBoard	2	82	46.81%	46.81%	0.00%	0.00%
SyncMyPix	15	125	21.57%	21.87%	–0.30%	–1.39%
TippyTipper	20	64	82.05%	79.01%	**3.04%**	**3.71%**
WeightChart	3	127	52.52%	51.11%	**1.41%**	**2.68%**
WhoHasMyStuff	8	111	74.13%	69.43%	**4.71%**	**6.35%**

that anti-fuzzing techniques reduce app program coverage, contributing to the reduced reliability of Android app fuzzers in generating evidentiary data. Since AFuzzShield requires runtime data to determine whether or not operations are due to fuzzers, if an app is designed in a straightforward manner and has very few branches or event handlers to be triggered, it is highly likely that fuzzers would have explored many programs in the app before being effectively evaded by AFuzzShield. For example, Table 2 shows that apps receiving non-positive impacts from AFuzzShield have only six instrumented GUI objects on average compared with apps effectively protected by AFuzzShield that have more than 20 instrumented GUI objects on average.

– **Other Fuzzer Evaluation:** PADAG was deployed on over 68 apps from AndroTest with and without AFuzzShield during the performance evaluation. Tables 3 and 4 show the PADAG evaluation results – the highlighted results indicate exercising apps for which fuzzers were detected successfully by AFuzzShield. Note that no available GUI objects to monitor were discovered in 19 out of 68 apps, 22 apps with clickable GUIs received no click events and three apps crashed after launching. These app evaluation results were classified as ineffective because AFuzzShield exhibited limitations in applying its knowledge to protect the apps from fuzzing.

  The results in Tables 3 and 4 reveal that, in the best case, AFuzzShield reduced 35.33% of app code coverage on average, down from the original

Table 3. PADAG evaluation results.

App Name	Inst.	LoS	LC_0	LC_A	Delta	R. Delta
aCal	44	45,161	9.62%	4.08%	**5.55%**	**57.69%**
Manpages	1	385	43.39%	48.85%	–5.47%	–12.61%
Wordpress	42	10,100	2.30%	1.45%	**0.85%**	**36.96%**
Translate	3	799	25.10%	20.35%	**4.74%**	**18.88%**
LearnMusicNotes	8	1,114	23.38%	17.60%	**5.79%**	**24.76%**
Jamendo	12	4,430	6.13%	6.12%	**0.01%**	**0.16%**
TippyTipper	20	2,623	31.33%	10.21%	**21.12%**	**67.41%**
SyncMyPix	15	10,431	9.91%	5.58%	**4.34%**	**43.79%**
BookCatalogue	68	27,235	3.64%	3.87%	–0.22%	–6.04%
AnyMemo	82	25,824	5.79%	4.81%	**0.98%**	**16.93%**
Dialer2	15	2,057	27.78%	21.74%	**6.05%**	**21.78%**
Divide&Conquer	2	814	42.25%	37.46%	**4.79%**	**11.34%**
QuickSettings	2	2,934	22.35%	23.65%	–1.29%	–5.77%
AndroidomaticK	6	1,307	24.89%	18.18%	**6.72%**	**27.00%**
K-9Mail	41	22,208	3.92%	3.97%	–0.05%	–1.28%
Blokish	1	93	29.54%	29.71%	–0.17%	–0.58%
MyExpenses	19	8,058	21.08%	16.36%	**4.72%**	**22.39%**
A2DP Volume	17	7,040	17.17%	10.27%	**6.90%**	**40.19%**
AardDictionary	4	2,197	14.67%	9.13%	**5.53%**	**37.70%**
RandomMusicPlay	6	1,053	32.88%	21.39%	**11.48%**	**34.91%**
Multi SNS	9	828	24.60%	9.68%	**14.92%**	**60.65%**
Ringdroid	6	2,928	4.12%	4.04%	**0.08%**	**1.94%**
Yahtzee	8	1,349	19.19%	5.95%	**13.24%**	**68.99%**
Baterrydog	5	985	20.51%	9.64%	**10.87%**	**53.00%**
SoundBoard	2	99	30.75%	26.62%	**4.13%**	**13.43%**
Nectroid	5	2,536	24.35%	20.85%	**3.49%**	**14.33%**
Alarm Clock	9	5,765	22.59%	19.95%	**2.64%**	**11.69%**
HotDeath	8	3,902	11.12%	9.42%	**1.70%**	**15.29%**
World Clock	4	1,242	53.03%	17.69%	**35.33%**	**66.62%**
AnyCut	3	436	29.50%	29.67%	–0.17%	–0.58%
MunchLife	4	506	39.80%	30.02%	**9.78%**	**24.57%**
aGrep	6	928	2.88%	11.16%	–8.28%	–287.5%
Mileage	13	4,628	12.88%	11.74%	**1.13%**	**8.77%**
LolcatBuilder	5	646	13.74%	11.16%	**2.58%**	**18.78%**
ImportContacts	2	1139	19.84%	16.43%	**3.41%**	**17.16%**
Battery Circle	6	739	46.51%	44.43%	**2.09%**	**4.49%**
WhoHasMyStuff	8	1,555	37.78%	30.29%	**7.48%**	**19.80%**
Photostream	4	1,375	10.26%	9.83%	**0.43%**	**4.19%**
SpriteMethodTest	1	1,018	15.14%	14.76%	**0.38%**	**2.51%**
PasswordMakerPro	4	1,535	25.17%	17.09%	**8.09%**	**53.33%**
myLock	3	885	17.21%	17.95%	–0.74%	–4.30%
aagtl	4	11,724	8.36%	8.01%	**0.35%**	**4.19%**

Table 4. PADAG evaluation results (continued).

App Name	Inst.	LoS	LC_0	LC_A	Delta	R. Delta
FileExplorer	1	126	35.59%	37.58%	–2.00%	–5.62%
LockPatternGen	2	669	37.09%	29.42%	**7.66%**	**20.65%**
CountdownTimer	4	1,415	43.90%	19.80%	**24.09%**	**54.87%**
HNDroid	1	1,038	5.22%	5.17%	**0.05%**	**0.96%**
WeightChart	3	23,67	18.92%	18.05%	**0.87%**	**4.60%**
MiniNoteViewer	21	3,673	11.49%	2.29%	**9.20%**	**80.07%**
passwordmanager	17	38,104	3.46%	0.70%	**2.76%**	**79.77%**

fuzzing result of 53.03%. More than half of the apps were effectively protected by AFuzzShield from being exercised by PADAG. While AFuzzShield effectively protects most apps, there are four instances where app fuzzing results became slightly better when AFuzzShield was deployed. Investigations of the results revealed the reason for the anomaly. Specifically, Dynodroid and GuiRipper restarted the Android emulator after running each trace and erased AFuzzShield intermediate data (i.e., logging traces and accumulated statistics) stored on the SD card. AFuzzShield's re-monitoring and re-computing the patterns contributed to the performance deterioration.

4.2 Discussion

While AFuzzShield enables analyses of the reliability of Android app fuzzers in generating app evidentiary data at runtime, certain technical limitations exist in the underlying technique. First, AFuzzShield performance depends on the original GUI layout design. If an app GUI layout is too simple and does not have enough GUI elements that can help AFuzzShield monitor runtime interaction patterns, AFuzzShield cannot effectively reduce fuzzer performance. Second, AFuzzShield does not consider situations where the entire device storage is frequently restored during runtime, resulting in performance deterioration. To address this issue, future research will attempt to force AFuzzShield to sync with a network to permanently store runtime data.

Another limitation comes from the best-effort matching of apps and human data in the evaluations. Without an appropriate dataset containing real-world apps with source code and corresponding human user interaction traces, it was possible only to derive the statistic model from Rico's relative interaction user interface data and apply the model to the AndroTest apps. This limitation could be addressed if the apps used to create the Rico dataset were open-source to support AFuzzShield deployment or data pertaining to human user traces could be collected for AndroTest apps.

5 Related Work

Conventional Android app testing is event-driven and GUI-based, which is different from the scope of this research. Therefore, the related work discussion focuses on Android app fuzzing techniques.

5.1 Android App Fuzzing Techniques

AndroTest [41] compares and summarizes existing Android fuzzing studies prior to 2016. Fuzzers are categorized into three groups by their exploration strategies, random-based [13, 22, 40], model-based [1, 3, 6, 16, 39] and systematic [2, 3, 5, 23, 34]. Even fuzzers with similar exploration strategies have their own minimization algorithms that reduce running time while maximizing code coverage. Sapienz [25] stands out from other fuzzers in that it combines the three exploration strategies and proposes to exploit a genetic algorithm to minimize test sequences. DetReduce [7] incorporates a minimization algorithm that reduces the Android GUI testing suites generated by existing fuzzers. APE [14] creates its initial GUI model using runtime information that produces finer granularity models than Stoat [32]. TimeMachine [11], which outperforms Sapienz and Stoat, records and explores GUI state via virtual machine snapshots instead of the traditional GUI models. In order to improve fuzzing performance, VET [37] attempts to detect and drive fuzzer user interaction automation to avoid tarpits such as looping in login and cancel buttons. In contrast, TOLLER [36] extracts user interaction information by accessing app runtime memory instead of depending on Android system services.

5.2 Android App Anti-Analysis Techniques

Existing anti-analysis techniques do not specifically protect Android apps from fuzzers; instead, they generally attempt to evade program analysis tools deployed on Android apps. Lim et al. [21] divide anti-analysis techniques as static or dynamic. Static approaches include obfuscation [15], repacking app code [19] and verifying the integrity of executable files [4]. While static approaches are designed to protect apps at the code level, dynamic techniques focus on detecting sandbox environments (e.g. Android emulators) on which most fuzzers depend. Specifically, dynamic approaches fingerprint the data patterns of sandbox artifacts and match them at runtime. Most approaches [10, 17, 24, 26, 35] distinguish sandbox environments by checking device identifiers. Other characteristics include the cumulative distribution function of intervals between sensor events [26], system file content [17], consistent Android versions [24], frames per second [35] and swiping trajectories [10].

The work of Diao et al. [10] is the closest to AFuzzShield. It leverages several interesting ideas beyond system profiling, including swiping trajectories, phishing activity and invisible user interaction traps, to distinguish human users from fuzzers. However, some of these ideas are difficult to implement with real-world apps because they require sufficient user data such as swiping trajectories or they

may introduce malfunctions in the original apps due to invisible user interactions. A key difference between the work of Diao and colleagues and AFuzzShield is the targeting and evaluation scales that are unique to AFuzzShield. Additionally, AFuzzShield is actually implemented on real-world apps and evaluated against existing Android app fuzzers.

Costamagna et al. [8] determine sandbox environments via usage profiles such as contact list information and installed apps instead of system profiles. However, usage profiles in a sandbox environment are usually empty or are constructed randomly, which are distinct from user device profiles. To address these issues in dynamic approaches, Harvester [28] automatically replaces every conditional constraint related to system characteristics with a Boolean value controlled by the user space and, therefore, covers the protected program statements. AFuzzShield stands out from existing approaches in that it fingerprints the patterns of Android app fuzzers instead of sandbox environments.

6 Conclusions

Android app fuzzers can reduce the manual effort required in mobile device forensics by improving the efficiency of generating mobile app evidentiary data in runtime by exercising apps. However, research has not investigated the reliability and app program coverage when anti-fuzzing techniques are employed. The AFuzzShield anti-fuzzing solution presented in this chapter helps understand the impacts of anti-fuzzing techniques on app programs. Statistical differences identified by comparing the programming patterns shared by Android app fuzzers and the interaction traces collected from human users were leveraged for this purpose. The evaluation results demonstrate that 70% of the real-world apps in AndroTest can be hindered successfully by anti-fuzzing techniques. Additionally, the more complex the app GUI, the lower the program coverage obtained by a fuzzer.

Future research will focus on extending AFuzzShield to cover more reliability analysis cases of fuzzers by employing a larger app database than AndroTest that contains more complex GUI designs that are closer to real-world apps as well as more real-world human user interaction traces corresponding to the archived apps.

Acknowledgment This research was partially supported by the National Institute of Standards and Technology (NIST) CSAFE under Cooperative Agreement no. 70NANB20H019, by the National Science Foundation under Grant nos. CNS 1527579, CNS 1619201, CNS 1730275, DEB 1924178 and ECCS 2030249, and by the Boeing Company.

References

1. D. Amalfitano, A. Fasolino, P. Tramontana, S. De Carmine and A. Memon, Using GUI ripping for automated testing of Android applications, *Proceedings of*

the Twenty-Seventh IEEE/ACM International Conference on Automated Software Engineering, pp. 258–261, 2012.
2. S. Anand, M. Naik, M. Harrold and H. Yang, Automated concolic testing of smartphone apps, *Proceedings of the Twentieth ACM SIGSOFT International Symposium on the Foundations of Software*, article no. 59, 2012.
3. T. Azim and I. Neamtiu, Targeted and depth-first exploration for systematic testing of Android apps, *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 641–660, 2013.
4. K. Chen, Y. Zhang and P. Liu, Leveraging information asymmetry to transform Android apps into self-defending code against repackaging attacks, *IEEE Transactions on Mobile Computing*, vol. 17(8), pp. 1879–1893, 2018.
5. C. Cheng, C. Shi, N. Gong and Y. Guan, EviHunter: Identifying digital evidence in the permanent storage of Android devices via static analysis, *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp. 1338–1350, 2018.
6. W. Choi, G. Necula and K. Sen, Guided GUI testing of Android apps with minimal restart and approximate learning, *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 623–640, 2013.
7. W. Choi, K. Sen, G. Necula and W. Wang, DetReduce: Minimizing Android GUI test suites for regression testing, *Proceedings of the Fortieth IEEE/ACM International Conference on Software Engineering*, pp. 445–455, 2018.
8. V. Costamagna, C. Zheng and H. Huang, Identifying and evading Android sandbox through usage-profile based fingerprints, *Proceedings of the First Workshop on Radical and Experiential Security*, pp. 17–23, 2018.
9. B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols and R. Kumar, Rico: A mobile app dataset for building data-driven design applications, *Proceedings of the Thirtieth Annual ACM Symposium on User Interface Software and Technology*, pp. 845–854, 2017.
10. W. Diao, X. Liu, Z. Li and K. Zhang, Evading Android runtime analysis through detecting programmed interactions, *Proceedings of the Ninth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pp. 159–164, 2016.
11. Z. Dong, M. Bohme, L. Cojocaru and A. Roychoudhury, Time-travel testing of Android apps, *Proceedings of the Forty-Second IEEE/ACM International Conference on Software Engineering*, pp. 481–492, 2020.
12. W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, L. Cox, J. Jung, P. McDaniel and A. Sheth, TaintDroid: An information-flow tracking system for realtime privacy monitoring of smartphones, *ACM Transactions on Computer Systems*, vol. 32(3), article no. 5, 2014.
13. Google Developers, UI/Application Exerciser Monkey, Mountain View, California (`developer.android.com/studio/test/other-testing-tools/monkey`), 2022.
14. T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu and Z. Su, Practical GUI testing of Android applications via model abstraction and refinement, *Proceedings of the Forty-First IEEE/ACM International Conference on Software Engineering*, pp. 269–280, 2019.
15. Guardsquare, ProGuard: The industry-leading Java optimizer for Android apps, Leuven, Belgium (`www.guardsquare.com/proguard`), 2023.
16. S. Hao, B. Liu, S. Nath, W. Halfond and R. Govindan, PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps, *Proceedings of the*

Twelfth Annual International Conference on Mobile Systems, Applications and Services, pp. 204–217, 2014.
17. Y. Jing, Z. Zhao, G. Ahn and H. Hu, Morpheus: Automatically generating heuristics to detect Android emulators, *Proceedings of the Thirtieth Annual Computer Security Applications Conference*, pp. 216–225, 2014.
18. J. Jung, H. Hu, D. Solodukhin, D. Pagan, K. Lee and T. Kim, Fuzzification: Anti-fuzzing techniques, *Proceedings of the Twenty-Eighth USENIX Security Symposium*, pp. 1913–1930, 2019.
19. L. Li, D. Li, T. Bissyande, 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(6), pp. 1269–1284, 2017.
20. Y. Li, Z. Yang, Y. Guo and X. Chen, Humanoid: A deep-learning-based approach to automated black-box Android app testing, *Proceedings of the Thirty-Fourth IEEE/ACM International Conference on Automated Software Engineering*, pp. 1070–1073, 2019.
21. J. Lim, Y. Shin, S. Lee, K. Kim and J. Yi, Survey of dynamic anti-analysis schemes for mobile malware, *Journal of Wireless Mobile Networks, Ubiquitous Computing and Dependable Applications*, vol. 9(3), pp. 39–49, 2018.
22. A. Machiry, R. Tahiliani and M. Naik, Dynodroid: An input generation system for Android apps, *Proceedings of the Ninth Joint Meeting on Foundations of Software Engineering*, pp. 224–234, 2013.
23. R. Mahmood, N. Mirzaei and S. Malek, EvoDroid: Segmented evolutionary testing of Android apps, *Proceedings of the Twenty-Second ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 599–609, 2014.
24. D. Maier, M. Protsenko and T. Muller, A game of droid and mouse: The threat of split-personality malware on Android, *Computers and Security*, vol. 54, pp. 2–15, 2015.
25. K. Mao, M. Harman and Y. Jia, Sapienz: Multi-objective automated testing for Android applications, *Proceedings of the Twenty-Fifth International Symposium on Software Testing and Analysis*, pp. 94–105, 2016.
26. T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis and S. Ioannidis, Rage against the virtual machine: Hindering dynamic analysis of Android malware, *Proceedings of the Seventh European Workshop on System Security*, 2014.
27. J. Qin, H. Zhang, S. Wang, Z. Geng and T. Chen, Acteve++: An improved Android application automatic tester based on Acteve, *IEEE Access*, vol. 7, pp. 31358–31363, 2019.
28. S. Rasthofer, S. Arzt, M. Miltenberger and E. Bodden, Harvesting runtime values in Android applications that feature anti-analysis techniques, *Proceedings of the Twenty-Third Annual Network and Distributed System Security Symposium*, 2016.
29. Robotiumtech, Robotium: User scenario testing for Android, GitHub (`github.com/RobotiumTech/robotium`), 2023.
30. V. Roubtsov, EMMA: A Free Java Code Coverage Tool (`emma.sourceforge.net`), 2006.
31. Statista, Number of available applications in the Google Play Store from December 2009 to March 2023, Hamburg, Germany (`bit.ly/30fsg6W`), 2022.
32. T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu and Z. Su, Guided stochastic model based GUI testing of Android apps, *Proceedings of the Eleventh Joint Meeting on Foundations of Software Engineering*, pp. 245–256, 2017.

33. M. Sun, T. Wei and J. Lui, TaintART: A practical multi-level information flow tracking system for Android RunTime, *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp. 331–342, 2016.
34. H. van der Merwe, B. van der Merwe and W. Visser, Verifying Android applications using Java PathFinder, *ACM SIGSOFT Software Engineering Notes*, vol. 37(6), 2012.
35. T. Vidas and N. Christin, Evading Android runtime analysis via sandbox detection, *Proceedings of the Ninth ACM Symposium on Information, Computer and Communications Security*, pp. 447–456, 2014.
36. W. Wang, W. Lam and T. Xie, An infrastructure approach to improving effectiveness of Android UI testing tools, *Proceedings of the Thirtieth ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 165–176, 2021.
37. W. Wang, W. Yang, T. Xu and T. Xie, VET: Identifying and avoiding UI exploration tarpits, *Proceedings of the Twenth-Ninth ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 83–94, 2021.
38. Z. Xu, C. Shi, C. Cheng, N. Gong and Y. Guan, A dynamic taint analysis tool for Android app forensics, *Proceedings of the IEEE Security and Privacy Workshops*, pp. 160–169, 2018.
39. W. Yang, M. Prasad and T. Xie, A grey-box approach for automated GUI model generation of mobile applications, *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, pp. 250–265, 2013.
40. H. Ye, S. Cheng, L. Zhang and F. Jiang, DroidFuzzer: Fuzzing Android apps with intent-filter tag, *Proceedings of International Conference on Advances in Mobile Computing and Multimedia*, pp. 68–74, 2013.
41. X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, T. Xie, Automated test input generation for Android: Are we really there yet in an industrial case? *Proceedings of the Twenty-Fourth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 987–992, 2016.