A-SIT

Zentrum für sichere Informationstechnologie - Austria

State-of-the-Art Black-Box-UI-Testing von Android-Anwendungen

State-of-the-Art Black-Box-UI-Testing von Android-Anwendungen

Autor:
Florian Draschbacher:
florian.draschbacher@iaik.tugraz.at

Datum: 23.08.2023

Abstract/Zusammenfassung:
Automatisiertes Black-Box-UI-Testing hat eine Vielzahl von Anwendungsfällen in der dynamischen Software-Analyse, in der Erforschung von Malware, sowie in der automatisierten Generierung von Test-Cases. Obwohl schon kurz nach der Veröffentlichung der ersten Android-Version erste Tools für das Black-Box-UI-Testing von Android-Anwendungen zur Verfügung standen, litten diese für lange Zeit unter erheblichen Einschränkungen der praktischen Nutzbarkeit. Bis heute kommen in der Praxis trotz zahlreicher neuer Ansätze meist Dumb-Fuzzing-Ansätze zum Einsatz, bei denen Anwendungen lediglich mit zufälligem User-Input gespeist werden.

Im Rahmen dieses Projektes wurde diese gängige Praxis auf ihre Sinnhaftigkeit geprüft. Dazu wurden in einer ausführliche Literatur-Recherche in einem ersten Schritt vielversprechende neue Ansätze erfasst. Anschließend wurden diese Ansätze anhand eines Test-Sets von Real-World-Anwendungen verglichen. Genauer analysiert wurden außerdem die verschiedenen Möglichkeiten zur Erfassung der Coverage von UI-Exploration-Tools.

Inhalt

1. Einleitung 2
2. Hintergrund 2
2.1. Android 2
2.2. Android-Apps 3
2.3. APK-Dateien 3
3. Dynamisches Testen im App-Kontext 3
4. Dynamisches Testen unter Android 4
4.1. UI-Exploration 5
4.2. Coverage-Erfassung 6
5. Evaluierung aktueller UI-Exploration-Ansätze 7
5.1. Getestete Tools 7
5.2. Datensatz 7
5.3. Test-Umgebung 7
5.4. Ergebnisse 7
6. Zusammenfassung 8
Referenzen 8

1. Einleitung

Im Rahmen der Sicherheitsanalyse von Android-Anwendungen muss oft die Frage beantwortet werden, ob eine Anwendung bösartiges Verhalten zeigt (also etwa Nutzerdaten stiehlt), oder anfällig für Angriffe ist (weil eine Sicherheitslücke vorliegt). Um hier Aussagen treffen zu können, stehen vor allem zwei verschiedene Ansätze zur Verfügung.

Bei der statischen Analyse wird nur das Installationspaket einer App untersucht, um Rückschlüsse auf ihr Laufzeit-Verhalten zu ziehen. Dazu wird in der Regel anhand des Programmcodes ein Graph erstellt, der Aufschluss über den Kontrollfluss und Datenabhängigkeiten geben soll. Obwohl dieser Ansatz für einige Anwendungsfälle gut funktioniert, erlaubt er besonders auf der Android-Plattform keine zuverlässigen Rückschlüsse über die Bösartigkeit oder Anfälligkeit einer App. Anders als etwa die Konkurrenz-Plattform iOS erlaubt Android Apps zur Laufzeit das Laden zusätzlicher Code-Komponenten, die nicht im Installationspaket enthalten sind. Es kann also ohne eine Programmausführung der volle Umfang des App-Codes nicht erfasst werden.

Abhilfe schafft hier der Ansatz der dynamischen Analyse. Sie erfasst das Laufzeitverhalten einer App, indem diese tatsächlich ausgeführt wird. Entscheidend für eine aussagekräftige Analyse ist bei diesem Verfahren, dass eine hohe Coverage erreicht wird, dass also ein großer Anteil des Programmcodes tatsächlich ausgeführt und damit analysiert werden konnte. Es ergeben sich hier zwei entscheidende Problemstellungen der dynamischen Analyse: Wie kann eine hohe Coverage erreicht werden? Wie kann die Coverage verlässlich erfasst werden?

Zur Erreichung einer hohen Coverage ist in der Regel die Qualität der UI-Exploration ausschlaggebend, oder genauer, welche Methode zum Einsatz kommt, um anhand des aktuellen App-Bildschirms zu entscheiden, wie die Benutzerfläche der App weiter bedient werden soll, um Funktionalität der App zu erreichen, die bisher noch nicht getestet wurde. In der Literatur wird hier bei vielen Projekten noch lediglich zufällig entschieden, und argumentiert, dass sich so die besten Ergebnisse erzielen lassen. Diese Behauptung haben wir in diesem Bericht detaillierter überprüft.

Zur Erfassung der Coverage stehen im Moment für Android zwar mehrere Möglichkeiten zur Verfügung, jedoch gibt es keine präzise Lösung, die keine Änderungen des zu analysierenden Android-Pakets erfordert. Jede solche Modifikation kann zu Fehlfunktionen führen, die die Analyse beeinträchtigen. Im Rahmen des hier dokumentierten Projektes wurden auch Überlegungen angestellt, wie diese Probleme in der Coverage-Erfassung umgangen werden können.

2. Hintergrund

In diesem Abschnitt sollen jene Technologien erklärt werden, die für das weitere Verständnis der späteren Ausführungen notwendig sind.

2.1. Android

Android ist das populärste Betriebssystem für Mobilgeräte wie Smartphones und Tablets. Das System wird im Rahmen des Android Open Source Project (AOSP) von einigen Geräte-Herstellern unter Federführung von Google entwickelt. Aus technischer Sicht handelt es sich um ein Betriebssystem auf Basis eines Linux-Kernels. Die Userspace-Komponenten weichen jedoch wesentlich von denen jeder

anderen Linux-Distribution ab. Sie wurden speziell für die Anforderungen von Mobilegeräten neu entwickelt.

2.2. Android-Apps

Android erlaubt Nutzern die Ausführung von beliebigen Anwendungen von Drittanbietern. Google stellt ein Software-Development-Kit (SDK) zur Verfügung gestellt, mithilfe dessen Entwickler in den Programmiersprachen Java, Kotlin, C und C++ Anwendungen schreiben können. Nach dem Kompilieren können Programme in eine Android Package (APK) Datei verpackt und so auf einem Android-Gerät installiert werden.

2.3. APK-Dateien

Das Android-System erlaubt die Installation von Anwendungen einzig im APK-Format. Dabei handelt es sich im Wesentlichen um ein ZIP-Archiv mit klar definierter Struktur. Zentrale Elemente innerhalb dieser Struktur mit Relevanz für dieses Projekt sind:

Das Applikations-Manifest

Eine vom App-Entwickler erstellte XML-Datei, die als „Vertrag" zwischen App und Android-System betrachtet werden kann. Hier spezifiziert der Entwickler unter anderem die Permissions und Features, die die App zur Laufzeit benötigt, und gibt bekannt, welche Funktionalität die App dem Betriebssystem bzw. anderen Apps am System zur Verfügung stellt. Die häufigsten exportierten Komponenten sind hier Activities (UI-Bildschirme), ContentProvider (geregelter Zugriff auf Daten der App) oder Services (Hintergrundfunktionalität).

DEX-Dateien

Dalvik Executable (DEX)-Dateien enthalten den kompilierten Programmcode, der aus Java oder Kotlin kompiliert wurde. Im Rahmen der Kompilierung wird dieser in Dalvik Bytecode transformiert. Dabei handelt es sich um eine Intermediate Representation, die die plattformunabhängige Ausführung des Codes am Gerät erleichtert. Erwähnenswert ist in diesem Zusammenhang, dass im Dalvik-Bytecode entscheidende Highlevel-Informationen zum ursprünglichen Programmcode erhalten bleibt. Im Unterschied zu Programmen, die im Maschinencode vorliegen, kann hier also die implementierte Funktionalität relativ einfach rekonstruiert und in weiterer Folge verändert werden.

SO-Dateien (Native Bibliotheken)

Entwickler können Teile einer App in nativen Programmiersprachen wie C und C++ entwickeln. Im APK-Paket werden diese dann als ELF Shared Object (Dateiendung .so) abgelegt. Mittels Java Native Interface (JNI) können sie aus Java-Code heraus aufgerufen werden.

Ressourcen

Innerhalb eines APK-Pakets werden alle Ressourcen in einem zentralen Index-File (resources.arsc) eingetragen. Value-Daten, (Strings, Integers, Styles, …) werden direkt im Index hinterlegt, während alle restlichen Daten (UI-Layouts, Bitmaps, Konfigurationen, …) in separaten Dateien im APK-Paket abgelegt werden.

3. Dynamisches Testen im App-Kontext

Dynamisches Testen wird nicht nur in der Sicherheitsanalyse verwendet, wo Aussagen über die Sicherheit (Anfälligkeit bzw. Bösartigkeit) einer App getroffen werden sollen. Die Technologie wird auch von App-Entwicklern verwendet, um Probleme zu finden, die durch Corner-Cases (etwa unvorhergesehene

Eingabekombinationen oder abwegige Navigation der Benutzeroberfläche) entstehen und zu Fehlverhalten oder Abstürzen führen.

Während sich das dynamische Testen einzelner weniger Anwendungen noch manuell erledigen lässt, wird ab einer gewissen Menge an Applikationen eine automatisierte Test-Prozedur notwendig. Steht hierfür eine Beschreibung der genauen App-Funktionalität (etwa durch den Quellcode der App) zur Verfügung, wird von White-Box-Testing gesprochen. Wenn allerdings vorab keine Informationen über die App-Funktionalität (oder die Struktur der Benutzeroberfläche) zur Verfügung stehen, spricht man vom Black-Box-Testing. Hier können Test-Prozeduren auf keinen Fall im Vorfeld definiert werden (etwa indem in definierten Intervallen bestimmte Elemente der Benutzeroberfläche betätigt werden).

Der grundsätzliche Prozess des automatisierten dynamischen Testens besteht aus einer stetigen Wiederholung der immer selben drei Schritte. In jeder Iteration wird zunächst entschieden, welche Interaktion mit der Benutzeroberfläche der App durchgeführt werden soll. Diese wird dann ausgeführt. Danach wird evaluiert, ob eine bestimmte Bedingung erfüllt ist, die den Test stoppt (etwa, wenn ein Crash erzeugt wurde, oder eine bestimmte Coverage erreicht wurde).

Aus konzeptioneller Sicht bestehen Systeme zur automatisierten dynamischen Analyse immer zumindest aus den folgenden 3 Funktionalitäts-Modulen:

1. Application Under Test (AUT)
   Jene Anwendung, die analysiert werden soll.

2. UI-Explorer
   Hier wird eine Entscheidung darüber getroffen, wie die Benutzeroberfläche bedient werden soll. Meist wird die gewählte Interaktion dann auch gleich ausgeführt.

3. State-Observer
   Je nach individuellem Anwendungsfall wird hier überprüft, ob ein Fehlerfall vorliegt und ob die Bedingung zur Beendigung des Tests erfüllt ist. Im Fall einer Sicherheitsanalyse kann hier etwa festgestellt werden, ob ein bestimmtes bösartiges Verhalten bemerkt wurde (etwa Zugriff auf eine bestimmte sensible API-Schnittstelle), oder ob eine Anfälligkeit für eine bestimmte Sicherheitslücke erkannt wurde (wenn etwa eine Crypto-API unsicher parametriert wurde). Auch die Test-Coverage wird hier ermittelt.

Zusätzlich zu diesen 3 Kernbestandteilen können häufig noch folgende Elemente benannt werden:

4. Test-Orchestrator
   Dirigiert im übertragenen Sinne die gesamte Testprozedur, indem hier die anderen Komponenten gestartet werden, Fehlerfälle behandelt werden, verschiedene AUTs auf verschiedene Test-Geräte verteilt werden und so weiter.

Häufig wird die AUT außerdem in einer speziell vorbereiteten Umgebung ausgeführt, in der Funktionsaufrufe genauer beobachtet werden können oder Lifecycle-Events von App-Komponenten künstlich erzeugt werden können. Diese Umgebung könnte konzeptionell im State-Observer oder Test-Orchestrator angesiedelt werden, oder aber als eigenständige Komponente betrachtet werden.

4. Dynamisches Testen unter Android

Im Rahmen dieses Projekts wurde dynamisches Testen insbesondere im Sicherheits-Kontext unter Android behandelt. Hier besteht eine AUT in der Regel aus einem Android-App-Paket (bestehend aus einem oder mehreren APK-Files), die in einer Test-Umgebung (entweder ein Android-Emulator oder ein

mit dem Test-Orchestrator verbundenes physisches Gerät) installiert und ausgeführt werden. Der Test-Orchestrator wird meist auf Basis der Android Debug Bridge (adb) realisiert, die über Kabel- oder Drahtlosverbindung die Steuerung eines oder mehrerer physischer oder virtueller Android-Geräte erlaubt.

Die größte Herausforderung im automatisierten dynamischen Testen unter Android ist im UI-Explorer zu finden. Hier müssen effektive Nutzereingaben getroffen werden, um möglichst viel der App-Funktionalität zu erfassen. Im Abschnitt 4.1 gehen wir auf die verschiedenen Ansätze ein, um diese Problemstellung zu lösen.

Der State-Observer für dynamisches automatisiertes Testen unter Android besteht üblicherweise aus einem test-spezifischen Teil und einem allgemeinen Teil. Auf die Coverage-Erfassung im allgemeinen Teil gehen wir anschließend in Abschnitt 4.2 ein. Beim test-spezifischen Teil kann es zum Beispiel um die Erfassung von Crypto API Misuse gehen [1].

4.1. UI-Exploration

Bei der Explorations-Strategie der verschiedenen aktuellen Ansätze für UI-Exploration unter Android lassen sich zwei wesentliche Strömungen feststellen.

Zufallsbasierte Lösungen erzeugen zufällige Input-Events. Dazu wird keinerlei Information über die Struktur der Benutzeroberfläche benötigt. Der Ansatz ist daher einfach zu realisieren und in Bezug auf die Entscheidungszeit sehr performant. Dennoch handelt es sich in Summe um einen eher ineffektiven Ansatz, da eine große Zahl an Input-Events generiert und ausgeführt werden muss, bis ein sinnvolles Event gefunden wird, also eine Änderung des Benutzeroberflächen-Zustands (etwa durch Navigation auf einen anderen App-Bildschirm) erreicht wird.

Modellbasierte Lösungen nutzen Informationen über die Benutzeroberfläche und bisher ausgeführte Aktionen, um ein Modell der AUT zu bauen. Dieses Modell kann dann wie eine Karte genutzt werden, um etwa Input-Events möglichst so zu wählen, dass bereits erreichte Benutzeroberflächen-Zustände nicht erneut getestet werden. Für diesen strukturierten Ansatz benötigt der UI-Explorer häufig neben dem rein grafischen Framebuffer auch Informationen über die Struktur der Benutzeroberfläche, also etwa über die Position oder Beschriftung von Buttons, die unterstützten Input-Events eines UI-Elements, usw. Er eignet sich daher nicht für Anwendungen, die solche Informationen nicht zur Verfügung stellen (etwa weil sie nicht die Standard-UI-Bibliothek verwenden).

Eine besondere Herausforderung für jede UI-Explorations-Methode sind sogenannte Exploration Tarpits [2] bzw. Gate GUIs. Dabei handelt es sich um App-Bildschirme, die nur überwunden werden können, wenn strukturierter, sinnvoller Input für Eingabemasken gefunden wird. Je weniger Hinweise auf den erwarteten Input die UI-Struktur hier liefert, desto mehr Kontext muss die UI-Exploration selbst sammeln bzw. interpretieren. Während etwa Eingabeelemente mit einer definierten Anzahl an Zuständen (etwa ein Schalter) einfach mit sinnvollen Daten zu befüllen sind (jeder mögliche Zustand ist in der Regel gültig), stellen insbesondere Freitextfelder eine große Herausforderung dar. Diese sind zwar für menschliche Nutzer in der Regel leicht zu befüllen, dennoch stoßen automatisierte Ansätze hier häufig an ihre Grenzen.

Die insbesondere in der Praxis am häufigsten benutzte Methode für UI-Exploration unter Android fällt in die Kategorie der zufallsbasierten Lösungen. Es handelt sich hier um das Tool Monkey [1], das als Teil des Android Open Source Project entwickelt und veröffentlicht wird. Der eigentliche Zweck des Tools ist das Stress-Testing von Android-Anwendungen. Hierfür ist in der Regel eine hohe Quantität an Input-Events

[1] Android Developers: UI/Application Exerciser Monkey:
https://developer.android.com/studio/test/other-testing-tools/monkey

nötig, während für die UI-Exploration auch die Qualität der Events eine entscheidende Rolle spielt. Monkey stellt also zwar einerseits eine bequeme Lösung für die UI-Exploration dar, da ein bestehendes Tool übernommen werden kann, das schon über Funktionalität für die Injection von Input-Events verfügt. Andererseits handelt es sich aber um keine optimale Lösung, da die Qualität der (zufällig generierten) Input-Events nicht den Anforderungen an eine sinnvolle UI-Exploration genügt.

Der State-Of-The-Art für UI-Exploration unter Android kann in der zweiten Kategorie von Lösungen gefunden werden, also in den modellbasierten Ansätzen. Einige der aktuellen Ansätze setzen auf Machine Learning. Besonders erwähnenswert ist in diesem Bereich die Arbeit Fastbot bzw. der Nachfolger Fastbot2 [3]. Der Ansatz benutzt Reinforcement Learning, um ein ML-Modell zu bauen. Konkret wird für jedes ausgeführte Input-Event (für den gegebenen UI-Zustand) mitaufgezeichnet, ob eine Änderung des UI-Zustands herbeigeführt wurde. So können später jene Input-Events bevorzugt werden, die innerhalb einer gegebenen Laufzeit zur höchsten Coverage führen. Obwohl das Tool für den weiten Einsatz in der Forschung gedacht ist, ist die Lizenz des kürzlich veröffentlichten Quellcodes unklar. Das Projekt kann also nicht ohne rechtliche Unsicherheit adaptiert oder in andere Projekte integriert werden.

Eine Alternative mit Open-Source-Lizenz kann hier in Droidbot [4] gefunden werden. Das Projekt hat einen besonderen Fokus auf Erweiterbarkeit und Flexibilität gesetzt. Zwar besteht schon Funktionalität zur Injektion von Input-Events und zum Bau eines Modells der UI-Zustände, allerdings muss ein Algorithmus zur Auswahl der Input-Events erst implementiert werden. Es handelt sich also eher um ein Framework als eine Lösung, die sofort zum Einsatz kommen kann.

Eine konkrete Lösung, die auf Droidbot aufbaut, kann in Humanoid [5] gefunden werden. Hier kommt wiederum ein Machine-Learning-Ansatz zum Einsatz, der Entscheidungen über sinnvolle Input-Events anhand eines ML-Modells trifft, das auf Basis von UI-Interaktionen echter menschlicher App-Anwender trainiert wurde. Während der Ansatz prinzipiell vielversprechend ist, ist die Implementierung mit einigen Einschränkungen behaftet. So bremst die Evaluierung des ML-Modells die Analyse und es werden nur vordefinierte Displayauflösungen unterstützt.

Besonders spannende Ansätze sind in Publikationen zu finden, die sich der Generierung von sinnvollem Text in Freitext-Felder widmen. Hier ist besonders QTypist [6] zu erwähnen, das in jeder Iteration ein Large Language Model (LLM) befragt, um eine Auswahl des nächsten Input-Events zu treffen. Hier ist anzumerken, dass QTypist das proprietäre Modell GPT-3 nutzt, das nur kostenpflichtig auf einem Server ausgeführt werden kann.

4.2. Coverage-Erfassung

Die Coverage-Erfassung stellt auf mobilen Plattformen ein besonderes Problem dar, da hier im Gegensatz zu Desktop-Systemen Anwendungen nicht unauffällig modifiziert werden können. Genauer müssen Anwendungen unter Android vom Entwickler signiert werden, bevor sie auf einem Gerät installiert werden können. Wird eine App modifiziert, so muss eine neue (vom Original unterscheidbare) Signatur angelegt werden. Um Repackaging-Angriffe zu unterbinden, überprüfen viele Apps zur Laufzeit ihre APK-Signatur. Wird nicht die ursprüngliche App-Signatur entdeckt, wird die weitere Ausführung verweigert. Wird App-Attestation verwendet, so besteht keine Möglichkeit, diesen Sicherheitsmechanismus zu umgehen. Um größtmögliche App-Kompatibilität sicherzustellen, ist also wünschenswert, dass zur Coverage-Erfassung die App nicht modifiziert werden muss.

Aktuell stehen allerdings zur exakten (zeilen-genauen) Coverage-Erfassung lediglich solche Ansätze zur Verfügung, die Modifikationen am APK-File voraussetzen. Hier handelt es sich konkret um die Lösung ACVTool [7].

Einige alternative Ansätze übersetzen DEX-Code in Java, um dann Java-Lösungen für die zeilengenaue Coverage-Messung zu nutzen. Diese Ansätze leiden in der Regel zusätzlich zu den oben diskutierten

Signatur-Problemen noch unter Kompatibilitäts-Problemen. Das Kompilat (DEX-Files) kann im Allgemeinen nicht fehlerfrei in Java rückübersetzt werden, da beim Kompilieren Informationen zum Quellcode verloren gehen.

Für Anwendungsfälle, in denen die AUT nicht modifiziert werden soll, wird daher häufig auf eine exakte (zeilengenaue) Coverage-Messung verzichtet, und stattdessen nur sehr grobe Activity-Coverage genutzt. Hierfür wird das App-Manifest der AUT analysiert, um eine Liste aller enthaltenen Activity-Klassen zu erhalten. Während des Tests werden dann aus dem System-Log Activity-Launch-Events herausgelesen. So kann erfasst werden, wie hoch der Anteil an Activities ist, die vom Test erreicht werden.

5. Evaluierung aktueller UI-Exploration-Ansätze

Im Rahmen der Publikation eines Ansatzes wird dieser meist auch anhand eines Datensatzes an Android-Apps evaluiert. Obwohl diese Evaluierung dazu dienen soll, Lösungen objektiv vergleichen zu können, kommen meist verschiedene Metriken und App-Datensätze zum Einsatz, so dass keine direkte Vergleichbarkeit gegeben ist. Um hier Abhilfe zu schaffen, haben wir im Rahmen dieses Projekts eine eigene Evaluierung verschiedener aktueller Ansätze durchgeführt.

5.1. Getestete Tools

Zum Einsatz kamen alle aktuellen Ansätze aus der Literatur, die (1) sich für Black-Box-Testen eignen und (2) der Öffentlichkeit zumindest als ausführbares Binary zur Verfügung gestellt wurden. Konkret handelt es sich hier um die beiden bereits oben erwähnten modellbasierten Ansätze Fastbot2 und Humanoid, sowie das zufallsbasierte Tool Monkey.

5.2. Datensatz

Um den Zeitaufwand der Evaluierung zu begrenzen, wurden die Größe des Datensatzes schon zu Beginn auf 50 Anwendungen beschränkt. Diese wurde aus den nach Download-Zahlen populärsten Apps vom größten Android-App-Store Google Play gewählt. Konkreter handelt es sich um die 10 beliebtesten Gratis-Anwendungen aus den Kategorien Business, Bildung, Gesundheit & Fitness, Lifestyle und Sport. Wenn eine In-App-Zahlung Voraussetzung für den (sinnvollen) Betrieb einer App war, wurde an ihrer Stelle die nächste App im Ranking ausgewählt.

5.3. Test-Umgebung

Die Tests wurden auf einem OnePlus 7 Pro durchgeführt, das mit dem offiziellen Android 11-Build ausgestattet wurde. Jedes Tool wurde je App 3 Mal für 20 Minuten laufen gelassen. Vor jedem Durchgang wurden die App-Daten am Gerät vollständig gelöscht. Als Indikator für den Erfolg eines Ansatzes wurde die Activity-Coverage herangezogen, die wie erwähnt ohne Modifikationen der AUT erfasst werden kann. Da die Test-Laufzeit für jedes Tool gleich beschränkt wurde, dient die Coverage hier also als Maß für die Performance des Tools, also dafür, wie viel Coverage pro Zeit das Tool erreicht.

5.4. Ergebnisse

Wie in Tabelle 1 gezeigt, konnte Fastbot2 im Test-Szenario die höchste Performance zeigen. Als überraschende Erkenntnis zeigt sich, dass modellbasierte Ansätze nicht immer den deutlich simpleren zufallsbasierten Ansätzen überlegen sind. Gründe hierfür können in der generellen Unzuverlässigkeit von Humanoid gefunden werden (regelmäßige Fehler), aber auch in der zeitaufwändigen Evaluierung des ML-Modells.

Dass auch der beste modellbasierte Ansatz (Fastbot2) nur einen in Relation zur zusätzlichen Implementierungs-Komplexität niedrigen Performancegewinn (etwa 25 %) gegenüber dem zufallsbasierten Monkey-Tool zeigt, ist wohl ein Hinweis darauf, dass moderne Apps über komplexe Benutzeroberflächen verfügen, die schwer zu modellieren sind.

Tabelle 1 Performance-Vergleich der getesteten UI-Exploration-Ansätze

Fastbot2	Humanoid	Monkey
17,53 %	11,19 %	13,89 %

Insgesamt muss auch festgestellt werden, dass alle getesteten Ansätze weniger als 20% der Activities der getesteten Apps abdecken. Zwar kann die Activity-Coverage nur als Hinweis für die Funktionalitäts-Coverage gewertet werden, aber dennoch kann davon ausgegangen werden, dass auch bei den besten aktuellen Ansätzen noch deutlicher Verbesserungsbedarf besteht.

6. Zusammenfassung

In diesem Bericht beleuchteten wir die aktuelle Situation von Black-Box-UI-Testing zur automatisierten dynamischen Sicherheits-Testung von Android-Anwendungen. Dazu fassten wir die technischen Grundlagen und Grundbestandteile entsprechender Systeme zusammen, und diskutierten die vielversprechendsten von der Wissenschaft vorgestellten Ansätze der jüngeren Vergangenheit. Schließlich zeigten wir anhand unserer eigenen unabhängigen Evaluierung einen objektiven Performance-Vergleich von verschiedenen der Öffentlichkeit zugänglichen Ansätzen zur UI-Exploration.

Referenzen

[1] L. Piccolboni, „CRYLOGGER: Detecting Crypto Misuses Dynamically," in *IEEE Symposium on Security and Privacy*, 2021.

[2] W. Wang, „Vet: identifying and avoiding UI exploration tarpits," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[3] Z. Lv, „Fastbot2: Reusable automated model-based gui testing for android enhanced by reinforcement learning," in *International Conference on Automated Software Engineering*, 2022.

[4] Y. Li, „DroidBot: A Lightweight UI-Guided Test Input Generator for Android," in *IEEE International Conference on Software Engineering Companion*, 2017.

[5] Y. Li, „Humanoid: A Deep Learning-based Approach to Automated Black-box Android App Testing," in *International Conference on Automated Software Engineering*, 2019.

[6] Z. Liu, „Fill in the Blank: Context-aware Automated Text Input Generation for Mobile GUI Testing," 2022. [Online]. Available: https://arxiv.org/abs/2212.04732. [Zugriff am 25 08 2023].

[7] A. Pilgun, „Fine-grained code coverage measurement in automated black-box android testing," *ACM Transactions on Software Engineering and Methodology,* Bd. 29, 2020.