UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Una investigación empírica para comparar diferentes herramientas de GUI testing en Android

Trabajo Fin de Máster

Máster Universitario en Ingeniería y Tecnología de Sistemas Software

AUTOR/A: Chorfi, Moujib

Tutor/a: Vos, Tanja Ernestina

Director/a Experimental: MARIN CAMPUSANO, BEATRIZ MARIELA

CURSO ACADÉMICO: 2023/2024

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Sistemas Informáticos y de Computación
Universitat Politècnica de València

Una investigación empírica para comparar diferentes herramientas de GUI testing en Android

TRABAJO FINAL DE MASTER (TFM)

Master MUITSS

Autor: Moujib Chorfi

Tutor: Tanja Ernestina Josefina Vos
Beatriz Mariela Marín Campusano

Curso 2022-2023

Resumen

Realizar una investigación empírica para comparar diferentes herramientas de GUI testing en Android es crucial para mejorar la calidad de las aplicaciones móviles y garantizar una experiencia de usuario satisfactoria. Al comparar y evaluar diferentes herramientas, es posible identificar fortalezas y debilidades en cuanto a su capacidad para detectar errores en la interfaz gráfica de usuario (GUI) de una aplicación, su facilidad de uso y su eficiencia en términos de tiempo y recursos. Además, esta investigación también puede contribuir al desarrollo de nuevas herramientas de testing que sean más precisas y eficientes en la detección de errores desde la GUI de las aplicaciones Android. En resumen, la realización de una investigación empírica para comparar diferentes herramientas de GUI testing es esencial para garantizar la calidad de las aplicaciones móviles y mejorar la experiencia de los usuarios.

Palabras clave:

Investigación empírica, herramientas de GUI testing, aplicaciones Android, calidad, experiencia de usuario, MINT.

Resum

Realitzar una investigació empírica per comparar diferents eines de GUI testing en aplicacions Android és crucial per millorar la qualitat de les aplicacions mòbils i garantir una experiència d'usuari òptima. En comparar i avaluar diferents eines, és possible identificar forces i debilitats quant a la seva capacitat per detectar errors en la interfície gràfica d'usuari (GUI) d'una aplicació, la seva facilitat d'ús i la seva eficiència en termes de temps i recursos. A més, aquesta investigació també pot contribuir al desenvolupament de noves eines de testing que siguin més precises i eficients en la detecció d'errors en la GUI de les aplicacions Android. En resum, la realització d'una investigació empírica per comparar diferents eines de GUI testing és essencial per garantir la qualitat de les aplicacions mòbils i millorar l'experiència dels usuaris.

Paraules clau:

Investigació empírica, eines de GUI testing, aplicacions Android, qualitat, experiència d'usuaris, MINT.

Abstract

Conducting empirical research to compare different GUI testing tools in Android applications is crucial for enhancing mobile app quality and ensuring a satisfactory user experience. By comparing and evaluating various tools, it becomes possible to identify strengths and weaknesses in terms of their ability to detect errors in the graphical user interface (GUI) of an application, their user-friendliness, and efficiency in terms of time and resources. Furthermore, this research can also contribute to developing new testing tools that are more precise and efficient in detecting errors from the GUI of Android applications. In summary, conducting empirical research to compare different GUI testing tools is essential for ensuring mobile app quality and enhancing user experience.

Keywords:

Empirical research, GUI testing tools, Android applications, quality, user experience, MINT.

Índice general

Índice de algoritmos

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

Android, el sistema operativo móvil más usado en el mundo, se ha convertido en un pilar fundamental de nuestra vida cotidiana. Desde teléfonos inteligentes hasta tabletas y dispositivos portátiles, Android ha conquistado el mercado con su flexibilidad y diversidad. Esta omnipresencia se traduce en una demanda constante de aplicaciones Android que mejoren nuestra productividad, entretenimiento y estilo de vida.

La proliferación de aplicaciones móviles ha sido asombrosa, pero detrás de cada aplicación exitosa se esconde un proceso de desarrollo y **testing** que busca garantizar su calidad y funcionalidad. En este contexto, la calidad de una aplicación Android se traduce directamente en la experiencia del usuario[38],[31]. Un mal funcionamiento o un diseño defectuoso pueden alejar a los usuarios en cuestión de segundos [33]. En este sentido, la necesidad de investigaciones dedicadas a mejorar las prácticas de testing de las aplicaciones Android es evidente.

La **ingeniería del testing** se enfrenta a desafíos únicos, principalmente debido a la complejidad de los sistemas software. El testing de software desempeña un papel central. Se estima que representa más del 50 % del costo total en proyectos de desarrollo [19]. El testing se erige como un pilar fundamental en el ciclo de vida del desarrollo de software, ya que garantiza una mayor calidad al identificar y subsanar errores y fallos antes del lanzamiento de la aplicación. No obstante, es importante reconocer que el testing de software no es una ciencia exacta. Explorar exhaustivamente todas las posibles situaciones de uso del software conduce a una explosión combinatoria, lo que hace que el proceso sea desafiante. Además, la detección automática de comportamientos incorrectos representa un problema recurrente en el campo del testing de software.

El Android testing se puede realizar a diferentes niveles, incluyendo el unit testing, que se enfoca en probar componentes individuales de la aplicación de forma aislada, y el **GUI testing**, que se centra en la interacción del usuario a través de la interfaz gráfica de usuario. El GUI testing permite evaluar el comportamiento de una aplicación a través de su interfaz gráfica. GUI testing implica la simulación de acciones de un usuario, como hacer clic en botones, introducir texto y realizar gestos, con el objetivo de verificar si la aplicación responde correctamente. El GUI testing reviste una importancia crítica en el desarrollo de aplicaciones, ya que valida el comportamiento del sistema en su conjunto a través de la interfaz gráfica. Además, permite simular la experiencia del usuario final para testear el correcto funcionamiento de la app, lo que contribuye significativamente a la calidad y satisfacción del usuario.

Hasta la fecha, gran parte del proceso de GUI testing en el mundo de aplicacAndroid ha dependido en gran medida de métodos *manuales* o *scripted*, tanto en ejecución como en la creación de pruebas. Aunque efectivos, estos enfoques pueden resultar costosos en términos de tiempo y recursos [14], lo que ha llevado a la exploración de nuevas técnicas, como *scriptless* testing, que busca automatizar y simplificar este proceso sin la necesidad de crear scripts. . En este trabajo de tesis, se exploran las prácticas actuales de GUI testing **scriptless** para Android y las investigaciones que buscan impulsar la calidad de las aplicaciones. Se analizan y comparan diferentes herramientas y enfoques de GUI testing específicamente diseñados para aplicaciones Android, evaluando su efectividad en términos

de cobertura, su capacidad para detectar errores, su facilidad de uso y su eficiencia en términos de tiempo y recursos utilizados.

1.1 Motivación

TESTAR (www.testar.org) [41] es una herramienta de scriptless testing desarrollada por la Universidad Politécnica de Valencia (UPV) con la colaboración de otros socios. El desarrollo de TESTAR comenzó durante el proyecto FITTEST [43] coordinado por la UPV durante los años 2010-2013. Posteriormente, el TESTAR continuó su desarrollo mientras se perfeccionaba la herramienta por varias compañías en proyectos ERASMUS+ como SHIP [42] y en proyectos financiados por los gobiernos de España y Valencia. En 2017, su desarrolló continuó en el contexto del proyecto TESTOMAT ITEA3.

Durante 2022, a solicitud del banco ING, la herramienta TESTAR se amplió con un complemento que permite la prueba automatizada de aplicaciones Android [32]. Esto resultó en MINTESTAR y en la publicación de un artículo [32] que muestra la complementariedad de TESTAR con las pruebas con scripts.

Con MINTESTAR como resultado, surgió la necesidad de comparar esta versión de TESTAR con otras herramientas de scriptless testing del estado del arte, describir las diferencias, similitudes y evaluar la efectividad de estas nuevas técnicas de TESTAR. Todo esto para el fin de escribir un articulo científico a publicar en una conferencia. Esta es la motivación del trabajo de tesis actual.

Dado que los resultados del trabajo de tesis de Máster se incluirán en una publicación científica, la contribución al avance del conocimiento es evidente. Este proyecto no sólo busca abordar desafíos prácticos en las pruebas de GUI de Android, sino que también aspira a contribuir al avance del conocimiento en esta área. A través de una investigación empírica rigurosa, se espera identificar las herramientas más efectivas y las mejores prácticas en este campo, lo que beneficiará tanto a la comunidad académica como a la industria del desarrollo de aplicaciones móviles.

Además, este proyecto representa una oportunidad invaluable para el desarrollo profesional y académico del estudiante. A través de la investigación, la experimentación y la colaboración con expertos en el campo del Android GUI testing, espero adquirir habilidades técnicas avanzadas y una comprensión profunda de los desafíos y oportunidades en este ámbito.

1.2 Descripción del problema

Las herramientas para scriptless GUI testing [41] generalmente facilitan la generación de secuencias de prueba mediante diferentes formas de selección de acciones a probar. El conjunto de acciones posibles en la aplicación se deriva automáticamente durante la prueba en función de la estructura de la interfaz gráfica de usuario (GUI). Una vez que se detectan las acciones posibles, la siguiente acción puede seleccionarse de varias maneras, que incluyen de manera aleatoria, algoritmos evolutivos [25, 22], algoritmos de aprendizaje por refuerzo que recompensan la exploración [26], etc.

El flujo lógico de alto nivel de estas herramientas se describe visualmente en la Figura 1.1. En más detalle, los pasos se pueden describir de la siguiente manera:

Paso 1: Iniciar la aplicación bajo prueba (application under test - AUT) y esperar hasta que esté lista para la interacción.

Paso 2: Inspeccionar la AUT para obtener información sobre la aplicación y los componentes individuales presentes en su estado actual. Como resultado, se obtiene una representación del estado actual de la aplicación.

Paso 3: Determinar las acciones disponibles a partir de la representación obtenida de la AUT.

Paso 4: Seleccionar una acción de la lista derivada de acciones disponibles para ejecutar, mediante el uso del ASM (Modelo de Estado Abstracto) disponible.

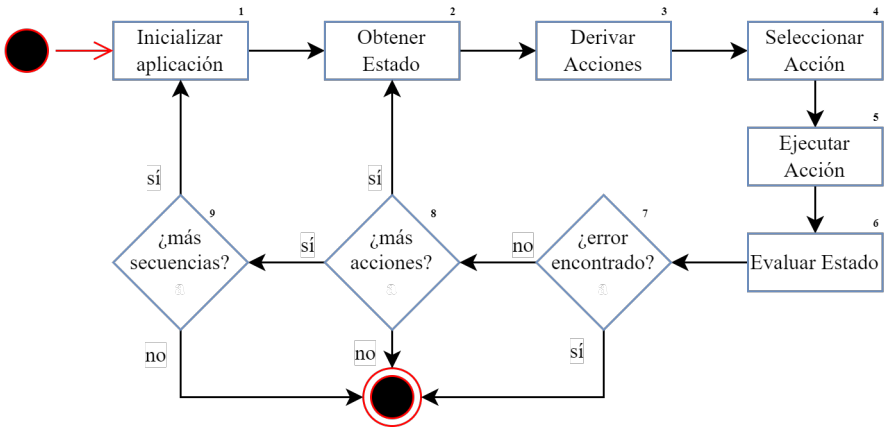

Figura 1.1: Flujo lógico de ejecución de scriptless testing.

Paso 5: Ejecutar la acción seleccionada en la AUT.

Paso 6: Verificar todos los oráculos de prueba definidos para cada estado (o al final de una secuencia de prueba) para detectar posibles fallos.

Paso 7: Si no se encuentran fallos, se repiten los pasos 2, 3, 4, 5 y 6 hasta que se haya generado la longitud de secuencia deseada.

Paso 8: Si no se ha alcanzado el número deseado de secuencias, se repite todo el proceso desde el paso 1. De lo contrario, la herramienta finaliza y sale de manera ordenada.

Como se indicó anteriormente, TESTAR [41] es una herramienta de scriptless testing que está siendo desarrollada por la UPV y otros colaboradores. En 2022, se ha ampliado con la capacidad de probar también aplicaciones para Android[32]. Sin embargo, TESTAR no es la única herramienta para scriptless testing; existen otras, y estas herramientas pueden diferir en cuanto a la detección de estados y acciones, selección de acciones, oráculos de prueba, etc. Para poder situar TESTAR en el estado del arte del scriptless testing a nivel del GUI, es necesario realizar un estudio empírico comparativo.

1.3 Objetivos

El objetivo general de este trabajo consiste en *analizar y comparar las herramientas de Android scriptless testing que constituyen el estado-del-arte, evaluando su efectividad en términos de cobertura de código, oráculos, capacidad de detección de errores y eficiencia en el uso de recursos y tiempo.*.

Para cumplir el objetivo general, se han trazado los siguientes objetivos específicos:

1. Realizar una revisión de la literatura relacionada con las herramientas de Android scriptless testing, identificando sus principales características, enfoques y desafíos.

2. Seleccionar un conjunto representativo de dichas herramientas para llevar a cabo la investigación, así como del conjunto de aplicaciones Android que serán utilizadas.

3. Implementar un entorno de prueba controlado, recopilando datos sobre la cobertura de código alcanzada y la eficacia en la detección de bugs.

4. Realizar un análisis crítico de los resultados y discutir las implicaciones de la comparación, resaltando las fortalezas y debilidades de las herramientas evaluadas.

1.4 Resultados Esperados

En esta sección, se establecen los resultados anticipados de la investigación. Se espera que los experimentos de comparación de herramientas de Android scriptless testing proporcionen información relevante sobre la efectividad de estas herramientas en términos de **cobertura de código** y **detección de fallos**. Los resultados esperados incluyen:

1. **Comparación de Cobertura de Código**: Se espera que el experimento muestre cómo las diferentes herramientas de testing afectan la cobertura de código en las aplicaciones Android seleccionadas. Esto se evaluará mediante métricas de cobertura que indicarán qué herramientas logran una mayor cobertura.

2. **Detección de fallos**: Se anticipa que el experimento revele las capacidades de detección de fallos de cada herramienta. Esto se evaluará mediante la identificación de fallos conocidos y la capacidad de las herramientas para detectarlos.

3. **Comparación de Rendimiento**: Se espera que los resultados también permitan comparar el rendimiento de las herramientas en términos de tiempo de ejecución y recursos necesarios.

4. **Análisis Cualitativo**: Además de los datos cuantitativos, se espera que los resultados incluyan un análisis cualitativo crítico de las fortalezas y debilidades de las herramientas evaluadas. Esto proporcionará una comprensión más profunda de los resultados cuantitativos.

1.5 Metodología

Para llevar a cabo este trabajo, se ha considerado una metodología mixta que combine enfoques cuantitativos y cualitativos. A continuación, se detallan los pasos y métodos que se seguirán en esta investigación:

1.5.1. Revisión de la Literatura

En esta etapa inicial, se realizará una revisión exhaustiva de la literatura relacionada con las herramientas de Android scriptless testing. El objetivo es identificar y comprender las principales características, enfoques y desafíos asociados con estas herramientas. Se analizarán estudios previos, investigaciones académicas y recursos técnicos relevantes para obtener una visión completa del estado actual del arte en este campo.

1.5.2. Selección de Herramientas y Aplicaciones de Android

Con base en la revisión de la literatura, se evaluarán las aplicaciones considerando cantidad, variedad y calidad. Se priorizarán aplicaciones estables con código fuente disponible. También se realizarán pruebas preliminares en aplicaciones de F-Droid y se seleccionará un benchmark específico de una investigación, **'Time-travel Testing of Android Apps'**,[24] que incluye 8 de las 100 aplicaciones más populares en **F-Droid** y **Google Play**, todas instrumentadas con JaCoCo para pruebas detalladas.

1.5.3. Implementación de un Entorno de Prueba Controlado

Se configurará un entorno de prueba controlado para llevar a cabo la investigación de manera sistemática. En este entorno, se ejecutarán pruebas utilizando las herramientas seleccionadas (Ver 2.1 , 2.2) siguiendo los escenarios de prueba predefinidos por cada herramienta. Estos escenarios son

diseñados específicamente por los desarrolladores de cada herramienta y representan situaciones típicas en las que se evaluará la aplicación Android .

Durante las pruebas, se recopilarán datos sobre la cobertura de código alcanzada por cada herramienta y su eficacia en la detección de fallos.

1.5.4. Análisis de Resultados

Los datos recopilados durante las pruebas serán analizados de manera crítica. Se realizará una evaluación comparativa de las herramientas en términos de su rendimiento en cuanto a cobertura de código, capacidad de detección de fallos. Los resultados se presentarán de manera clara y se discutirán en profundidad, resaltando las fortalezas y debilidades de cada herramienta evaluada.

1.5.5. Implicaciones y Conclusiones

Se extraerán conclusiones significativas de los resultados obtenidos y se discutirán las implicaciones de la comparación realizada. Esta sección destacará la relevancia de la investigación y su contribución al campo de las pruebas automatizadas en entornos Android.

1.6 Estructura del documento

El documento se estructura en varios capítulos que abordan diferentes aspectos de la investigación. A continuación, se presenta una breve descripción de cada capítulo:

En el capítulo 2 Estado del arte, se presenta una revisión detallada de las herramientas de Android scriptless testing, incluyendo Dynodroid, Sapienz y AutoDroid.

El capítulo 3 MINTESTAR, o simplemente Mint, se centra en la herramienta Mint, proporcionando una descripción detallada de su funcionamiento y algoritmo.

El capítulo 4 Diseño de los experimentos describe el diseño experimental, incluyendo las preguntas de investigación, la selección de herramientas y aplicaciones, métricas y detalles de los experimentos.

El capítulo 5 Evaluación presenta los resultados obtenidos para cada pregunta de investigación y su análisis correspondiente.

En el capítulo 6 Conclusiones y trabajo futuro, concluye el trabajo y se discuten las implicaciones de los resultados, además de proponer áreas para investigaciones futuras.

CAPÍTULO 2
Estado del arte

Esta sección presenta una visión detallada de las herramientas de scriptless testing en Android utilizadas en esta investigación.

En la fase de revisión de literatura, se empleó la plataforma digital Google Scholar, que posibilita llevar a cabo búsquedas precisas. Dicha plataforma ofrece la posibilidad de búsquedas basadas en palabras clave, permitiendo la obtención de investigaciones relacionadas con: *android testing*, *GUI testing* y *scriptless testing*. Se utilizó Google Scholar como motor de búsqueda debido a su consolidación como recurso valioso para acceder a la literatura académica.

Luego de una extensa búsqueda, que incluyó la revisión de títulos y resúmenes de papers en Google Scholar, se obtuvo un total de 14 herramientas en el campo de scriptless GUI testing. Cada paper fue evaluado para determinar su relevancia e importancia en la obtención de información sobre las herramientas. La selección se basó en criterios como la novedad de los papers, la presencia de las herramientas en experimentos previos y la concordancia de las metricas utilizadas con las del presente estudio. Para facilitar la comprensión y la comparación, se ha preparado la tabla 2.1 que recopila información crucial sobre cada herramienta, cuyo análisis tuvo en cuenta los siguientes aspectos fundamentales para su comparación:

- **Técnicas de testing:** métodos utilizados para la exploración de las aplicaciones. En algunos casos la generación de un modelo es esencial para su posterior exploración, mientras otros casos la generación de pruebas es completamente aleatoria.
- **Oráculos:** mecanismo que se utiliza para determinar si el comportamiento de las aplicaciones es correcto o incorrecto.
- **Representación de los estados:** se refiere a la forma en que las herramientas capturan o interpretan los diferentes estados o condiciones del software bajo prueba. Esta representación es esencial para la evaluación el comportamiento del software y la detección de posibles errores, por medio de los oráculos.
- **Representación de las acciones:** se refiere a la forma de capturar o interpretar las acciones realizadas (o disponibles) durante la prueba. La representación de las acciones dictará el proceso de su selección, y como consecuencia, afectará los mecanismos de exploración de la aplicación bajo prueba.
- **Acciones disponibles:** interacciones que representan las funcionalidades y operaciones que el software bajo prueba ofrece y que pueden ser evaluadas. Ejemplos de acciones comunes pueden ser de navegación (explorar menús, pestañas, enlaces, etc.), de entrada de datos (ingresar información en formularios, campos de texto o áreas de entrada), de interacción con elementos de la interfaz (clic en botones, enlaces, imágenes).
- **Texto de entrada:** describe cómo se genera el texto de entrada para aquellas acciones disponibles de entrada de datos. El texto de entrada puede ser aleatoriamente generado, o basado

en métodos más elaborados. Según su complejidad, puede afectar tanto la forma en que se representan las acciones como el número de acciones disponibles.

Por otra parte, con el objetivo de comparar estas herramientas en un estudio empírico, se obtuvo la siguiente información relativa a los métodos de experimentación empleados para validar las herramientas existentes:

- **Métricas:** criterios para evaluar y comparar las herramientas disponibles.
- **Duración:** período de duración de una sesión de testing, que puede medirse tanto en tiempo como en número de episodios o acciones seleccionadas.
- **Herramientas comparadas:** conjunto de herramientas previamente existentes que fueron utilizadas como puntos de referencia para evaluar la eficacia de cada herramienta presentada.
- **Aplicaciones de prueba**: tipo (comercial o código abierto) y cantidad de aplicaciones utilizadas para la evaluación de cada herramienta presentada.
- **Plataforma:** entorno utilizado para la ejecución y evaluación de las aplicaciones. Una primera opción es la utilización de un emulador, simulando las características y comportamiento de un dispositivo Android, sin necesidad de tener acceso al hardware físico real. Por contraste, se puede utilizar un dispositivo real, como un smartphone, para probar y ejecutar aplicaciones de manera directa en un dispositivo real.

La tabla 2.2 muestra un resumen de los diferentes enfoques de experimentación llevados a cabo por cada investigación. Las tablas 2.1 y 2.2 servirán como referencia fundamental a medida que estudiamos cada herramienta y sus resultados en los siguientes apartados.

A continuación, se discute cada herramienta con mayor profundidad, describiendo las técnicas de testing y los algoritmos utilizados, la implementación de los oráculos, la representación del estado e identificación de acciones y otros aspectos relevantes de cada una de las herramientas. Este análisis detallado permitirá comprender mejor cómo funcionan estas herramientas.

Tabla 2.1: Principales características de las herramientas de scriptless GUI testing en Android.

Herramienta	Técnica de Testing	Oráculos	Representación del Estado	Representación de las Acciones	Acciones Disponibles	Texto de Entrada
Dynodroid [35]	Combinación de acciones menos ejecutadas con testing aleatorio	Excepciones (excepción fatal)	Árbol de widgets (vista de jerarquía)	Eventos de aplicación (SDK) y de sistema	Tap, long tap, drag, texto de entrada	Cadenas predefinidas.
Sapienz[36]	Exploración sistemática y basada en búsqueda, testing aleatorio	Detección de fallos	Sin concepto de estado, conjunto de widgets	Interacciones basadas en eventos	Texto de entrada, toque, movimiento, rotación, trackball, voltear, navegación,operaciones sobre el sistema	Cadenas predefinidas obtenidas por ingeniería inversa sobre la apk, cadenas predefinidas genéricas.
AutoDroid[38]	Pruebas aleatorias,pruebas basadas en modelos, aprendizaje por refuerzo (q-learning), pruebas combinatorias, crash Análisis	Detección de fallos	Pantalla actual, visibilidad y propiedades de los elementos de la GUI (botones, campos de texto, menús, etc.), contenido del texto mostrado, opciones seleccionadas	Interacciones basadas en eventos, exploración de la interfaz gráfica de usuario	Acciones de la GUI, eventos GUI	Aleatorio
DroidBot[31]	Basado en modelos, exploración dinámica	Detección de fallos	DroidBot identifica acciones basadas en la exploración de los elementos e interacciones de la GUI de la aplicación.	Árbol de widgets y capturas de pantalla	Hacer clic, escribir (ingresar texto), desplazarse, deslizar y varios otros gestos o interacciones	El texto de entrada se puede generar aleatoriamente o predefinido.
Humanoid[34]	Basado en modelos, aprendizaje profundo, análisis estático, análisis dinámico	Detección de fallos, problemas de rendimiento	Comportamiento y respuestas de la aplicación de Android	Identifica acciones basadas en las interacciones disponibles con la GUI de la aplicación.	Clics, pulsaciones prolongadas, deslizamientos, entrada de texto, pulsaciones de botones y otras interacciones de la interfaz de usuario	Puede generar texto de entrada para simular la entrada del usuario y explorar diferentes partes de la aplicación.
RegDroid [45]	Pruebas Aleatorias, pruebas basadas en modelos, pruebas basadas en búsqueda, pruebas basadas en aprendizaje	Excepciones	Árbol de widgets (vista de jerarquía)	Eventos de aplicación (SDK) y de sistema	Tocar, mantener pulsado, arrastrar, entrada de texto	Cadena fija.
Combodroid [44]	Combinaciones de casos de uso (combinatoria)	Excepciones	N/A	Exploración de GUI	Clic deslizar, ingresar texto	Generado sistemáticamente seleccionando combinaciones específicas de casos de uso o escenarios de prueba.
Timemachine [24]	Pruebas aleatorias, pruebas basadas en modelos, basadas en búsqueda, basadas en aprendizaje	Comparaciones con datos de referencia conocidos	El estado podría estar representado por el árbol de widgets o una captura de pantalla de la interfaz de usuario en un momento particular en el tiempo	Analizar las opciones de interacción disponibles (en función del tiempo) proporcionadas por la interfaz de usuario de la aplicación.	Clic, desplazamiento, entrada de texto	Tanto entradas aleatorias como de texto predefinido.
QTesting [37]	Aprendizaje por reforzamiento (Q-learning)	Detección de fallos	La interfaz gráfica de usuario (GUI) de una aplicación Android, composición de widgets	Formular eventos de interacción del usuario en aplicaciones como Acciones, las acciones que se pueden realizar en los elementos de la GUI	Clic, escribir, arrastrar y soltar, etc.	Texto de entrada aleatorio y predefinido.
AimDroid [29]	Pruebas aleatorias, pruebas basadas en modelos, pruebas basadas en eventos	Detección de fallos	Árbol de widgets, capturas de pantalla	Interacciones o eventos de usuario que se realizan en la interfaz gráfica de usuario (GUI) de la aplicación	Clic, escribir/introducir, desplazar, deslizar/gesto, selección de menú, navegación hacia atrás	Texto de entrada aleatorio y predefinido.
APE[30]	Técnica basada en modelos, técnicas de pruebas aleatorias	Detección de fallos/crash	Representación de la interfaz gráfica de usuario (GUI) de la aplicación	APE identifica acciones en función de los elementos de la GUI disponibles	Clic, escribir, desplazar, gesto, selección de menú, pulsación de tecla, acciones de múltiples pasos	Texto de entrada aleatorio y predefinido.
Droidmate2 [23]	Técnicas de pruebas basadas en modelos, pruebas aleatorias	detección de fallos/-crashes	Árbol de widgets o una captura de pantalla	Las acciones se identifican en función de la exploración de los elementos y las interacciones de la GUI de la aplicación.	Clics, deslizamientos, entradas de texto y gestos	Tanto aleatorias como predefinidas, según los requisitos de prueba
ARES[39]	Pruebas aleatorias, pruebas basadas en modelos, aprendizaje profundo por refuerzo	Detección de fallos/crashes	Árbol de widgets o una captura de pantalla	ARES identifica acciones en función de la exploración de los elementos y las interacciones de la GUI de la aplicación.	Clic, escribir (entrada de texto), desplazar, deslizar y varios otros gestos o interacciones	El texto de entrada puede generarse de forma aleatoria o predefinida.
Stoat [40]	Combinación de análisis estático y dinámico	Vulnerabilidades de seguridad	Modelo resultante del análisis estático	Acciones basadas en eventos, a partir de los widgets detectados en el análisis estático	Acciones detectadas con UIAutomator y acciones a nivel del sistema	N/A

Tabla 2.2: Comparación de métodos de experimentación de las herramientas de scriptless GUI testing en Android.

Herramienta	Métricas	Duración	Herramientas Comparadas	Aplicaciones	Plataforma
Dynodroid [35]	Cobertura de código, cobertura de eventos, detección de errores	4 episodios de 2K acciones	Monkey (3 episodios, 10K acciones) y testing manual ($\geq$ 2 episodios)	Cobertura: 50 aplicaciones de F-Droid, Errores: 1000 aplicaciones de Google Play	Emulador
Sapienz [36]	Cobertura de código, número de crashes, tamaño de las secuencias con crashes	1 hora / 30 minutos	Monkey, dynodroid	68 aplicaciones	Emulador y dispositivo real
AutoDroid [38]	Cobertura de código, cobertura de GUI, detección de fallas, tiempo de ejecución	Presupuesto de tiempo fijo de dos horas para completar el conjunto de pruebas	Monkey, catdroid tool	5 aplicaciones elegidas de F-droid	Emulador de android 10.0 pixel con API 29
DroidBot[31]	Cobertura de código y detección de fallas	Una hora para completar el conjunto de pruebas	Monkey, androidRipper, dynoDroid, swiftHand, puma, droidmate, droidBot	Cualquier aplicación de Android	Emulador y dispositivos reales (ambos)
Humanoid [34]	Detección de fallas, problemas de rendimiento y vulnerabilidades de seguridad en las aplicaciones de Android probadas	1 hora para cada aplicación de código abierto y 3 horas para cada aplicación de mercado	Monkey, puma, stoat, droidMate, sapienz, droidBot	68 de código abierto aplicaciones obtenidas de AndroTest [6] y 200 comerciales populares aplicaciones descargadas de Google Play	Estación de trabajo con dos CPU Intel Xeon E5-2620, 64 GB de RAM y una GPU NVidia GeForce GTX 1080 Ti
RegDroid [45]	Cobertura de Código y detección de Errores	50 pruebas GUI aleatorias (cada prueba contiene 100 eventos), que tomaron aproximadamente 12 horas	ODIN, genie, setDroid, diffdroid, stoat	Ocho aplicaciones Android de código abierto populares y representativas utilizadas en nuestro estudio (K=1,000, M=1,000,000)	Emulador y Dispositivos Reales (ambos).
ComboDroid [44]	Cobertura de código: cobertura de instrucciones de bytecode, examen de registros del sistema android	Resultado de cobertura de 12 horas	Monkey, sapienz, ape	Nueve aplicaciones seleccionadas al azar con al menos 10,000 descargas	Emulador
TimeMachine[24]	Cobertura de código, detección de fallas	N/A	Monkey, sapienz, stoat	Nueve aplicaciones seleccionadas al azar con al menos 10,000 descargas de las 100 principales aplicaciones instrumentadas de Google Play	Dispositivo real: dos máquinas físicas con 64 GB de memoria principal, ejecutando un sistema operativo Ubuntu 16.04 de 64 bits.
QTesting[37]	Cobertura de código, detección de fallas	N/A	Monkey,sapienz,stoat	50 aplicaciones android de código abierto	Máquina física con 4 núcleos, CPU de 3.60GHz y 16GB de RAM en Ubuntu 16.04.
AimDroid[29]	Cobertura de código, detección de fallas	Una hora para probar cada aplicación en un dispositivo android	Monkey, Sapienz	50 aplicaciones populares de código cerrado	Dispositivo Android del mundo real: tablet Nexus 7 (2013 wifi)
APE [30]	Cobertura de código, detección de fallas	Una hora para probar cada aplicación	Monkey, sapienz, stoat	15 aplicaciones ampliamente utilizadas compatibles con emuladores x86	Los emuladores se ejecutaron en una MacBook Pro 2016
DroidMate-2 [23]	Cobertura de código, detección de fallas	En ejecuciones cortas (5 minutos) y ejecuciones más largas (1 hora)	DroidBot, monkey	11 aplicaciones Android diferentes, seleccionadas al azar	Dispositivo real Google Nexus 5X, emuladores con 2 GB of RAM y android 10.0 (API nivel 29)
ARES [39]	Cobertura de código, detección de fallas	Con un límite de tiempo de una hora y se repitió 10 veces, para un total de 4,560 horas (190 días)	Fate, monkey, sapienz, timeMachine, q-Testing	100 aplicaciones seleccionadas al azar de entre las 500 aplicaciones más populares de F-Droid disponibles en GitHub	Emulador Android 10 API 29
Stoat[40]	Cobertura del modelo	Se asignaron 3 horas por aplicación	Monkey y sapienz.	Se aplicó en las aplicaciones más populares de Google Play	Stoat se ejecutó en 3 máquinas físicas

2.1 Herramientas del estado del arte

2.1.1. Dynodroid

Dynodroid representa una solución innovadora para la generación de entradas relevantes en aplicaciones Android sin necesidad de modificarlas. Esta herramienta aborda el desafío de probar aplicaciones Android al considerarlas como programas orientados a eventos, que interactúan con su entorno por medio de una secuencia de eventos a través del marco de trabajo de Android. Dynodroid monitoriza la reacción de la aplicación ante cada evento de manera eficiente y liviana. Esta información se utiliza para guiar la generación del próximo evento de la aplicación. Dynodroid también se destaca por su capacidad para combinar eventos generados por máquinas, que son mejores generando una gran cantidad de entradas, con eventos generados por seres humanos, que a su vez son mejores generando textos de entrada mas inteligentes, permitiendo un enfoque más completo y efectivo para las pruebas.[35]

El algoritmo de Dynodroid (ver algoritmo 1) es la columna vertebral de esta herramienta. De forma similar al flujo lógico mostrado en la figura 1.1, Dynodroid observa los posibles eventos a ejecutar, selecciona los más relevantes y los ejecuta. Para la selección del próximo evento a ejecutar, se penalizan aquellas interacciones que han sido seleccionadas previamente con mayor frecuencia. Este enfoque permite que Dynodroid evalúe la aplicación en diferentes estados y genere pruebas exhaustivas, contribuyendo a una evaluación efectiva de la aplicación. Dynodroid está disponible en GitHub [9].

Algoritmo 1: Algoritmo de Dynodroid

```
Data: Número n >0 de eventos a generar
Result: Lista L de n eventos
L := lista vacía;
e := evento para instalar y ejecutar la aplicación bajo prueba;
s := estado del programa inicial;
for i de 1 a n do
    Agregar e a L;
    s := ejecutor(e, s);
    E := observador(s);
    e := selector(E);
```

2.1.2. Sapienz

Sapienz es una herramienta diseñada para aplicaciones Android con el objetivo de minimizar la longitud de las secuencias de prueba y maximizar la cobertura y la detección de errores. Su estrategia se basa en una combinación de técnicas que incluyen el fuzzing aleatorio, la exploración sistemática y la búsqueda multiobjetivo. El fuzzing aleatorio permite a Sapienz generar entradas inválidas, inesperadas o aleatorias con el objetivo de descubrir vulnerabilidades o errores en las aplicaciones. Para mejorar su efectividad, Sapienz incorpora técnicas de exploración sistemática y basadas en búsqueda, que tienen en cuenta la estructura de la aplicación, la cobertura del código y patrones de comportamiento para guiar la exploración. La herramienta utiliza interacciones basadas en eventos. Es importante destacar que Sapienz se ha descontinuado desde agosto de 2017, pero su código fuente sigue disponible en GitHub [18].

El algoritmo 2 muestra el funcionamiento de Sapienz. Se comienza instrumentando la aplicación bajo prueba, para poder acceder a métricas como la cobertura de código. Sapienz utiliza un algoritmo evolutivo que utiliza optimización multiobjetivo, donde se comienza con una población inicial de secuencias de prueba generadas aleatoriamente y luego utiliza operadores de selección, cruce y mutación para generar nuevas secuencias. Las secuencias generadas son evaluadas utilizando una

Algoritmo 2: Algoritmo de Sapienz

```
Data: AUT A, probabilidad de cruce p, probabilidad de mutación q, generación máxima
      gmax, tiempo de ejecución t
Result: Modelo UI M, frente de Pareto PF, informes de prueba C
M ← K0; PF ← ∅; C ← ∅; ▷inicialización
generación g ← 0;
iniciar dispositivos D; ▷preparar ejecutor de aplicaciones
inyectar MotifCore en D; ▷para exploración híbrida
análisis estático en A; ▷para siembra de cadenas
instrumentar e instalar A;
inicializar población P; ▷hibrida de genes aleatorios y de motivo
evaluar P con MotifCore y actualizar (M, PF, C);
while g <gmax y ¬ tiempo de espera(t) do
    g ← g+1;
    Q ← variación de toda la suite de pruebas(P, p, q); ▷ver Algoritmo 2
    evaluar Q con MotifCore y actualizar (M, PF, C);
    F ← ∅; ▷frentes no dominados
    F ← ordenarNoDominados(P ∪ Q, |P|);
    P′ ← ∅; ▷individuos no dominados
    foreach frente F en F do
        if |P′| ≥ |P| then
            romper;
        calcular distancia de agrupamiento para F;
        foreach individuo f en F do
            P′ ← P′ ∪ f;
        P′ ← ordenado(P′, ≺c); ▷ver ecuación 3 para el operador ≺c
        P ← P′[0 : |P|]; ▷nueva población
return (M, PF, C);
```

función de *fitness* que mide la cobertura de código y la capacidad para descubrir errores. El algoritmo se repite hasta alcanzar la generación máxima o el tiempo límite.

2.1.3. AutoDroid

AutoDroid [38] es una herramienta para aplicaciones Android que se ejecutan en entornos sensibles y que deben responder a cambios en dicho entorno, como cambios en la conectividad de la red, nivel de batería, orientación de la pantalla, entre otros. La gran cantidad de eventos de interfaz de usuario a menudo dificulta el proceso de testing. Autodroid fue extendido posteriorment [21] permitiendo la generación automática de pruebas guiadas por la cobertura de pares intercalados de secuencias de eventos GUI y eventos de contexto. De manera sistemática, se entrelazan eventos de contexto y GUI en las pruebas utilizando el algoritmo de pares intercalados.

AutoDroid (ver algoritmo 3), genera sistemáticamente una suite de pruebas al seleccionar contextos iniciales y eventos GUI, ejecutando eventos y registrando el estado de la GUI. AutoDroid mantiene un historial de combinaciones de eventos como parte del proceso de construcción de la suite de pruebas y utiliza la información histórica para seleccionar y ejecutar eventos que maximizan la cobertura de n-tuplas de eventos, donde n es una fuerza de combinación de eventos especificada.

Algoritmo 3: Algoritmo de AutoDroid

```
Data: Paquete de aplicación Android (AUT), modelo de contexto combinatorio (M),
      estrategias de selección de contexto inicial, selección de eventos, criterios de
      terminación y finalización de pruebas
Result: Suite de pruebas (T)
Call := generar matriz de cobertura de contexto a partir de M;
T := conjunto vacío de pruebas;
repeat
    ti := caso de prueba vacío;
    Ccurr := EstrategiaInicialContexto(Call);
    agregar evento de contexto inicial, Ccurr, al caso de prueba ti;
    instalar y ejecutar AUT, agregar evento de inicio al caso de prueba ti;
    scurr := estado GUI inicial;
    while no se cumple el criterio de terminación do
        Eall := eventos GUI en el estado GUI actual scurr;
        esel := EstrategiaSeleccionEvento(Scurr, Eall, Call);
        ejecutar esel;
        agregar esel al caso de prueba ti;
        scurr := estado GUI actual;
    agregar ti a T;
    finalizar caso de prueba (limpiar caché/SD, desinstalar la aplicación, etc.);
until se cumple el criterio de finalización;
return T;
```

2.1.4. DroidBot

DroidBot es una herramienta ligera para la generación de entradas de prueba guiadas por la interfaz de usuario (UI) en aplicaciones de Android. Su enfoque principal es explorar la interfaz gráfica de las aplicaciones y generar acciones de prueba de manera automatizada.

DroidBot utiliza técnicas basadas en modelos y exploración dinámica para generar acciones de prueba. Identifica las acciones mediante la exploración de los elementos y las interacciones de la interfaz gráfica de la aplicación, lo que incluye clics, ingreso de texto, desplazamientos, deslizamientos y otros gestos o interacciones.

La herramienta se ha comparado con varias herramientas en experimentos para evaluar la calidad del software. Algunas de las herramientas comparadas incluyen Monkey, AndroidRipper, DynoDroid, SwiftHand, PUMA, DroidMate y DroidBot mismo. Estos experimentos se llevaron a cabo tanto en emuladores como en dispositivos reales, lo que demuestra la versatilidad de DroidBot en entornos de prueba.

DroidBot proporciona métricas de cobertura de código y detección de fallas para evaluar la efectividad del proceso de prueba. La herramienta está disponible públicamente en GitHub como un proyecto de código abierto [7], y ha recibido commits recientes hasta abril de 2023, lo que indica un desarrollo activo y mantenimiento continuo. La arquitectura general de DroidBot se muestra en la Figura 2.1.

2.1.5. Humanoid

Humanoid es una herramienta de prueba de caja negra automatizada para aplicaciones de Android, basada en enfoques de aprendizaje profundo. El enfoque principal de Humanoid es identificar fallas, problemas de rendimiento y vulnerabilidades de seguridad en las aplicaciones mediante el análisis dinámico y la simulación de interacciones de usuario.

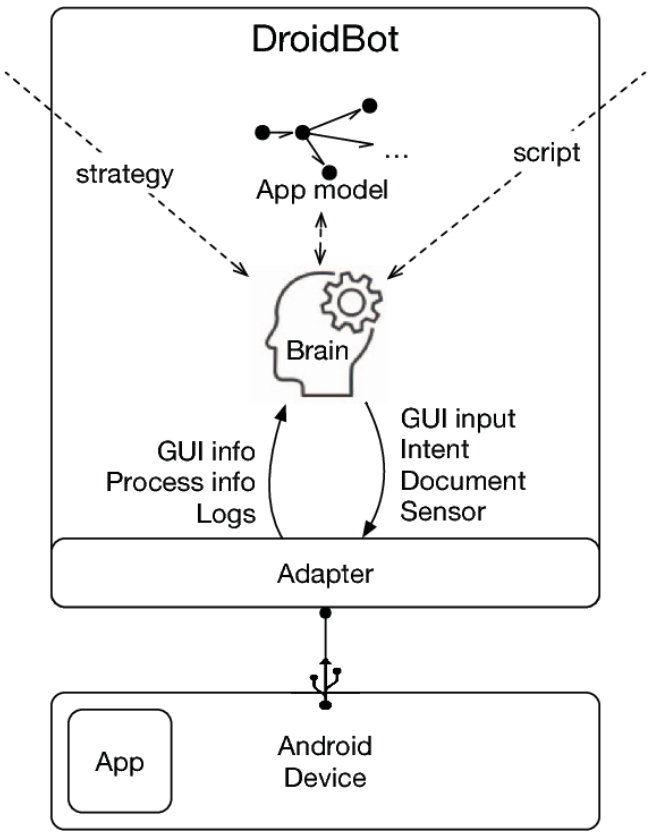

Figura 2.1: DroidBot:Descripción general
[31]

La herramienta está disponible públicamente en GitHub [13] como un proyecto de código abierto, con el último commit realizado en febrero de 2023, lo que indica un desarrollo activo y mantenimiento continuo.

Humanoid utiliza técnicas de análisis estático y dinámico, así como aprendizaje profundo, para analizar el comportamiento y las respuestas de la aplicación de Android durante la fase de análisis dinámico. Identifica acciones mediante la simulación de entradas de usuario y la interacción con la aplicación, lo que incluye acciones como clics, presiones largas, deslizamientos, entrada de texto, pulsaciones de botones y otras interacciones de la interfaz de usuario.

La herramienta se enfoca en métricas relacionadas con la detección de fallas, problemas de rendimiento y vulnerabilidades de seguridad en las aplicaciones de Android probadas. Se han realizado experimentos comparativos con otras herramientas, como Monkey, PUMA, Stoat, DroidMate y DroidBot, aunque no se especifica en el extracto cuáles fueron los resultados de dichos experimentos.

En cuanto a la selección de aplicaciones, Humanoid se probó en 68 aplicaciones de código abierto obtenidas de AndroTest y 200 aplicaciones comerciales populares descargadas de Google Play.

En resumen, Humanoid es una herramienta de prueba de caja negra automatizada para aplicaciones de Android que utiliza enfoques de aprendizaje profundo y análisis estático y dinámico. Su objetivo es identificar fallas, problemas de rendimiento y vulnerabilidades de seguridad en las aplicaciones probadas. Con su enfoque en el análisis del comportamiento de la aplicación y la simulación de interacciones de usuario, Humanoid ofrece una forma eficaz de mejorar la calidad y la seguridad del software móvil. Se muestra la Arquitectura del modelo de interacción de Humanoid en la Figura 2.2.

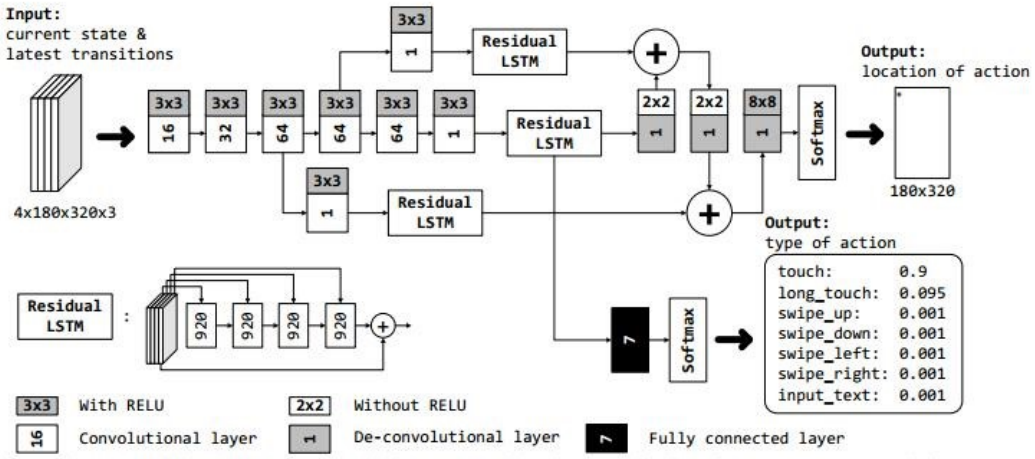

Figura 2.2: Arquitectura del modelo de interacción de Humanoid

2.1.6. RegDroid

RegDroid [17] se erige como una herramienta innovadora y fundamental en el panorama de las pruebas de aplicaciones Android. Su singularidad radica en su enfoque enfático en la cobertura de código y la detección de fallas, combinando diversas técnicas, tales como pruebas GUI aleatorias, pruebas basadas en modelos, búsquedas y aprendizaje automático.

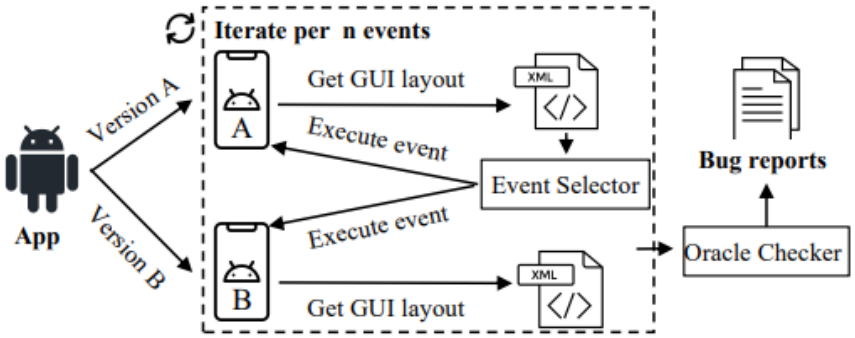

Figura 2.3: Arquitectura del modelo de interacción de RegDroid [45]

La versatilidad de RegDroid se pone de manifiesto al permitir la ejecución de pruebas tanto en emuladores como en dispositivos reales, lo que la convierte en una solución adaptable y adaptable a una amplia gama de escenarios de prueba.

Los estudios de investigación [45] que han involucrado a RegDroid han arrojado resultados prometedores, validando su capacidad para identificar de manera efectiva errores y problemas en aplicaciones móviles. Estos estudios se han realizado utilizando un conjunto de aplicaciones Android de código abierto, ampliamente reconocidas y representativas, lo que refuerza su confiabilidad y eficacia en el ámbito de las pruebas de aplicaciones.

En un contexto más amplio, RegDroid se erige como un activo crucial para elevar la calidad y la confiabilidad de las aplicaciones móviles en el competitivo mercado actual[17]. Su capacidad para realizar pruebas minuciosas, gracias a una amplia gama de técnicas, puede tener un impacto significativo en la satisfacción del usuario y la reputación de las empresas desarrolladoras de aplicaciones.

En síntesis, RegDroid emerge como una herramienta esencial en la investigación y el desarrollo de aplicaciones Android, brindando una solución completa y eficiente para elevar la calidad y confiabilidad de las aplicaciones en un entorno móvil cada vez más exigente.

App Name	#Stars	ID	Bug State	New Bug?	Bug Symptom
AnkiDroid	5.4K	1	Fixed	Yes (#12053)	Incorrect interaction logic
		2	Fixed	Yes (#11220)	UI element does not react
		3	Fixed	No	Incorrect interaction logic
		4	Fixed	Yes (#11363)	UI element does not react
Amaze	4.0K	5	Fixed	Yes (#3378)	Missing UI elements
		6	Fixed	Yes (#3394)	Redundant UI elements
		7	Fixed	No	Missing UI elements
		8	Fixed	No	Redundant UI elements
AntennaPod	4.4K	9	Fixed	Yes (#5977)	Missing UI elements
		10	Fixed	Yes (#5863)	UI Element does not react
Markor	2.4K	11	Fixed	Yes (#1800)	Functionality does not take effect
Omni-Notes	2.5K	12	Fixed	Yes (#865)	UI Element does not react
		13	Fixed	Yes (#867)	Functionality does not take effect
		14	Fixed	No	Missing UI elements

Figura 2.4: Resultados de RegDroid
[45]

2.1.7. ComboDroid

ComboDroid [44] es una destacada herramienta en el ámbito de la generación de entradas de prueba para aplicaciones Android. Su enfoque principal radica en la mejora de la cobertura de código mediante la generación sistemática de combinaciones específicas de casos de uso y escenarios de prueba. Esta estrategia se basa en aprovechar combinaciones de casos de uso, lo que permite una evaluación exhaustiva de la aplicación bajo prueba.

La generación de acciones en ComboDroid se realiza mediante técnicas de exploración de la interfaz gráfica de usuario (GUI).

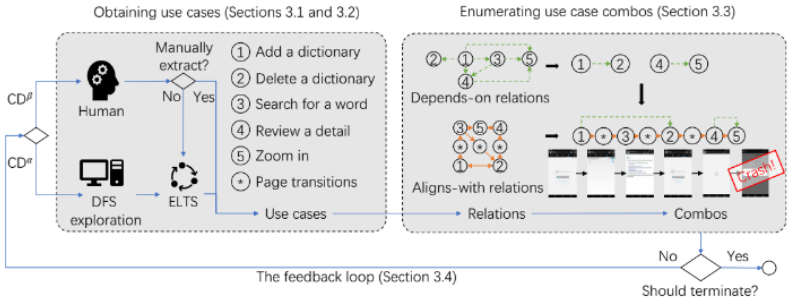

Figura 2.5: Arquitectura del modelo de interacción de Combodroid
[44]

Entre las acciones que puede identificar y ejecutar se encuentran clics, deslizamientos y entradas de texto. Cabe destacar que la generación de texto de entrada se realiza de manera sistemática al seleccionar combinaciones específicas de casos de uso o escenarios de prueba. Este enfoque no solo garantiza una amplia cobertura de código sino que también facilita la detección de posibles fallos y problemas en la aplicación.

ComboDroid ha sido objeto de experimentos comparativos con otras herramientas de pruebas automatizadas para aplicaciones Android, como Monkey, Sapienz y APE. Estos experimentos se llevaron a cabo utilizando un entorno de emulación de Android. Como resultado, ComboDroid demostró su eficacia en la generación de pruebas de alta calidad que mejoran significativamente la cobertura de código en las aplicaciones evaluadas.

Para aquellos interesados en explorar y utilizar ComboDroid, su código fuente está disponible en GitHub[5], lo que permite a la comunidad de desarrollo acceder y contribuir a esta herramienta de pruebas. El compromiso continuo con el desarrollo se refleja en el último commit realizado el 26 de mayo de 2021, lo que subraya el compromiso de mantener y mejorar esta herramienta en evolución.

Algoritmo 4: Aalgoritmo de Combodroid

```
1: function EJECUTARP(E)
2:     G ← ObtenerGUI()
3:     L ← [], M ← ∅, T ← ∅ for cada e ∈ E do
       r() ⊥                                              ▷ e puede ser enviado
4:     M ← EnviarEventoAAplicacion(e.t, e.r(), e.z)       ▷ Enviar evento e a P, esperar un estado
   quiescente y obtener la secuencia de invocación del método correspondiente
5:     M ← M :: M
6:     T ← T ∪ {e, M}
7:     G ← ObtenerGUI()
8:     L ← L :: [] else
9:
        return ⊥
10:
11:
12:    return L, M, T
13: end function
```

2.1.8. TimeMachine

TimeMachine [24] es una herramienta que permite realizar pruebas de *viaje en el tiempo* en aplicaciones Android. Está disponible públicamente en GitHub y ha recibido su último commit en julio de 2022, lo que indica un desarrollo activo. TimeMachine utiliza una combinación de técnicas de prueba, como pruebas aleatorias, pruebas basadas en modelos, pruebas basadas en búsqueda y en aprendizaje.

La herramienta emplea oráculos que realizan comparaciones con datos de referencia conocidos para evaluar la corrección del comportamiento de la aplicación. El estado de la aplicación puede representarse mediante el árbol de widgets o una captura de pantalla de la interfaz de usuario en un momento específico. Para interactuar con la aplicación, TimeMachine analiza las opciones de interacción disponibles proporcionadas por la interfaz de usuario de la aplicación, como clics, desplazamientos e introducción de texto. Se pueden utilizar tanto textos aleatorios como textos predefinidos como entrada durante el proceso de prueba. La herramienta se centra en medir la cobertura de código y detectar fallos dentro de la aplicación.

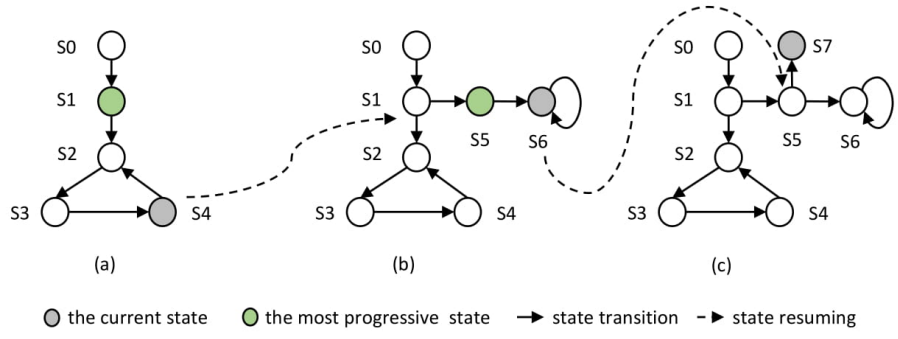

Figura 2.6: Arquitectura del modelo de interacción de TimeMachine [24]

En los experimentos realizados, TimeMachine se comparó con otras herramientas como Monkey, Sapienz y Stoat. La selección de aplicaciones incluyó las 100 aplicaciones instrumentadas más populares de Google Play.

En cuanto al entorno de prueba, TimeMachine se ejecutó en dos máquinas físicas con 64 GB de memoria principal, utilizando un sistema operativo Ubuntu 16.04 de 64 bits. Esto indica que se puede

utilizar en dispositivos reales para realizar pruebas. En resumen, TimeMachine ofrece un enfoque integral para las pruebas de aplicaciones Android, aprovechando diversas técnicas y comparaciones con datos de referencia para mejorar la calidad del software.

2.1.9. QTesting

Q-testing es un enfoque basado en el aprendizaje por refuerzo para pruebas de GUI en aplicaciones de Android. Combina el aprendizaje por refuerzo y las pruebas basadas en modelos para generar casos de prueba automatizados y evaluar la calidad de la interfaz de usuario gráfica.

En Q-testing, la GUI de una aplicación de Android se representa como un estado, y las acciones disponibles en los elementos de la GUI se definen como las acciones que se pueden realizar. El objetivo es encontrar una secuencia óptima de acciones que maximice los criterios de calidad predefinidos, como la cobertura de código, la detección de fallos o la cobertura de interacciones con el usuario.

El enfoque de aprendizaje por refuerzo en Q-testing implica entrenar a un agente de aprendizaje utilizando el algoritmo Q-Learning. Este agente aprende una política que guía sus decisiones al estimar la utilidad esperada de cada acción en un estado dado. Durante las pruebas, el agente interactúa con la GUI de la aplicación, seleccionando acciones en función del estado actual y los valores almacenados en una tabla Q.

A medida que el agente realiza pruebas, recibe recompensas o penalizaciones según el resultado de las acciones seleccionadas. Estas recompensas se utilizan para actualizar la tabla Q y mejorar la política del agente con el tiempo.

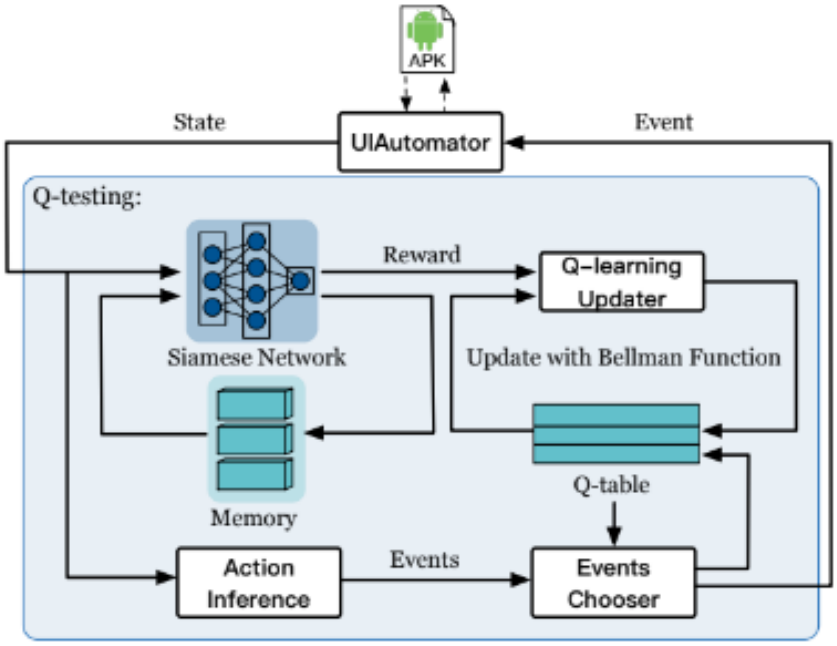

Figura 2.7: Arquitectura del modelo de interacción de Qtesting [37]

En resumen, Q-testing utiliza el aprendizaje por refuerzo para automatizar las pruebas de GUI en aplicaciones de Android. Es un enfoque prometedor que puede mejorar la eficiencia y la calidad de las pruebas al encontrar secuencias óptimas de acciones en función de los criterios de calidad establecidos.

En los experimentos realizados con Q-testing, se comparó su rendimiento con otras herramientas de pruebas como Monkey, Sapienz y Stoat. Se utilizaron tanto aplicaciones de código abierto con Monkey, Stoat y Sapienz, así como aplicaciones de código cerrado con Monkey y Sapienz. La selec-

ción de las aplicaciones se realizó a partir de una muestra de las 50 aplicaciones de Android más utilizadas en Google Play.

Para llevar a cabo los experimentos, se utilizó un dispositivo real que consistía en una máquina física con un procesador de 4 núcleos, 3.60GHz y 16GB de RAM, ejecutando Ubuntu 16.04. Se evaluaron métricas como la cobertura de código y la detección de fallos para analizar el rendimiento de Q-testing en comparación con las otras herramientas.

Los resultados de los experimentos mostraron que Q-testing logró obtener una alta cobertura de código y una detección efectiva de fallos en las aplicaciones probadas. En comparación con las otras herramientas, Q-testing demostró ser competitivo e incluso superó en algunos aspectos, destacando su capacidad para generar casos de prueba basados en aprendizaje por refuerzo.

2.1.10. AimDroid

AimDroid es un marco de pruebas automatizado diseñado para aplicaciones de Android. Utiliza una variedad de técnicas de prueba, como pruebas aleatorias, pruebas basadas en modelos, análisis dinámico y análisis de cobertura, para evaluar la calidad del software. AimDroid se centra en la exploración de la interfaz gráfica y la detección de problemas, como fallos y vulnerabilidades. Selecciona acciones de usuario, como clics, escritura y desplazamiento, para interactuar con la interfaz de usuario durante el proceso de prueba. Además, realiza análisis estático y dinámico para examinar el árbol de widgets y capturar capturas de pantalla.

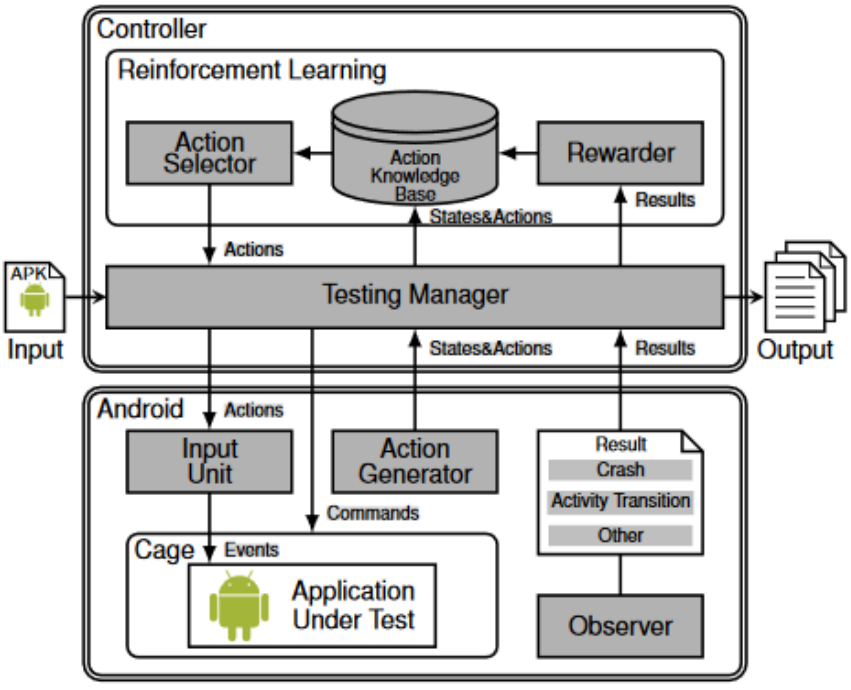

Figura 2.8: Arquitectura del modelo de interacción de Aimdroid [29]

AimDroid ha sido evaluado en experimentos comparativos con herramientas como Monkey y SAPIENZ, utilizando una variedad de aplicaciones comerciales populares. Los experimentos se llevaron a cabo en un dispositivo Android real y un entorno de emulación en un PC de alto rendimiento. Los resultados de los experimentos proporcionan una visión general de la calidad del software y su comportamiento en diferentes herramientas de prueba.

2.1.11. APE

APE (Android Automated GUI Test Generation and Execution) es una herramienta de código abierto diseñada específicamente para pruebas automatizadas de aplicaciones de Android. Se centra en generar y ejecutar casos de prueba que interactúan con la interfaz gráfica de usuario (GUI) de la aplicación. APE utiliza una combinación de técnicas de prueba basadas en modelos y pruebas aleatorias para explorar la GUI, generar casos de prueba y evaluar el comportamiento de las aplicaciones de Android.

APE ofrece características y capacidades clave, como pruebas basadas en modelos, generación de eventos, análisis de cobertura, pruebas aleatorias, ejecución de pruebas y parámetros configurables.

Algorithm 1: Model construction.

Input: Testing budget B, initial abstraction function $\mathcal{L}$.
Output: The model M.

1 $(M, S, \pi) \leftarrow ((\varnothing, \varnothing, \varnothing, \mathcal{L}), \varnothing, \varnothing)$ ▷ *Initialization.*
2 **while** $B > 0$ **do**
3 $B \leftarrow B - 1$ ▷ *Decrease the testing budget.*
4 $T' \leftarrow$ `CaptureGUITree()` ▷ *Use* `uiautomator`.
5 $M \leftarrow$ `UptAndOptModel`(M, S, π, T') ▷ *See Algorithm 2.*
6 $S \leftarrow \mathbb{L}_{\mathcal{L}}(T')$ ▷ *Get the current state.*
7 $\pi \leftarrow$ `SelectAndSimulateAction`(S) ▷ *See Section III-D.*
8 **return** M

Figura 2.9: APE: Algoritmo 1
[30]

Con la ayuda de estas funciones, APE puede explorar de manera sistemática los diferentes caminos e interacciones dentro de la aplicación, generar eventos relevantes para simular interacciones de usuario, medir la cobertura de código, ejecutar pruebas y capturar resultados, y ajustar los parámetros de prueba según las necesidades específicas.

APE es una herramienta versátil que se puede utilizar tanto para pruebas manuales como automatizadas de aplicaciones de Android. Al automatizar el proceso de pruebas, APE ayuda a mejorar la eficiencia y efectividad de las pruebas de aplicaciones de Android.

2.1.12. DroidMate-2

DroidMate-2 [23]es una plataforma diseñada para la generación de pruebas en aplicaciones de Android. Utiliza técnicas de prueba basadas en modelos, pruebas aleatorias y pruebas sistemáticas para explorar y evaluar el comportamiento de las aplicaciones.

Una de las características principales de DroidMate-2 es su capacidad para identificar acciones a partir de la exploración de los elementos de la interfaz gráfica y las interacciones de la aplicación. Puede realizar acciones como clics, deslizamientos, entradas de texto y gestos para simular el comportamiento del usuario.

La herramienta utiliza métricas como la cobertura de código y la detección de fallas para evaluar la calidad de las pruebas generadas. Se han realizado comparaciones con herramientas como DroidBot y Monkey en experimentos para evaluar la calidad del software.

En los experimentos, DroidMate-2 se probó con 11 aplicaciones de Android seleccionadas al azar. La plataforma es compatible tanto con dispositivos reales como con emuladores, y se ha utilizado un dispositivo real Google Nexus 5X en los experimentos.

DroidMate-2 proporciona una plataforma versátil[8] para la generación de pruebas en aplicaciones de Android. Su capacidad para utilizar diferentes técnicas de prueba y explorar la interfaz

Algorithm 2: Update and optimize the model.

Function UptAndOptModel($M = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{L}), S, \pi, T'$)
  Input: The new GUI tree T', the model $M = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{L})$, the previous state S, and the previous action π.
  Output: The new model.
  $S' \leftarrow \mathbb{L}_{\mathcal{L}}(T')$
  $M \leftarrow (\mathcal{S} \cup \{S'\}, \mathcal{A} \cup S', \mathcal{T} \cup \{(S, \pi, S')\}, \mathcal{L})$
  repeat $M \leftarrow$ ActionRefinement(M, T') **until** M *is not updated*
  $M \leftarrow$ StateCoarsening(M, T')
  return StateRefinement$(M, (S, \pi, S'))$
Function ActionRefinement$(M = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{L}), T')$
  foreach $\pi' \in \mathbb{L}_{\mathcal{L}}(T')$ **do**
    if $|\overline{\mathcal{L}}(\pi') \cap T'| > \alpha$ **then**
      $R \leftarrow$ GetReducer(π')
      foreach $R' \in \{R' | R' \in \mathbb{R} \wedge R \not\sqsubseteq R'\}$ **do**
        $\mathcal{L}' \leftarrow \mathcal{L} \cup \{R \rightarrow R'\} \mid \mathcal{L} \cup \{(R, \pi', R')\}$
        $\Pi \leftarrow \{\mathcal{L}'(\sigma) | \sigma \in \overline{\mathcal{L}}(\pi') \cap T'\}$
        if $|\Pi| > 1$ **then**
          return RebuildModel$(M, \{\mathbb{L}_{\mathcal{L}}(T')\}, \mathcal{L}')$
  return M
Function StateCoarsening$(M = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{L}), T')$
  $\mathcal{L}' \leftarrow$ GetPrev$(\mathcal{L})$
  $\mathbb{S} \leftarrow \{\mathbb{L}_{\mathcal{L}}(T) | T \in \overline{\mathbb{L}_{\mathcal{L}'}}(\mathbb{L}_{\mathcal{L}'}(T'))\}$
  if $|\mathbb{S}| > \beta$ **then return** RebuildModel$(M, \mathbb{S}, \mathcal{L}')$ **else return** M
Function StateRefinement$(M = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{L}), (S, \pi, S'))$
  foreach $(S, \pi, S'') \in \{(S, \pi, S'') | (S, \pi, S'') \in \mathcal{T} \wedge S'' \neq S'\}$ **do**
    foreach $\pi' \in S$ **do**
      $R \leftarrow$ GetReducer(π')
      foreach $R' \in \{R' | R' \in \mathbb{R} \wedge R \not\sqsubseteq R'\}$ **do**
        $\mathcal{L}' \leftarrow \mathcal{L} \cup \{R \rightarrow R'\} \mid \mathcal{L} \cup \{(R, \pi', R')\}$
        $\mathbb{S}_1 \leftarrow \{\mathbb{L}_{\mathcal{L}'}(T) | (T, \sigma, T') \in \overline{\Sigma_{\mathcal{L}'}}((S, \pi, S'))\}$
        $\mathbb{S}_2 \leftarrow \{\mathbb{L}_{\mathcal{L}'}(T) | (T, \sigma, T'') \in \overline{\Sigma_{\mathcal{L}'}}((S, \pi, S''))\}$
        if $\mathbb{S}_1 \cap \mathbb{S}_2 = \varnothing$ **then**
          return RebuildModel$(M, \mathbb{S}, \mathcal{L}')$
  return M

Figura 2.10: APE: Algoritmo 2
[30]

gráfica de las aplicaciones ayuda a mejorar la calidad y la cobertura de las pruebas en el desarrollo de software para Android.

2.1.13. ARES

ARES es una herramienta de prueba de caja negra para aplicaciones de Android que utiliza el Aprendizaje Profundo por Reforzamiento (Deep Reinforcement Learning). Su objetivo es mejorar la calidad y efectividad de las pruebas mediante la exploración de la interfaz gráfica de las aplicaciones.

ARES utiliza técnicas de prueba aleatorias, pruebas basadas en modelos y aprendizaje profundo para generar y ejecutar acciones en las aplicaciones. Identifica las acciones a través de la exploración de los elementos y las interacciones de la interfaz gráfica de la aplicación, lo que incluye clics, ingreso de texto, desplazamientos, deslizamientos y otros gestos o interacciones.

La herramienta se ha comparado con otras herramientas, como FATE, Monkey, Sapienz, TimeMachine y Q-Testing, en experimentos para evaluar la calidad del software. En los experimentos, ARES

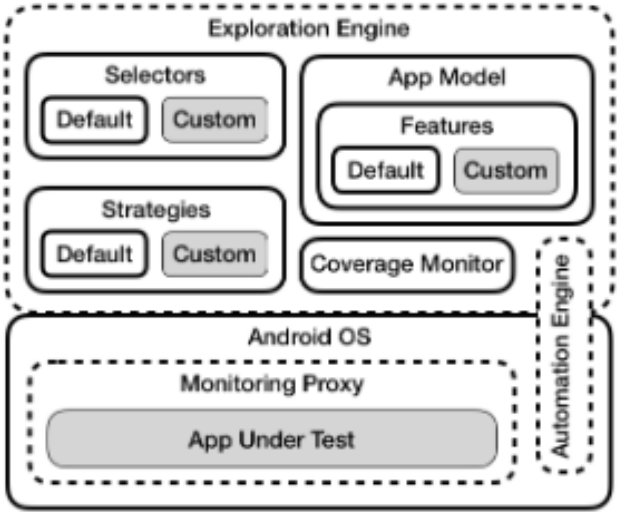

Figura 2.11: Arquitectura del modelo de interacción de Droidmate2 [23]

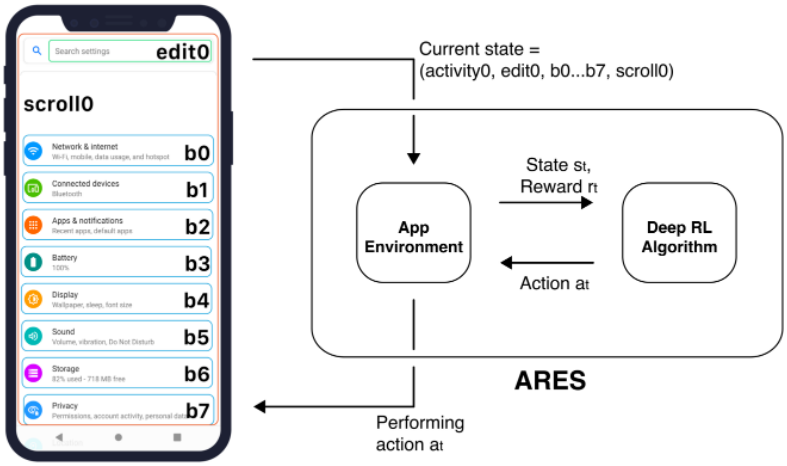

Figura 2.12: ARES [39]

se probó con 100 aplicaciones seleccionadas al azar de entre las 500 aplicaciones más destacadas de F-Droid disponibles en GitHub.

Para los experimentos, se utilizaron emuladores equipados con 2 GB de RAM y Android 10.0 (API Nivel 29) o Android 4.4, específicamente para la comparación de herramientas.

ARES se ha desarrollado como un proyecto de código abierto y está disponible públicamente en GitHub. Ha habido commits recientes hasta abril de 2023, lo que indica un desarrollo y mantenimiento continuo de la herramienta.

En resumen, ARES es una herramienta de prueba innovadora que utiliza aprendizaje profundo por reforzamiento para mejorar la calidad y efectividad de las pruebas de aplicaciones de Android. Con su enfoque en la exploración de la interfaz gráfica y su capacidad para generar acciones de manera inteligente, ARES muestra un gran potencial para el campo de las pruebas de software móvil

2.1.14. Stoat

Stoat es una herramienta de prueba de GUI basada en modelos,[40] guiada y estocástica, diseñada específicamente para aplicaciones de Android. Su enfoque principal es la identificación de vulnera-

bilidades de seguridad en las aplicaciones, mediante la combinación de técnicas de análisis estático y exploración dinámica.

La herramienta se encuentra disponible públicamente en GitHub [20] como un proyecto de código abierto, con commits recientes hasta abril de 2023, lo que demuestra un desarrollo activo y mantenimiento continuo.

Stoat utiliza el análisis estático para identificar rutas de código sensibles a la seguridad en la aplicación. Luego, genera casos de prueba que ejercen estas rutas identificadas durante la fase de análisis estático.[27] Estas acciones pueden incluir el envío de solicitudes de red, el acceso a datos sensibles, la activación de llamadas a funciones específicas o la manipulación de permisos dentro de la aplicación.

La herramienta se centra en métricas como la cobertura de código, la cobertura del modelo y la diversidad de pruebas para evaluar la calidad del software. Se han realizado experimentos comparativos con otras herramientas, aunque no se especifica cuáles en el extracto proporcionado.

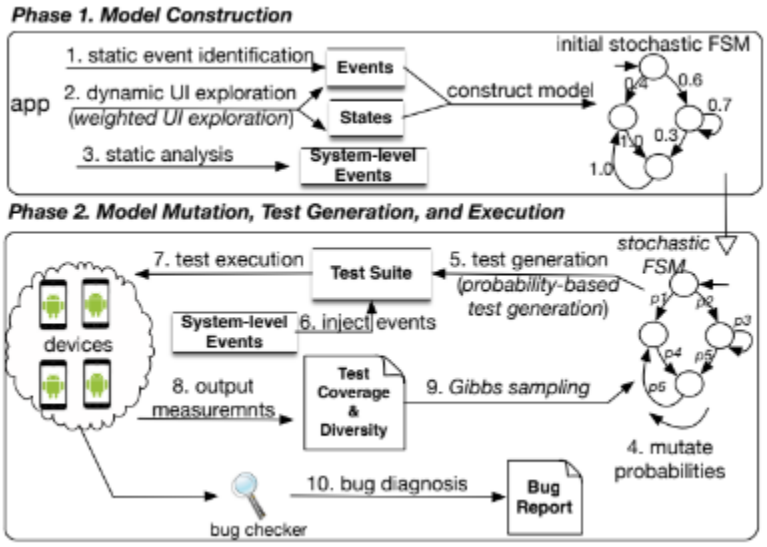

Figura 2.13: ARES
[40]

En cuanto a la selección de aplicaciones, Stoat se aplicó en las aplicaciones más populares de Google Play. Además, la herramienta se ejecutó en tres máquinas físicas con 18 emuladores y 6 dispositivos reales, lo que indica la capacidad de Stoat para funcionar en entornos tanto de emuladores como de dispositivos reales.

En resumen, Stoat es una herramienta de prueba de GUI basada en modelos, guiada y estocástica, que se enfoca en la identificación de vulnerabilidades de seguridad en aplicaciones de Android. Con su enfoque en el análisis estático y la exploración dinámica, Stoat ofrece una forma eficiente de mejorar la calidad y la seguridad del software móvil.

2.2 Observaciones del Estado del Arte

Se destacan las siguientes observaciones clave a partir del análisis exhaustivo de las herramientas de prueba de GUI en Android:

- **Eficiencia en la Detección de Fallos:** Se observa que herramientas como Dynodroid, Sapienz y AutoDroid han demostrado ser efectivas en la detección de fallos en la interfaz gráfica de usuario (GUI) de aplicaciones Android. Estas herramientas han sido ampliamente utilizadas en investigaciones anteriores.

- **Estabilidad y Compatibilidad:** A pesar de sus capacidades, se ha observado que algunas de las herramientas, como Sapienz, enfrentan problemas de compatibilidad debido a su dependencia en Python 2.7, una versión descontinuada. Con la transición de Python hacia la versión 3, el código de Sapienz se ha vuelto incompatible con las nuevas versiones y presenta dificultades durante la instalación y ejecución. Esto resalta la importancia de la estabilidad en términos de instalación y la compatibilidad con las últimas tecnologías, como se encuentra en herramientas como Dynodroid, AutoDroid, DroidBot y Humanoid. Esta característica es fundamental para garantizar la viabilidad y la confiabilidad de las pruebas en entornos de desarrollo móvil en constante evolución.

2.2.1. Problemas Encontrados en las Herramientas del Estado del Arte

A continuación, se detallan los problemas identificados en varias herramientas del estado del arte utilizadas para pruebas de GUI en Android:

- **Crawldroid:** Una limitación significativa de Crawldroid radica en su insuficiente documentación e instrucciones adecuadas en su repositorio de GitHub. La ausencia de un archivo 'README', así como la falta de instrucciones explícitas acerca de la instalación, implementación y ejecución de la herramienta, constituyen un obstáculo considerable. A pesar de la posibilidad de clonar el proyecto, la carencia de pautas detalladas y claras obstaculiza una utilización efectiva y eficiente de Crawldroid en contextos prácticos. [6]

- **Sapienz:** Enfrenta problemas de compatibilidad debido a su dependencia de Python 2.7, una versión obsoleta. Con la evolución de Python hacia la versión 3.11, el código de Sapienz se ha vuelto incompatible con las nuevas versiones, lo que dificulta su instalación y ejecución. Además, su integración con bibliotecas como matplotlib ha presentado dificultades en entornos modernos. En resumen, Sapienz se encuentra desactualizado y carece de compatibilidad con las versiones actuales de Python.[18]

- **Combodroid:** Combodroid enfrenta un problema común en programas escritos en Kotlin, la excepción "kotlin.KotlinNullPointerException". Esta excepción ocurre cuando se intenta acceder o utilizar un objeto que en realidad no tiene ningún valor asignado, es decir, es "nulo"(null). En Kotlin, se considera fundamental garantizar la seguridad contra excepciones de referencia nula. La excepción KotlinNullPointerException indica un problema en el código donde se intenta trabajar con un objeto nulo, lo que puede llevar a comportamientos inesperados o errores en tiempo de ejecución. Es esencial manejar adecuadamente esta excepción mediante técnicas como verificaciones de nulidad para asegurar la robustez y estabilidad de la aplicación.[5]

- **AimDroid:** AimDroid requiere el sistema operativo MacOS, lo que limita su disponibilidad a otras plataformas. Esto restringe su utilidad a aquellos que utilizan MacOS y excluye a quienes utilizan otros sistemas operativos.[29]

- **SynthesiSE:** Esta herramienta carece de un repositorio de GitHub o una página web para su instalación. Solo se encuentra disponible a través de su paper de investigación, lo que hace que su acceso y uso sean más complicados para los usuarios.[28]

- **Droidmate 2:** En el repositorio de GitHub de Droidmate 2, se hace referencia a una página web (www.droidmate.org) que normalmente debería proporcionar el software gratuito para su descarga e implementación. Sin embargo, se ha encontrado que esta página no está disponible, y los intentos de acceso han resultado en errores HTTP 404. Esto dificulta la obtención de la herramienta.[8]

- **Ares:** Ares presenta problemas de compatibilidad con Appium [3], que no es estable con los emuladores y funciona mejor con dispositivos físicos. Esto limita su eficacia y aplicabilidad en entornos de emulación.[39]

- **Stoat:** El problema principal con Stoat es su dependencia de la versión Ruby 2.1, que ha dejado de recibir soporte oficial desde el 31 de marzo de 2017. Esto significa que su código fuente necesita actualizaciones para ser compatible con versiones más nuevas de Ruby y para resolver errores fatales que surgen con la versión 2.1.[20]
- **TimeMachine:** TimeMachine requiere herramientas específicas como adb, aapt, avdmanager y un emulador configurado correctamente. Esto puede ser problemático, ya que la herramienta solo se conecta al emulador a través de adb. Los intentos de modificar el código fuente para lograr una conexión exitosa pueden generar otros errores debido a las múltiples dependencias involucradas.[24]

2.3 Selección de herramientas para el estudio comparativo

Después de una exhaustiva revisión de la literatura existente en el ámbito de las pruebas automatizadas en aplicaciones móviles Android, se han elegido cuidadosamente un conjunto de herramientas clave con que se va a comparar MINTESTAR.

Tabla 2.3: Herramientas Seleccionadas para Pruebas de GUI en Android

Nombre de la Herramienta	GitHub Link	Técnica de Prueba	Oráculos
Dynodroid	https://github.com/dynodroid/dynodroid	Pruebas dinámicas, Generación de entradas de texto	Oráculos visuales
AutoDroid	https://github.com/user1342/AutoDroid	Pruebas aleatorias, Pruebas basadas en modelos, Reinforcement Learning, Pruebas combinatorias	Detección de Fallos
DroidBot	https://github.com/honeynet/droidbot	Pruebas basadas en modelos, Exploración Dinámica	Detección de la Interfaz de Usuario
Humanoid	https://github.com/yzygitzh/Humanoid	Pruebas basadas en modelos, Deep Learning, Análisis Estático y Dinámico	Detección de la Interfaz de Usuario
RegDroid	https://shorturl.at/agiIJ	Pruebas basadas en modelos, basadas en búsquedas, basadas en aprendizaje	Detección de Fallos

Estas herramientas han sido identificadas como líderes en el estado del arte y, por lo tanto, representan opciones fundamentales para el estudio experimental. La selección de estas herramientas se basa en varios factores, incluida su capacidad para abordar las preguntas de investigación planteadas, su compatibilidad con el entorno de prueba, así como su reputación en la comunidad de desarrollo móvil. Después de leer la literatura y los artículos de todas las herramientas encontradas en el estado del arte, se ha pasado a la siguiente etapa, que consiste en probar cada herramienta para la selección final. Durante las pruebas, se encontraron errores y problemas técnicos con la mayoría de ellas, los cuales están detalladamente explicados en la sección anterior (2.2.1). Este factor es crucial en la elección, ya que tras las pruebas, solo 5 herramientas de un total de 14 no presentaron problemas técnicos, destacándose como las más adecuadas para la investigación.

- **Dynodroid**: Ha sido ampliamente utilizado en investigaciones anteriores, destacándose en experimentos como el realizado por **Sapienz** [36] y el estudio llevado a cabo por **Droidbot** [31].

- **AutoDroid**: Ha sido esencial en experimentos relevantes, incluyendo:
  - Experimento de **Monkey**[12].
  - Experimento de **CatDroid**[4].
- **Droidbot**: También ha tenido un impacto significativo en la comunidad de investigación, siendo utilizado en experimentos como **TESTAR**[32], y formando parte de investigaciones llevadas a cabo por **Droidmate2**[23] y **Humanoid**[34].
- **Humanoid**: Ha sido empleado en experimentos relevantes, incluyendo aquellos conducidos por **Droidbot** [31].
- **RegDroid**: Ha sido empleado en experimentos relevantes, incluyendo aquellos conducidos por **Stoat** [40].

La elección de herramientas adecuadas es esencial para garantizar la validez y la fiabilidad de los resultados experimentales, y cada una de las herramientas seleccionadas desempeñará un papel crucial en la evaluación comparativa que se llevará a cabo en este estudio. Además, estas herramientas se han destacado por su estabilidad en términos de instalación y su capacidad para mantenerse actualizadas con las tecnologías más recientes, lo que las hace aún más adecuadas para el estudio comparativo.

CAPÍTULO 3

MINTESTAR, o simplemente MINT

MINTESTAR, o MINT abreviado, representa una innovadora herramienta en el campo de las pruebas de aplicaciones móviles. Su enfoque principal se centra en abordar uno de los desafíos más cruciales en el desarrollo de software que es garantizar una experiencia excepcional para el cliente (CX). En este capítulo, profundizaremos en los detalles de MINT.

3.1 ¿Qué es MINT?

MINT es una herramienta de prueba diseñada para simplificar y optimizar el proceso de pruebas de aplicaciones móviles en dispositivos Android. Su enfoque se centra en automatizar la exploración de aplicaciones, lo que significa que MINT puede navegar y probar aplicaciones sin requerir una interacción constante de un ser humano para crear y mantener scripts de prueba. Esta característica es fundamental, ya que permite a los equipos de desarrollo y prueba abordar el desafío de probar de manera exhaustiva todas las interacciones que impactan la experiencia del cliente (CX).[15]

3.2 El Problema que Aborda

El desarrollo de aplicaciones móviles se ha convertido en un campo altamente competitivo y dinámico, donde la velocidad de lanzamiento y la calidad de la CX son cruciales. Sin embargo, uno de los desafíos más significativos que enfrentan los desarrolladores y probadores es la capacidad de explorar y probar exhaustivamente todas las interacciones que afectan la CX [15]. Las pruebas manuales y la creación de scripts de prueba convencionales a menudo resultan ser lentas y laboriosas, lo que limita la capacidad de realizar lanzamientos rápidos y efectivos.

MINT se erige como una solución disruptiva a este problema. Al automatizar la exploración de aplicaciones y aplicar reglas personalizables, MINT es capaz de detectar y priorizar interacciones de usuario, lo que permite una cobertura de prueba más amplia sin la necesidad de una interacción humana constante. Esta capacidad no solo acelera el proceso de prueba, sino que también mejora la calidad de la CX al identificar problemas críticos que, de otra manera, podrían pasar desapercibidos.

3.3 Funcionamiento de MINT

MINT opera utilizando tres componentes fundamentales: **pruebas** y **secuencias**, **reglas** y **oráculos** [15]. Estos elementos son cruciales para llevar a cabo pruebas automatizadas efectivas en aplicaciones Android.

3.3.1. Pruebas y Secuencias

En MINT, cada prueba se compone de múltiples secuencias, y cada secuencia está formada por una serie de pasos. Para comenzar, MINT ofrece una configuración predeterminada para las pruebas que incluye:

Número de secuencias: <3>

Número de pasos: <25>

Figura 3.1: Flujo de trabajo de MINT

Configurar pruebas en MINT implica tomar decisiones clave sobre qué reglas y oráculos deseas utilizar y cuáles deseas omitir. Estas decisiones determinarán cómo MINT interactúa con tu aplicación y cómo evalúa su comportamiento.[15]

3.3.2. Reglas

MINT interactúa con una aplicación siguiendo un conjunto de reglas predefinidas. Estas reglas definen qué interacciones son posibles y permitidas en la aplicación, como hacer clic en elementos, proporcionar información, etc. Las reglas tienen diferentes grados de importancia y, en conjunto, crean un modelo que guía a MINT a través de la aplicación para cubrir una amplia gama de estados. Esta metodología permite que MINT explore tu aplicación sin requerir conocimiento previo.

Las reglas pueden clasificarse en diversas categorías:

Reglas genéricas: Aplicables a cualquier aplicación (por ejemplo, interactuar con elementos clicables).

Reglas específicas para una categoría de aplicaciones: Diseñadas para aplicaciones de un tipo particular (por ejemplo, proporcionar direcciones de correo electrónico en un campo de texto contextualizado como correo electrónico).

Reglas específicas de dominio: Creadas para aplicaciones desarrolladas internamente y relacionadas con números de cuenta o identificadores internos.

Las categorías 1 y 2 suelen ser mantenidas y proporcionadas por la comunidad de MINT. Como usuario final, se tiene la opción de agregar reglas de la categoría 3 según sea necesario.

A continuación, se presentan algunas de las reglas predeterminadas de MINT:

Reglas de Navegación :

- Desplazarse y hacer clic en cualquier widget cliclable que aún no se haya mostrado y pueda desplazarse hasta él.
- Desplazar el paginador en una dirección.
- Despriorizar clics que se han realizado históricamente.
- Hacer clic en cualquier widget mostrado y cliclable

Reglas de Entrada

- Generar texto genérico para cualquier campo de texto.
- Generar texto en formato de dirección de correo electrónico para widgets que acepten direcciones de correo.
- Generar texto para widgets que acepten nombres de personas.
- Generar una fecha actual, futura o pasada para widgets que acepten fechas.
- Generar una dirección postal para widgets que acepten direcciones postales.
- Generar texto genérico (por ejemplo, UTF8, Unicode, ...) para cualquier tipo de entrada.

3.3.3. Oráculos

Después de cada interacción, MINT almacena el estado de la aplicación. Los oráculos observan estos estados y emiten veredictos. Un ejemplo de oráculo podría ser aquel que detecta texto superpuesto, una línea de registro de errores o un estado de agujero negro en la aplicación (un estado del que no se puede salir).

Oráculos Predeterminados

ESTABILIDAD: Este oráculo considera el registro del sistema Android como fuente de información.

RENDIMIENTO: El Oráculo de Dispositivo Android supervisa el uso de CPU, RAM, uso de red y gráficos (GPU).

ACCESIBILIDAD: Este oráculo detecta descripciones de contenido faltantes en iconos e imágenes.

Estos componentes son vitales para la operación de MINT y su capacidad para llevar a cabo pruebas exhaustivas en aplicaciones móviles.

3.4 Configuración de MINT:

En esta sección, se explicarán en detalle los pasos necesarios para configurar **MINT**.Todas las instrucciones a continuación han sido obtenidas y analizadas desde el repositorio de GitHub de MINT[15], lo que garantiza una guía precisa y actualizada para su configuración y uso.

Para comenzar, es fundamental instalar los artefactos de MINT en su **Repositorio local de Maven:**

Algoritmo 5: Instalación de Artefactos MINT en el Repositorio Local de Maven

1: **Paso 1:** Abra su proyecto local de MINT.
2: **Paso 2:** Ejecute el siguiente comando en su proyecto MINT:
3: `./gradlew publishToMavenLocal`
4: **Paso 3:** Espere a que el proceso de publicación se complete con éxito.
5: **Paso 4:** Los artefactos de MINT ahora se encuentran en su repositorio local de Maven y están listos para su uso en el proyecto.

El siguiente paso implica configurar **la dependencia de MINT** en el código fuente de la aplicación objeto de prueba, siguiendo las instrucciones proporcionadas en su archivo **build.gradle** del módulo de la aplicación [15].

Algoritmo 6: Configurar la dependencia de MINT en tu proyecto

Data: Archivo `build.gradle` del módulo de la aplicación
Result: Agregar la dependencia de MINT en tu proyecto
Agrega las siguientes líneas en el archivo `build.gradle`:

```
repositories {
    ...
    mavenLocal()
}

dependencies {
    def mint_version = 'x.y.z-hash'
    androidTestImplementation "org.mint:android:${mint_version}"
}
```

El siguiente paso implica la configuración del **complemento Gradle de MINT**. Este complemento facilita el trabajo con MINT en la aplicación. A continuación, se presenta cómo se configura el complemento MINT.[15].

Algoritmo 7: Configuración del **Complemento Gradle de MINT en build.gradle**

1: **Paso 1:** Abra el archivo build.gradle en el nivel superior (top-level build.gradle) de su proyecto.
2: **Paso 2:** En la sección de `buildscript`, dentro de `dependencies`, defina la versión del complemento MINT que desea utilizar. Puede reemplazar 'x.y.z-hash' con la versión específica que desee:

```
buildscript {
    dependencies {
        def mint_plugin_version = 'x.y.z-hash'
        classpath "org.mint.tooling:mint-gradle-plugin:${mint_plugin_version}"
    }
}
```

El siguiente paso implica la configuración de la dependencia del complemento MINT en el archivo **build.gradle** del **módulo de la aplicación**.

Algoritmo 8: Configuración del Complemento de MINT en build.gradle del Módulo de la Aplicación

1: **Paso 3:** Abra el archivo build.gradle del módulo de la aplicación de su proyecto.
2: **Paso 4:** Agregue la siguiente configuración al comienzo del archivo build.gradle para prevenir conflictos con otros complementos:

```
apply plugin: 'mint-tooling'

mintTooling {
    // obligatorio, el nombre del paquete de la aplicación
    packageName "org.example.app.debug"
    // opcional, la ubicación donde se almacenarán las salidas de MINT
    targetDir "${project.buildDir}/mint-reports/"
}
```

El siguiente consiste en la configuración dentro del bloque android en el archivo **build.gradle** del **módulo de la aplicación**. Se debe agregar una exclusión para **META-INF/DEPENDENCIES** como se muestra a continuación:

Algoritmo 9: Configuración de Exclusión META-INF/DEPENDENCIES en build.gradle

```
1: android {
2:         packagingOptions {
3:                 exclude 'META-INF/DEPENDENCIES'
4:         }
5: }
```

A continuación, se muestra cómo generar **informes MINT**y realizar operaciones relacionadas para obtener una visión completa de las pruebas y los resultados. Estos informes son esenciales para analizar y evaluar el rendimiento de su aplicación móvil y garantizar una experiencia de usuario óptima.

Algoritmo 10: Creación de Informes MINT y Operaciones Relacionadas

1: Crear un informe MINT basado en pruebas recién ejecutadas: `./gradlew mintReport`
2: Crear un informe MINT basado en datos de pruebas históricos: `./gradlew mintReport -x connectedDebugAndroidTest`
3: Crear un informe MINT basado en datos de pruebas históricos en una ubicación específica: `./gradlew mintReport -x connectedDebugAndroidTest -target build/mint/tmp -no-pull`
4: Recopilar datos históricos de MINT desde el dispositivo (en la ubicación especificada por 'targetDir', si está configurada): `./gradlew collectReportingData`
5: Limpiar cualquier dato de MINT en el dispositivo de prueba: `./gradlew mintClean`

A continuación, se presenta un ejemplo de cómo crear una prueba utilizando **MINT**. Este ejemplo está escrito en el lenguaje de programación **Kotlin** y utiliza las bibliotecas de prueba de Android. El objetivo es proporcionar una visión general de cómo se estructura un test MINT y cómo se puede comenzar a explorar una aplicación móvil utilizando esta herramienta de prueba. A partir de este ejemplo, se podrá adaptar y expandir las pruebas según las necesidades específicas de cada proyecto y aplicación.

Algoritmo 11: Creación de su primer test utilizando MINT

```
package org.mint.exampleapp

import androidx.test.ext.junit.rules.ActivityScenarioRule
import androidx.test.ext.junit.runners.AndroidJUnit4
import org.junit.Rule
import org.junit.Test
import org.junit.runner.RunWith
import org.mint.MINTRule

@RunWith(AndroidJUnit4::class)
class ExampleMintTest
@Rule
@JvmField
var activityScenarioRule = ActivityScenarioRule(MainActivity::class.java)

@Rule
@JvmField
var mint = MINTRule()

@Test
fun mintExampleTestRun()
mint.explore()
```

Cuando se busca realizar **pasos manuales** antes de iniciar la exploración de una aplicación móvil, es decir, cuando se necesita un enfoque híbrido que combine las capacidades de Espresso y MINT, se puede seguir el siguiente ejemplo :

Algoritmo 12: Ejecución de pasos en MINT con configuración híbrida

```
@Test
function MINTEXAMPLETESTRUN
   mint.step {
     onView(withId(R.id.notes_button)).perform(click())
     onView(withId(R.id.add_note_button)).check(matches(isNotEnabled()))
   }.step {
     onView(withId(R.id.add_note_title_edit_text)).perform(typeText("title1"))
     onView(withId(R.id.add_note_button)).check(matches(isEnabled()))
   }.step {
     onView(withId(R.id.add_note_description_edit_text)).perform(typeText("desc"))
     onView(withId(R.id.add_note_button)).perform(click())
   }.explore()
end function
```

CAPÍTULO 4
Diseño del estudio comparativo

4.1 Preguntas de investigación

El objetivo de esta investigación empírica es llevar a cabo una comparación exhaustiva de las diferentes herramientas de pruebas de interfaz gráfica de usuario (GUI) en el entorno de Android seleccionado en el capitulo 2. Para lograrlo, se plantearán **preguntas de investigación** específicas que guiarán nuestra investigación y ayudarán a alcanzar nuestros objetivos.

Además, se formularán **hipótesis** que serán probadas durante el curso de los experimentos.

Preguntas de investigación:

Q1: ¿Cuál es **la cobertura de código** que cada herramienta puede lograr durante las pruebas automatizadas en aplicaciones Android

Q2: ¿Qué tan efectivas son estas herramientas en **la detección de fallos** en la interfaz gráfica de usuario de aplicaciones móviles Android?

Q3: ¿Cuál es la eficiencia en términos de **Facilidad de uso** de la herramienta MINT en comparación con otras herramientas de pruebas de GUI en Android?

Q4: ¿Cuál es **el tiempo de ejecución** que cada herramienta puede lograr durante las pruebas automatizadas de aplicaciones Android?

Estas preguntas se formulan con el objetivo de evaluar el desempeño y la efectividad de las herramientas de pruebas de GUI en el contexto de aplicaciones móviles Android. Para responder a estas preguntas, se ha diseñado un experimento en un entorno controlado, cuyos detalles se abordarán en las próximas secciones.

4.2 Selección de Apps

Para seleccionar las aplicaciones móviles adecuadas para llevar a cabo los experimentos, se realizó una revisión de la literatura relacionada con las pruebas de GUI en Android,[38]. Además, se evaluaron numerosos conjuntos de aplicaciones utilizados en investigaciones previas [31],[34] para determinar su idoneidad en los propios experimentos de esta tesis. Para garantizar la representatividad y la diversidad en nuestra selección, se consideraron factores como la cantidad de aplicaciones, la variedad de tipos de aplicaciones y la calidad de las mismas. La cantidad de aplicaciones seleccionadas se basó en investigaciones previas y en la cantidad media utilizada en estudios similares. La variedad se logró al elegir aplicaciones de diferentes tipos para evaluar cómo se comportan las herramientas en una amplia gama de escenarios. En términos de calidad, se seleccionaron aplicaciones que contaban con código fuente disponible y una alta estabilidad, lo que se determinó mediante la cantidad de descargas y revisiones positivas en las tiendas de aplicaciones.

Además de esta revisión, se llevaron a cabo pruebas preliminares en varias aplicaciones disponibles en F-Droid [11] y al final se consideró un benchmark específico utilizado en la investigación titulada 'Time-travel Testing of Android Apps'[24]. Este benchmark se destacó por su relevancia, ya que incluye 8 de las 100 aplicaciones más populares en F-Droid y Google Play. Además, todas estas aplicaciones están instrumentadas con JaCoCo, lo que permite acceder tanto al código fuente como al archivo APK para llevar a cabo pruebas exhaustivas y análisis detallados.

Tabla 4.1: Aplicaciones Móviles y Disponibilidad

Nombre de la Aplicación	**Disponible en**
APhotoManager	Uptodown / F-Droid / Play Store
AmazeFileManager	F-Droid /Android File Host / Play Store
Firefox Lite	http://mozilla.org/MPL/2.0/
MaterialFBook	F-Droid
Omni-Notes	Play Store / F-Droid
Geohashdroid	F-Droid
Open-Event-Attendee-Android	Play Store / F-Droid
OpenLauncher	Play Store / F-Droid

La elección de estas aplicaciones se basa en criterios rigurosos que buscan garantizar su representatividad y adecuación para nuestros objetivos de investigación. Estos criterios incluyen la cantidad de aplicaciones, la variedad de tipos de aplicaciones y la calidad de las mismas. En las secciones siguientes, se proporcionará información detallada sobre las aplicaciones seleccionadas y cómo se utilizarán en los experimentos.

4.3 Métricas

En esta sección, presentamos las métricas clave que utilizaremos para evaluar el rendimiento y la efectividad de las pruebas en el estudio experimental. Estas métricas son indicadores esenciales que permitirán cuantificar y medir de manera objetiva los resultados obtenidos en los experimentos. Cada métrica desempeña un papel fundamental en la evaluación comparativa de las herramientas de pruebas y ayuda a responder a las preguntas de investigación planteadas. A continuación, se detalla cada métrica, su significado y cómo se mide en el contexto del experimento.

1. **Cobertura de Código:** La métrica de cobertura de código evalúa cuántas líneas de código y ramas de decisión en el código fuente de la aplicación son ejecutadas durante las pruebas. Esta métrica se mide generalmente utilizando herramientas de seguimiento de ejecución, como JaCoCo o Cobertura, que registran qué partes del código se ejecutaron durante las pruebas. La métrica se expresa como **un porcentaje del código** total cubierto en relación con el código total

de la aplicación. Un mayor porcentaje de cobertura de código generalmente indica una prueba más exhaustiva. En el **experimento** realizado, se midió la cobertura de código utilizando las herramientas 2.1 de prueba después de la ejecución del experimento. Esto permitió realizar una comparación y evaluación objetiva de la efectividad de cada herramienta en términos de cuánto código logró cubrir durante las pruebas.

2. **Detección de Fallos:** Cuantifica la cantidad de errores o fallos identificados durante las pruebas. Estos errores pueden incluir problemas en la interfaz gráfica de usuario **GUI**, bloqueos de la aplicación, excepciones no controladas y otros problemas que afecten negativamente la experiencia del usuario. La métrica se mide registrando y contando la cantidad de fallos detectados durante el experimentos. Se compara la cantidad de fallos encontrados por cada herramienta de prueba para determinar cuál es más efectiva en la detección de errores.

3. **Tiempo de Ejecución:** Mide la duración necesaria para ejecutar un conjunto de pruebas. Se mide registrando el tiempo desde el inicio hasta la finalización de la ejecución de las pruebas. Los resultados se expresan en unidades de tiempo, como segundos o minutos. Un menor tiempo de ejecución generalmente indica que una herramienta de prueba es más eficiente y rápida en la ejecución de pruebas.

4. **Recursos Utilizados:** La evaluación de los recursos utilizados implica medir los recursos de hardware, como la CPU y la memoria, utilizados durante las pruebas. Se realiza mediante herramientas de monitoreo de recursos que registran el consumo de recursos durante la ejecución de las pruebas. Un uso eficiente de los recursos es deseable para evitar el agotamiento de recursos, lo que podría afectar negativamente a otros procesos en el dispositivo de prueba.

5. **Facilidad de Uso:** La métrica de facilidad de uso evalúa la operatividad y accesibilidad de una herramienta de prueba desde la perspectiva del usuario. Se basa en la experiencia del usuario al utilizar la herramienta y su capacidad para navegar y comprender la interfaz de usuario, la documentación proporcionada y otros aspectos relacionados con la configuración y el uso de la herramienta. La facilidad de uso se expresa generalmente a través de una puntuación o una evaluación cualitativa. Esta métrica proporciona información crítica sobre la experiencia del usuario y su comodidad al trabajar con la herramienta de prueba. En el experimento, se evalúa la facilidad de uso de cada herramienta de manera objetiva para comprender cómo influye en la eficiencia y la efectividad de las pruebas.

4.4 Descripción de los experimentos

En esta sección, se proporciona una visión detallada de cómo se llevarán a cabo los experimentos destinados a evaluar el rendimiento y la efectividad de las herramientas de prueba seleccionadas, con un enfoque particular en MINT. El diseño de estos experimentos se ha basado en un entorno altamente controlado y reproducible que garantiza la integridad y la validez de los resultados.

4.4.1. Arquitectura y Configuración del Entorno Experimental

El entorno experimental se compone de un conjunto de **5 máquinas virtuales Windows** y **5 máquinas virtuales Ubuntu (Ubuntu 14:04 / 16:04 / 18:04 / 20:04 / 22:04)**, alojadas en los servidores de la Universidad Politécnica de Valencia. Estas máquinas virtuales están configuradas con Apache Guacamole,[2], lo que permite un acceso directo y conveniente a través del siguiente enlace: **https://qdesktop.testar.org/**. Esto facilita la gestión y el control de las máquinas virtuales desde una ubicación centralizada, que es esencial para la realización de los experimentos. Estas máquinas virtuales se han configurado especialmente para reflejar un entorno de desarrollo realista y escalable. Cada máquina virtual está equipada con procesadores Intel Xeon y una cantidad adecuada de memoria **RAM de 8,00 Go** para garantizar un rendimiento óptimo durante las pruebas y con un tipo de sistema: sistema operativo de 64 bits, **procesador x64**.

Tipo de SO	Procesador	RAM	Tipo de procesador
Windows (5)	Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz	8,00 Go	processeur x64
Ubuntu 14:04	Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz	8,00 Go	processeur x64
Ubuntu 16:04	Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz	8,00 Go	processeur x64
Ubuntu 18:04	Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz	8,00 Go	processeur x64
Ubuntu 20:04	Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz	8,00 Go	processeur x64
Ubuntu 22:04	Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz	8,00 Go	processeur x64

Tabla 4.2: Máquinas Virtuales Workstation (Windows y Ubuntu)

Las máquinas virtuales Windows albergarán las herramientas de prueba, incluyendo MINT y sus dependencias, mientras que las máquinas virtuales Ubuntu proporcionarán una infraestructura de soporte adicional. La disponibilidad de estas máquinas virtuales permite la ejecución de las pruebas en un entorno controlado y altamente escalable, lo que garantiza que los resultados sean confiables y reproducibles.

Además, se han preparado 3 máquinas virtuales del emulador Android, separadas de las máquinas Windows, pero interconectadas en una red privada. Esto permite una conexión eficiente a las máquinas Android desde las máquinas Windows utilizando el protocolo **ADB (Android Debug Bridge)** ver[1]. Cada máquina virtual Android se ha configurado con una version de Android para garantizar la coherencia en las pruebas de MINT.

Versión de Android	SDK/API Level	Codename	IP
9.0	Level 28	Pie	10.102.0.196
8.1	Level 27	Oreo	10.102.0.197
4.4	Level 19	KITKAT	10.102.0.195

Tabla 4.3: Máquinas Virtuales Android

La siguiente **figura 4.1** ilustra la arquitectura del entorno experimental.

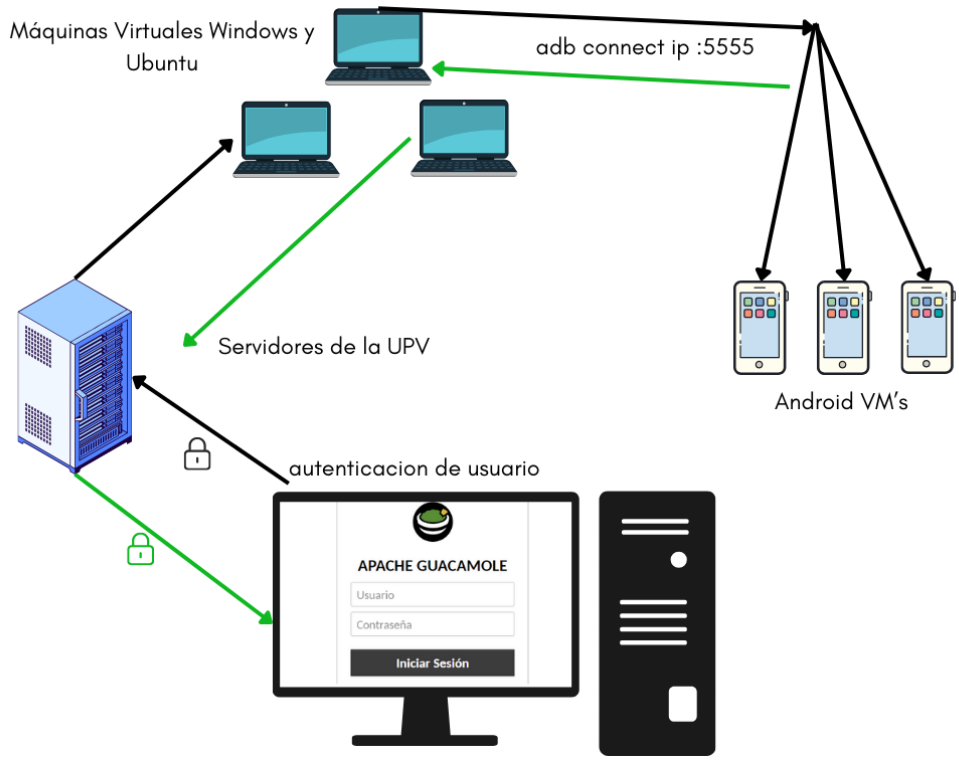

Figura 4.1: Arquitectura del entorno del experimento

4.5 Implementación del Experimento

4.5.1. Instalación de Herramientas de Android GUI Testing

La implementación del experimento comienza con la instalación de las herramientas de Android GUI Testing en las máquinas virtuales correspondientes. Cada herramienta requiere un proceso de instalación y configuración específico. A continuación, se detallan los pasos generales que se seguirán para cada herramienta:

- **Dynodroid:** La implementación de **Dynodroid** [35]implica la configuración de una máquina virtual (VM) de VirtualBox que ejecute este sistema de prueba. Antes de comenzar, es necesario descargar una imagen de máquina virtual autocontenida (VHD) [10] de **Dynodroid** desde su repositorio oficial [9] o desde una fuente confiable[10]. Una vez obtenida la imagen VHD, se procede a crear una VM en VirtualBox configurando el sistema operativo como Ubuntu de 32 bits.

  La VM se configura con recursos adecuados, como memoria RAM y espacio en disco, para garantizar un rendimiento óptimo durante las pruebas. Además, se configura la red de la VM como .ªdaptador puente"para permitir la conectividad a Internet y la transferencia de archivos.

  Luego, se carga la imagen VHD de **Dynodroid** en la VM, estableciendo la configuración de almacenamiento en VirtualBox. Una vez que la VM está lista y se inicia, se accede a ella mediante un nombre de usuario y contraseña predefinidos, que son "dynodroid".[9]

  A continuación, se preparan las pruebas copiando el archivo APK de la aplicación a probar en la VM, y se coloca en la carpeta .ªpps.en el escritorio de la VM. Se pueden realizar ajustes adicionales según la documentación de **Dynodroid**[9], como la edición del archivo "dynodroid.properties"si es necesario.

  Finalmente, **Dynodroid** se ejecuta en la VM a través de una terminal, donde se navega a la ubicación de **Dynodroid** y se ejecuta el comando .ªnt run". Esto permite que **Dynodroid** comience a ejecutar pruebas en la aplicación APK proporcionada, generando resultados que serán útiles para la evaluación y análisis posteriores.

- **AutoDroid**: La implementación de **AutoDroid** se lleva a cabo en una de las máquinas virtuales Windows preparadas previamente. Antes de comenzar, es fundamental instalar las dependencias necesarias. Estas dependencias están especificadas en el archivo de requisitos (`REQUIREMENTS.txt`) y pueden ser instaladas siguiendo el siguiente comando:

  ```
  pip install -r REQUIREMENTS.txt
  ```

  Una vez que se han instalado las dependencias, es necesario proporcionar un archivo de configuración JSON válido a **AutoDroid** como argumento de línea de comandos. A continuación, se muestra un ejemplo sencillo de archivo de configuración que recuperará todas las aplicaciones de todos los dispositivos conectados, una por una, y extraerá los archivos APK a archivos ZIP utilizando AndroGuard:

  ```
  {
    "devices": ["*"],
    "apps": ["*"],
    "commands": ["adb pull !app_path !app_id.apk", "reverse: !app_id.apk"]
  }
  ```

  Una vez que se ha creado un archivo de configuración válido para **AutoDroid**, se puede iniciar la interacción con los dispositivos ejecutando el archivo Python `AutoDroid.py` con el archivo de configuración como parámetro de línea de comandos:

```
python AutoDroid.py ejemplo_config.json
```

Comandos y Bloques: El archivo de configuración de **AutoDroid** puede contener una serie de comandos para ejecutar en los dispositivos de destino. Estos comandos se ejecutan localmente en su máquina, por lo que los programas y archivos llamados deben estar presentes. Los comandos pueden estar en formato de lista o como pares clave-valor (mapa/diccionario). Los pares clave-valor se definen como bloques de comandos, donde la clave es el nombre del bloque y el valor es una lista de comandos. Puede utilizar el bloque constante (descrito a continuación) `block:<nombre del bloque>` para ejecutar un bloque y proporciona una característica de bucle/simple llamada. Aquí hay un ejemplo de cómo utilizar bloques:

```
{
  "devices": ["*"],
  "apps": ["*"],
  "commands": {
    "test_user_input": ["adb -s !device_id shell monkey -v 5 -p !app_id"],
    "retrieve_apk": ["adb -s !device_id pull !app_path !app_id.apk", "sleep:5"]
  }
}
```

Dispositivos y Aplicaciones: Dos campos adicionales fundamentales para **AutoDroid** son los campos `devices` y `apps`. Estos campos definen el ID del dispositivo ADB para los dispositivos de destino (una lista de cadenas) y la notación de dominio inverso (es decir, com.example.aplicacion) para las aplicaciones de destino en el dispositivo (una lista de cadenas). Ambos campos pueden estar vacíos, ser una lista de cadenas o estar definidos como "*"donde se seleccionarán todos los dispositivos y aplicaciones disponibles. Cuando se proporciona un valor en estos campos, el programa recorrerá todos los comandos en orden para cada aplicación en cada dispositivo.

Variables: **AutoDroid** permite la creación de un número infinito de variables en un script. Estas variables se construyen en formato de par clave-valor. Cuando la clave de una variable se encuentra en un comando, se reemplazará por el valor de la variable. Un ejemplo de configuración que utiliza variables se muestra a continuación,

- **DroidBot** : La implementación de DroidBot se lleva a cabo en una de las máquinas virtuales Windows preparadas previamente. Para llevar a cabo la implementación de **DroidBot**, es importante asegurarse de que se cumplan los siguientes requisitos previos:

  - Python (se admiten tanto las versiones 2 como 3).
  - Java.
  - Android SDK.
  - Agregar el directorio `platform_tools` del Android SDK al PATH.
  - (Opcional) OpenCV-Python si desea ejecutar DroidBot en modo CV (Computer Vision).

  A continuación, se detallan los pasos para la instalación:

```
git clone https://github.com/honeynet/droidbot.git
cd droidbot/
pip install -e .
```

  Si la instalación se realiza correctamente, debería poder ejecutar `droidbot -h`.

  Para utilizar DroidBot, asegúrese de tener la siguiente información:

  - La ruta del archivo .apk de la aplicación que desea analizar.

- Un dispositivo o un emulador conectado a su máquina host a través de adb.

Para iniciar DroidBot, ejecute el siguiente comando:

```
droidbot -a <ruta_al_apk> -o directorio_de_salida
```

Si está utilizando varios dispositivos, es posible que deba utilizar la opción :

`-d <número_de_serie_del_dispositivo>` para especificar el dispositivo de destino. La forma más sencilla de determinar el número de serie de un dispositivo es ejecutando `adb devices`.

En algunos dispositivos, es posible que deba activar manualmente el servicio de accesibilidad para DroidBot (necesario para obtener la jerarquía de vistas actual).

Si desea probar una gran cantidad de aplicaciones, puede agregar la opción `-keep_env` para evitar la reinstalación del entorno de prueba en cada ejecución.

También se puede utilizar un script en formato JSON para personalizar la entrada para estados específicos. A continuación, se muestran algunos ejemplos de scripts. Simplemente use `-script <ruta_al_script.json>` para usar DroidBot con un script.

Si las aplicaciones no admiten la obtención de vistas a través de Accesibilidad (por ejemplo, la mayoría de los juegos basados en Cocos2d, Unity3d), es posible que encuentre útil la opción `-cv`.

Puede utilizar la opción `-humanoid` para permitir que DroidBot se comunique con Humanoid para generar entradas de prueba similares a las de un humano.

Puede encontrar otras características útiles ejecutando `droidbot -h`.

- **Humanoid:** Para llevar a cabo la implementación de Humanoid en una de las máquinas virtuales Windows preparadas en el entorno experimental, se deben seguir ciertos pasos clave. En primer lugar, es esencial asegurarse de que la máquina virtual esté en funcionamiento y que todas las configuraciones necesarias estén correctamente establecidas.

  Una vez que la máquina virtual está en línea, se procede a la instalación de los prerrequisitos, que incluyen Python 3.10, Tensorflow, DroidBot y PyFlann. Además, si se planea entrenar el modelo Humanoid desde cero, se requerirá el conjunto de datos RICO. Estos elementos son fundamentales para garantizar que Humanoid funcione correctamente en el entorno Windows.

  El siguiente paso implica la ejecución del servicio de Humanoid en la máquina virtual. Esto se logra utilizando el comando adecuado, que se especifica en la documentación de Humanoid [13]. Una vez que el servicio de Humanoid está en funcionamiento y escuchando en un puerto específico, se pueden iniciar instancias de DroidBot en la misma máquina virtual, configurándolas para que utilicen el servicio de Humanoid a través de la opción `-humanoid localhost:50405`.

  Es importante destacar que Humanoid es especialmente eficaz cuando se utiliza con políticas basadas en modelos, como `dfs_greedy`. La comunicación entre DroidBot y Humanoid permite generar entradas de prueba que imitan el comportamiento humano en la interacción con aplicaciones Android.

  Al implementar Humanoid de esta manera en una de las máquinas virtuales Windows, estamos habilitando la generación de pruebas basadas en modelos que pueden mejorar significativamente nuestro experimento de prueba de interfaz gráfica de usuario en aplicaciones Android. Los informes generados por Humanoid proporcionarán datos valiosos para el análisis posterior y la evaluación de la calidad de las aplicaciones bajo prueba.

  Esta implementación detallada garantiza que nuestro experimento se lleve a cabo de manera efectiva y contribuye a la rigurosidad y precisión de nuestros resultados.

- **RegDroid** La herramienta RegDroid [45]se implementó con éxito en una de las máquinas virtuales Windows configuradas previamente en el entorno experimental. Esta implementación se realizó siguiendo una serie de pasos específicos para garantizar la correcta ejecución de las pruebas en aplicaciones Android. A continuación, se detalla el proceso de implementación:

  Primero, se descargó el repositorio de RegDroid desde el repositorio oficial utilizando [17] el siguiente comando en la línea de comandos de la máquina virtual Windows:

  ```
  git clone https://github.com/Android-Functional-bugs-study/home.git
  ```

  Luego, se verificó que la máquina virtual cumpliera con los requisitos necesarios, que incluyen la presencia del Android SDK con API nivel 26 o superior y Python 3.8 instalado. Las bibliotecas requeridas, como uiautomator2, androguard y cv2, se agregaron según las indicaciones proporcionadas en el repositorio.

  Para facilitar las pruebas, se configuró un emulador Android compatible con RegDroid siguiendo las instrucciones proporcionadas en la documentación de Android. Este emulador se creó con la API nivel 26 y otras especificaciones recomendadas.

  Una vez configurado el emulador, se iniciaron dos instancias idénticas del mismo y se les asignaron números de puerto distintos para su identificación. Esto permitió la ejecución paralela de las pruebas en ambas instancias.

  ```
  emulator -avd Android8.0 -read-only -port 5554
  emulator -avd Android8.0 -read-only -port 5556
  ```

  La herramienta RegDroid se puso en funcionamiento ejecutando el comando adecuado :

  Ahora, está listo para ejecutar RegDroid. Utilice el siguiente comando en el directorio del proyecto descargado para ejecutar la aplicación de muestra:

  ```
  python3 start.py -app_path ./App/AmazeFileManager-3.7.1.apk \
  -emulator_path path_to_emulator \
  -app_path ./App/AmazeFileManager-3.7.2.apk \
  -append_device emulator-5554 \
  -append_device emulator-5556 \
  -output 3.7.1-3.7.2 \
  -testcase_count 1 \
  -event_num 20
  ```

  en el directorio del proyecto descargado. Se especificaron los archivos APK de las aplicaciones a probar, la ubicación del emulador, el número de dispositivos de emulación a utilizar y otros parámetros relevantes.

  Los resultados de las pruebas, incluidos los informes de secuencias que desencadenan defectos y capturas de pantalla, se almacenaron en el directorio de salida designado.

  Esta implementación de RegDroid en la máquina virtual Windows permitió llevar a cabo pruebas exhaustivas en las aplicaciones Android seleccionadas, con el objetivo de identificar y documentar posibles defectos. Los informes generados serán fundamentales para el análisis y la evaluación de la calidad de las aplicaciones bajo estudio en el contexto de este experimento.

Cada herramienta se configurará siguiendo las mejores prácticas, que incluyen las configuraciones recomendadas por los desarrolladores de cada herramienta y las configuraciones basadas en investigaciones previas. Además, se integrarán con los emuladores Android para que estén listas para realizar pruebas.

4.5.2. Ejecución de Pruebas

Una vez que las herramientas de Android GUI Testing se han instalado correctamente y los casos de prueba han sido diseñados, el siguiente paso crucial es la ejecución de las pruebas. Cada herramienta se utiliza para llevar a cabo pruebas en emuladores Android que representan diversas versiones del sistema operativo. Sin embargo, es importante destacar que la ejecución de estas pruebas puede requerir ciertos ajustes y configuraciones específicas, dependiendo de la herramienta en uso.

En el caso de **RegDroid**, siguiendo las investigaciones previas y las mejores prácticas establecidas, se ha elegido una configuración de **100 eventos** de prueba por cada aplicación. Esta configuración se refleja en el comando de ejecución, que incluye el parámetro **-event_ num 100**.

Esta elección está respaldada por estudios previos y proporciona una base sólida para la comparación con otras herramientas.

DynoDroid, por otro lado, también ofrece flexibilidad en la configuración de pruebas. En su archivo de configuración llamado **"dynodroid.properties"**, se ha establecido la configuración para ejecutar pruebas con 100 eventos. Esta configuración preestablecida garantiza que las herramientas (Dynodroid,RegDroid) se sometan a una prueba equitativa y comparativa.

Estas configuraciones predefinidas permiten una ejecución uniforme y coherente de las pruebas, lo que facilita la comparación entre las herramientas y garantiza la validez de los resultados obtenidos en este estudio de investigación.

4.5.3. Recopilación de Datos

Durante la ejecución de las pruebas, se llevará a cabo una rigurosa recopilación de datos con el propósito de evaluar el rendimiento y la eficacia de cada herramienta. Esta fase de recopilación de datos incluirá la captura de métricas fundamentales, tales como la cobertura de código, la tasa de errores y el tiempo de ejecución. Todos estos datos serán registrados de manera detallada y sistemática, garantizando su integridad y precisión para análisis posteriores.

Es importante destacar que la recopilación de resultados se realizará de dos maneras distintas. Por un lado, herramientas como **MINT**, **Droidbot** y **Regdroid** generarán informes en formatos que incluyen HTML, XML o archivos de texto (.txt), los cuales contendrán información detallada sobre los resultados obtenidos. Estos informes proporcionarán una visión clara y estructurada de las métricas y resultados obtenidos durante las pruebas.

Por otro lado, algunas herramientas, como **Humanoid**, **AutoDroid** y **Dynodroid**, requerirán un proceso de revisión más exhaustivo de los registros **(logs) del terminal** generados durante la ejecución de las pruebas.

La recopilación de datos se llevará a cabo con meticulosidad y precisión, garantizando que los resultados obtenidos sean confiables y representativos de la efectividad de cada herramienta en el contexto de pruebas de GUI en Android.

CAPÍTULO 5
Evaluación

5.1 Resultados por cada pregunta de investigación

5.1.1. Cobertura de Código

La Tabla: 5.1 proporciona una visión detallada de la cobertura de código alcanzada por cada herramienta de prueba de GUI en aplicaciones móviles Android para las aplicaciones seleccionadas. Esta métrica es esencial para responder a **la pregunta de investigación Q1**, que se centra en la cobertura de código lograda durante las pruebas automatizadas. Al comparar las métricas de cobertura de código para cada aplicación y herramienta, podemos evaluar la eficacia de las herramientas en la exploración exhaustiva de las rutas de ejecución dentro de las aplicaciones Android.

Tabla 5.1: Cobertura de Código para Aplicaciones y Herramientas

Aplicación/Herramienta	MINT	Dynodroid	Droidbot
APhotoManager	-	25 %	25 %
AmazeFileManager	22 %	15 %	19 %
Firefox Lite	-	35 %	30 %
MaterialFBook	39 %	20 %	26 %
Omni-Notes	27 %	15 %	25 %
Geohashdroid	24 %	15 %	33 %
Open-Event-Attendee-Android	31 %	30 %	4 %
OpenLauncher	-	10 %	-

Después de realizar las pruebas y obtener los resultados de la cobertura de código, se identificaron limitaciones o dificultades específicas al utilizar herramientas como **AutoDroid**, **Humanoid** y **RegDroid**. A continuación, se detallan las limitaciones de cada herramienta para proporcionar un vista clara de las consideraciones en el análisis de cobertura de código:

AutoDroid, una herramienta útil para automatizar pruebas en dispositivos Android, su limitación es que no proporciona informaciónes sobre **la cobertura de código** . Esta limitación se debe a que **AutoDroid** se enfoca en simplificar tareas específicas en dispositivos Android, como descargar aplicaciones y realizar pruebas, pero no está diseñado para analizar en detalle cómo se ejecuta el código fuente.

En el caso de **Humanoid**, se ha identificado una limitación significativa en su repositorio de **GitHub**[13], ya que no proporciona información detallada, como scripts, comandos de terminal, u otras instrucciones claras, sobre cómo obtener los resultados de cobertura de código

En el caso de **RegDroid**,hay un problema. Aunque es bueno encontrando errores en aplicaciones sin el código fuente (solo con el archivo APK), tiene dificultades cuando se trata de aplicaciones que incluyen su código fuente (aplicaciones instrumentadas). La complicación viene porque **RegDroid**

no puede instalar las aplicaciones en el emulador cuando se usa el código fuente, lo que significa que no puede hacer las pruebas ni proporcionar la cobertura. En resumen, **RegDroid** funciona bien para descubrir problemas en aplicaciones simples, pero no es adecuado para evaluar la cobertura cuando se incluye el código fuente.

5.1.2. Observación de los Resultados (Cobertura de Codigo)

En el análisis de la **cobertura de código**, se observaron diferencias significativas entre las herramientas **MINT** ,**Dynodroid** y **Droidbot**. En el caso de **MINT**, se logró calcular la cobertura de código para la mayoría de las aplicaciones y herramientas evaluadas, lo que proporcionó una visión valiosa del alcance de las pruebas automatizadas. Sin embargo, es importante señalar que **MINT** enfrentó limitaciones con respecto a tres aplicaciones específicas: APhotoManager, Firefox Lite y OpenLauncher. Estas aplicaciones presentaron configuraciones y código obsoletos que no eran compatibles con MINT en su estado actual. Para obtener resultados precisos en estas aplicaciones, sería necesario actualizar y adaptar su código a las últimas versiones y estándares.El mismo caso con **Droidbot** que enfrentó limitaciones tecnicas con respecto al aplicacion OpenLauncher.

Por otro lado, **Dynodroid** logró calcular la cobertura de código para todas las aplicaciones evaluadas. Sin embargo, al comparar las coberturas obtenidas por las tres herramientas, se observó que la cobertura de código alcanzada por **Dynodroid** y **Droidbot** fue relativamente más baja en comparación con la obtenida por **MINT**. Este resultado sugiere que MINT logra una mayor exhaustividad en la evaluación del código de las aplicaciones, identificando áreas que Dynodroid y Droidbot podrían no haber alcanzado.

Estas diferencias en la cobertura de código resaltan la importancia de la elección de la herramienta de prueba en función de los objetivos específicos de la evaluación. Mientras que **MINT** brinda una visión más completa de la cobertura de código, es fundamental considerar su compatibilidad con las aplicaciones objetivo. Por otro lado, **Dynodroid** y **Droidbot** ofrecen una alternativa que, aunque pueden tener una cobertura de código ligeramente inferior, son más versátil en términos de aplicaciones compatibles."

Es esencial destacar que **la presentación de los resultados** también difiere entre **MINT** , **Dynodroid** y **Droidbot** .

MINT proporciona sus resultados en un formato de informe **HTML**. Este informe detallado ofrece una visión completa de la cobertura de código, estadísticas y otros detalles relevantes.

En el caso de **Droidbot**, para obtener la cobertura se implementa la **instrumentación** de los archivos APK. Este proceso implica la inserción de código adicional en el **código fuente de las aplicaciones**, permitiendo la generación automática de archivos de cobertura durante la ejecución. Al finalizar la ejecución, estos archivos se extraen y almacenan, y mediante la herramienta Jacoco, se genera un informe completo que revela la cobertura alcanzada durante las pruebas.

Por otro lado, **Dynodroid** presenta sus resultados en los registros del **terminal de Ubuntu**, que se generan durante la ejecución de las pruebas. Estos registros contienen información detallada sobre la cobertura de código y otros aspectos importantes.

La elección entre estas formas de presentación de resultados puede depender de las preferencias del usuario y de la facilidad de acceso a la información. Mientras que los informes de **MINT** y **Droidbot** proporcionan una estructura organizada y detallada, los registros de terminal de **Dynodroid** requieren una revisión más minuciosa. Ambas presentaciones tienen sus ventajas y pueden ser útiles en diferentes contextos y para diferentes usuarios, lo que resalta la importancia de considerar las necesidades específicas al seleccionar una herramienta de prueba.

5.1.3. Detección de Fallos

La Tabla 5.2 presenta los resultados de la detección de fallos en la interfaz gráfica de usuario de las aplicaciones móviles Android mediante el uso de diversas herramientas de prueba. Esta métrica es fundamental para abordar **la pregunta de investigación Q2**, que se enfoca en la efectividad de las herramientas en la detección de problemas y errores en la interfaz de usuario. Al analizar los resultados de detección de fallos para cada aplicación y herramienta, podemos evaluar qué herramientas son más capaces de identificar problemas en la interfaz de usuario.

Tabla 5.2: Detección de Fallos para Aplicaciones y Herramientas

Aplicación/Herramienta	MINT	Dynodroid	DroidBot	Humanoid	RegDroid
APhotoManager	-	2	1	1	2
AmazeFileManager	3	0	1	1	1
Firefox Lite	-	0	2	2	6
MaterialFBook	4	1	1	1	0
Omni-Notes	3	6	3	3	15
Geohashdroid	3	0	0	0	0
Open-Event-Attendee-Android	2	0	0	0	3
OpenLauncher	-	2	0	0	5

AutoDroid presenta limitaciones en la detección de errores debido a su enfoque principal en la automatización de interacciones a nivel de dispositivo en entornos Android, más que en la identificación exhaustiva de posibles bugs. Aunque es eficiente en tareas como la descarga masiva de APKs y la prueba de aplicaciones en varios dispositivos, no está diseñado específicamente como una herramienta de análisis de vulnerabilidades o **detección de bugs**.Su funcionalidad se centra en interacciones a nivel de usuario y en la automatización de procesos específicos, lo que puede no abordar integralmente la exploración profunda necesaria para la identificación detallada de bugs en el código fuente de las aplicaciones. Por lo tanto, al utilizar AutoDroid, es importante reconocer que la herramienta puede no proporcionar resultados exhaustivos en términos de detección de bugs debido a su enfoque generalizado en la interacción con dispositivos Android.

5.1.4. Observación de los Resultados (Detección de Fallos)

Una observación destacada en el análisis de los resultados es la similitud en la detección de errores entre las herramientas **DroidBot** y **Humanoid**. Aunque estas herramientas emplean enfoques técnicos ligeramente diferentes, ambas comparten la capacidad de identificar errores de manera efectiva en las aplicaciones analizadas. **DroidBot**, se basa en técnicas de modelado y exploración dinámica para generar acciones de prueba, detectando interacciones de la interfaz de usuario como clics, ingreso de texto, desplazamientos y otros gestos[7].

Por otro lado, **Humanoid** utiliza un enfoque similar, pero incorpora el aprendizaje profundo para generar entradas que simulan el comportamiento humano.[13]

La coincidencia en los resultados de detección de errores entre **DroidBot** y **Humanoid** en este experimento es una observación valiosa. Sin embargo, es importante señalar que esta similitud no implica que ambas herramientas sean idénticas, ya que pueden mostrar diferencias en otros contextos de prueba. La efectividad de estas herramientas puede variar según las características específicas de las aplicaciones y los escenarios de prueba.

RegDroid ha demostrado una notable capacidad para detectar una cantidad significativamente mayor de fallos en aplicaciones específicas, como APhotoManager, Firefox Lite, Omni Notes, Open Launcher y Open Event Attendee Android, en comparación con el resto de las herramientas evaluadas.

Esta distinción en la detección de fallos puede atribuirse a la estrategia única que emplea **RegDroid** en sus pruebas automatizadas. En primer lugar, **RegDroid** utiliza dos emuladores simultáneamente para someter una aplicación a pruebas exhaustivas, [17] lo que aumenta la probabilidad de encontrar fallos que podrían pasar desapercibidos en un entorno de prueba convencional. En segundo lugar, **RegDroid** no se limita solo a las interacciones típicas con la interfaz de usuario de la aplicación, sino que también integra la detección de **"bugs symptom"**. Esto significa que **RegDroid** tiene la capacidad de identificar posibles problemas en la interfaz gráfica de usuario que podrían considerarse como posibles fallos actuales o futuros. Esta funcionalidad se basa en una base de datos de fallos previamente registrados[16], lo que permite a **RegDroid** anticipar y detectar problemas potenciales de manera proactiva.

La capacidad de **RegDroid** para sobresalir en la detección de fallos, respaldada por su enfoque dual de emuladores y su capacidad para identificar problemas en la GUI mediante "bugs symptom", lo convierte en una herramienta valiosa en el arsenal de pruebas de aplicaciones Android. Sin embargo, es importante destacar que la efectividad de **RegDroid** puede variar según la naturaleza de la aplicación y el escenario de prueba específico.

Las aplicaciones *APhotoManager*, *Firefox Lite* y *OpenLauncher*, debido a su antigüedad, no son compatibles con el plugin Mint. Por lo tanto, no pudieron ser incluídas en la comparación con la herramienta **Mint**.

Mint fue capaz de encontrar tres distintos tipos de errores en todas las aplicaciones: dos de ellos relacionados con accesibilidad y un tercer tipo de error basado en los logs de la aplicación. El primer error de accesibilidad consiste en widgets táctiles que no cumplen con un tamaño mínimo, 48x48 dp de forma predeterminada. EL segundo error está relacionado con widgets que no tienen un texto legible. Sin embargo, se detectó falsos positivos en, por ejemplo, casos de widgtes con *tres puntos* que típicamente representan configuración o funcionalidad adicional, a pesar de no tener una representación textual.

En el caso de la aplicación MaterialFBook, **Mint** obtuvo, además, una secuencia de acciones que derivó en el cierre de la aplicación. Este error requiere mayor análisis para entender su naturaleza.

5.1.5. Facilidad de Uso

Para evaluar la facilidad de uso de las herramientas seleccionadas en el contexto de pruebas de GUI en Android, se ha realizado una evaluación cualitativa basada en múltiples criterios. Esta medida es esencial para abordar la pregunta de investigación **Q3**, que se enfoca en comprender cuán fácil o difícil es el proceso de implementación y utilización de estas herramientas en el entorno de pruebas.

La evaluación de la facilidad de uso se basa en los siguientes **criterios:**

1. **Estado de la Herramienta**: Se considera el estado de actualización y mantenimiento de la herramienta. Si una herramienta está actualizada y se mantienen las versiones más recientes, se tiende a considerarla más fácil de usar.
2. **Documentación Completa**: Se analiza la calidad y completitud de la documentación disponible en el repositorio de la herramienta en GitHub. Una documentación detallada y clara puede hacer que la herramienta sea más accesible.
3. **Facilidad de Instalación**: Se evalúa la facilidad de instalación, incluyendo la cantidad de dependencias requeridas y la complejidad de la configuración inicial.
4. **Eficiencia de Implementación**: Se considera cómo se implementa la herramienta en las pruebas. Si requiere una gran cantidad de configuraciones y scripts adicionales, se podría calificar como más difícil de usar en comparación con una herramienta que automatiza gran parte del proceso.
5. **Simplicidad en la Ejecución de Pruebas**: Se analiza si la herramienta simplifica la ejecución de pruebas, permitiendo a los usuarios ejecutar pruebas sin tener que escribir una cantidad significativa de pasos, configuraciones o scripts.
6. **Presentación de Resultados**: Se observa si la herramienta presenta los resultados de las pruebas de una manera clara y accesible, como en una página HTML o un informe estructurado, en lugar de requerir la lectura manual de registros (logs).

Los criterios mencionados anteriormente se utilizan para clasificar cada herramienta en uno de los tres niveles de facilidad de uso: "Fácil", "Moderadoc "Difícil". Esta clasificación proporciona una visión general de la accesibilidad y eficiencia de cada herramienta en el proceso de pruebas, lo que permite a los investigadores y profesionales seleccionar la herramienta más adecuada según sus necesidades y capacidades técnicas.

Tabla 5.3: Facilidad de Uso de Herramientas para Pruebas de GUI en Android

Nombre de la Herramienta	Nivel de Facilidad de Uso
MINT	Moderado
Dynodroid	Moderado
AutoDroid	Fácil
DroidBot	Moderado
Humanoid	Moderado
Regdroid	Moderado

5.1.6. Tiempo de Ejecución

Las Tablas 5.4 ofrecen una valiosa comparación del tiempo de ejecución por cada herramienta de prueba de GUI en aplicaciones móviles Android. Estos datos son fundamentales para abordar la pregunta de investigación Q4, que se centra en la eficiencia en términos de tiempo de las herramientas, con un enfoque particular en la comparación con la herramienta **Mint**. Al analizar detenidamente el tiempo de ejecución consumidos por cada herramienta durante las pruebas en diversas aplicaciones, podemos determinar cuáles de ellas demuestran un rendimiento más eficiente y resultan más adecuadas para la realización de pruebas automatizadas en el contexto de aplicaciones Android. La información recopilada en estas tablas proporciona una visión clara de cómo se comparan las herramientas en estos aspectos clave, lo que contribuye significativamente a la respuesta de la pregunta de investigación Q4.

5.1.7. Tiempo de Ejecución de Pruebas y eventos

En el caso de **RegDroid**, siguiendo las investigaciones previas y las mejores prácticas establecidas, se ha elegido una configuración de **100 eventos** de prueba por cada aplicación. Esta configuración se refleja en el comando de ejecución, que incluye el parámetro **-event_num 100**.

Dynodroid, por otro lado, también ofrece flexibilidad en la configuración de pruebas. En su archivo de configuración llamado **"dynodroid.properties"**, se ha establecido la configuración para ejecutar pruebas con 100 eventos. Esta configuración preestablecida garantiza que las herramientas (DynoDroid y RegDroid) se sometan a una prueba equitativa y comparativa.

Es importante destacar que estas configuraciones predefinidas permiten una ejecución uniforme y coherente de las pruebas, lo que facilita la comparación entre las herramientas y garantiza la validez de los resultados obtenidos en este estudio de investigación. En contraste, con respecto **al resto de las herramientas evaluadas**, se han mantenido los parámetros por defecto según lo indicado en sus respectivas documentaciones y papers de investigación.

Tabla 5.4: Tiempo de Ejecución para Aplicaciones y Herramientas

Aplicación/Herramienta	MINT	Dynodroid	AutoDroid	DroidBot	Humanoid	RegDroid
APhotoManager	-	15 min	2 horas	2 horas	2 horas	100 eventos, 32 min
AmazeFileManager	200 eventos, 45 min	23 min	2 horas	2 horas	2 horas	100 eventos, 32 min
Firefox Lite	-	16 min	2 horas	2 horas	2 horas	100 eventos, 15 min
MaterialFBook	200 eventos, 21 min	10 min	2 horas	2 horas	2 horas	100 eventos, 32 min
Omni-Notes	200 eventos, 15 min	14 min	2 horas	2 horas	2 horas	100 eventos, 14 min
Geohashdroid	200 eventos, 36 min	12 min	2 horas	2 horas	2 horas	100 eventos, 14 min
Open-Event-Attendee-Android	200 eventos, 18 min	12 min	2 horas	2 horas	2 horas	100 eventos, 30 min
OpenLauncher	-	14 min	2 horas	2 horas	2 horas	100 eventos, 15 min

CAPÍTULO 6
Conclusiones y trabajo futuro

Este estudio ha presentado una evaluación empírica exhaustiva de herramientas de GUI testing para aplicaciones Android. La selección de estas herramientas se basó en una revisión rigurosa de la literatura académica y su disponibilidad para su instalación en un entorno de pruebas controlado. Durante la ejecución de experimentos, se evaluaron diversos aspectos clave, incluida la cobertura de las pruebas, la capacidad para detectar fallos y la facilidad de instalación y uso, así como el tiempo de ejecución.

Una vez instaladas las herramientas, se han ejecutado experimentos en un entorno controlado para comprobar la cobertura de las pruebas ejecutadas, la posibilidad de encontrar fallos, y la percepción de facilidad de instalación y uso y tiempo de ejecución.

El proceso de instalación no ha resultado tal como se había planificado. Al instalar las herramientas se ha tenido que solucionar problemas de infraestructura y versiones de los componentes de software necesarios para ejecutar las herramientas que no estaban especificados en ellas. Sin embargo, los resultados presentados en esta memoria son valiosos para la investigación, pues detallan información que puede ser utilizada para continuar con la comparación de herramientas de testing automático. Además, los resultados pueden ser utilizados para ayudar a tomar decisiones informadas a empresas que quieran optar por usar estas herramientas de testing para aplicaciones Android, teniendo en cuenta las ventajas y desventajas de cada una de las herramientas investigadas.

Como trabajo futuro, se plantea la mejora de la arquitectura de los experimentos, con el objetivo de evaluar herramientas de GUI testing en una variedad de sistemas, como aplicaciones de escritorio, iOS, videojuegos, entre otros. Además, se espera llevar a cabo pruebas en entornos del mundo real para evaluar la efectividad práctica de estas herramientas. Un aspecto inmediato del trabajo futuro consiste en la ejecución de los experimentos utilizando aplicaciones Android 9, en lugar de Android 8, para comprender cómo se comportan las herramientas seleccionadas en la última versión del sistema operativo, conocida por su mayor eficiencia y capacidad de respuesta.

En resumen, esta investigación proporciona una base sólida para futuros estudios en el campo del testing automatizado de aplicaciones móviles Android y ofrece una guía valiosa para aquellos que buscan seleccionar las herramientas más adecuadas para sus necesidades de pruebas. Los desafíos superados en el camino solo han fortalecido la validez y el impacto de este estudio, y se espera que las lecciones aprendidas aquí continúen impulsando avances en la calidad y la eficiencia de las pruebas de aplicaciones móviles en el futuro.

Bibliografía

[1] Android debug bridge (adb). https://developer.android.com/tools/adb.

[2] Apache guacamole. https://guacamole.apache.org/.

[3] Appium. http://appium.io/. [Online; accessed 25-12-2019].

[4] Catdroid github repository. https://github.com/CATAndroidTesting/CAT#artifact-package-for-the-cat-paper.

[5] Combodroid github repository. https://github.com/the-themis-benchmarks/combodroid.

[6] Crawldroid github repository. https://github.com/sy1121/CrawlDroid.

[7] Droidbot github repository. https://github.com/honeynet/droidbot.

[8] Droidmate2 github repository. https://github.com/uds-se/droidmate.

[9] Dynodroid github repository. https://github.com/dynodroid/dynodroid.

[10] Dynodroid vhd. https://drive.google.com/file/d/1b60pSjRiI-uGBNQZmrVLFmHlszob4oxK/view.

[11] F-droid. https://f-droid.org/en/.

[12] The (google) monkey. https://developer.android.com/studio/test/monkey.html. [Online; accessed 25-12-2019].

[13] Humanoid github repository. https://github.com/yzygitzh/Humanoid.

[14] Manual testing. https://katalon.com/resources-center/blog/manual-testing.

[15] Mint github repository. https://github.com/ing-bank/mint/blob/main/README.md.

[16] Regdroid dataset. https://github.com/Android-Functional-bugs-study/home/tree/main/Dataset.

[17] Regdroid github repository. https://github.com/Android-Functional-bugs-study/home#regdroid.

[18] Sapienz github repository. https://github.com/Rhapsod/sapienz.

[19] Software testing statistics. https://truelist.co/blog/software-testing-statistics/.

[20] stoat github repository. https://github.com/tingsu/Stoat.

[21] D. Adamo, D. Nurmuradov, S. Piparia, and R. Bryce. Combinatorial-based event sequence testing of android applications. *Information and Software Technology*, 99:98–117, 2018.

[22] S. Bauersfeld, S. Wappler, and J. Wegener. A metaheuristic approach to test sequence generation for applications with a gui. In *Proceedings of the Third International Conference on Search Based Software Engineering*, SSBSE'11, pages 173–187, Berlin, Heidelberg, 2011. Springer-Verlag.

[23] N. P. Borges, J. Hotzkow, and A. Zeller. Droidmate-2: A platform for android test generation. pages 916–919, 2018.

[24] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury. Time-travel testing of android apps. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 481–492, 2020.

[25] A. I. Esparcia, F. Almenar, T. E. J. Vos, and U. Rueda. Using genetic programming to evolve action selection rules in traversal-based automated software testing: results obtained with the TESTAR tool. *Memetic Computing*, 10(3):257–265, 2018.

[26] A. I. Esparcia-Alcázar, F. Almenar, U. Rueda, and T. E. J. Vos. Evolving rules for action selection in automated testing via genetic programming - a first approach. In G. Squillero and K. Sim, editors, *Applications of Evolutionary Computation*, pages 82–95, Cham, Switzerland, 2017. Springer International Publishing.

[27] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su. Large-scale analysis of framework-specific exceptions in android apps. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 408–419, 2018.

[28] X. Gao, S. H. Tan, Z. Dong, and A. Roychoudhury. Android testing via synthetic symbolic execution. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 419–429, 2018.

[29] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, and J. Lu. Aimdroid: Activity-insulated multi-level automated testing for android applications. pages 103–114, 09 2017.

[30] 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. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 269–280, 2019.

[31] Honeynet. Droidbot: a lightweight test input generator for android. 2016.

[32] T. Jansen, F. P. Ricós, Y. Luo, K. van der Vlist, R. van Dalen, P. Aho, and T. E. J. Vos. Scriptless gui testing on mobile applications. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pages 1103–1112, 2022.

[33] P. Kacandes. Bugs: The secret killer to 5-star mobile app reviews.

[34] Y. Li, Z. Yang, Y. Guo, and X. Chen. Humanoid: A deep learning-based approach to automated black-box android app testing. 2019.

[35] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. 2013.

[36] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for android applications. 2016.

[37] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li. Reinforcement learning based curiosity-driven testing of android applications. page 153–164, 2020.

[38] S. Piparia, D. Adamo, R. Bryce, H. Do, and B. Bryant. Combinatorial testing of context aware android. 2021. IEEE Catalog Number: CFP2185N-ART.

[39] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella. Deep reinforcement learning for black-box testing of android apps. *ACM Trans. Softw. Eng. Methodol.*, 31(4), jul 2022.

[40] T. Su. Fsmdroid: Guided gui testing of android apps. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 689–691, 2016.

[41] T. E. J. Vos, P. Aho, F. Pastor Ricos, O. Rodriguez-Valdes, and A. Mulders. testar – scriptless testing through graphical user interface. *Software Testing, Verification and Reliability*, 31(3):e1771, 2021. e1771 stvr.1771.

[42] T. E. J. Vos and A. I. Esparcia. Software testing innovation alliance - the SHIP project -. In *Joint Proceedings of the Doctoral Symposium and Projects Showcase Held as Part of STAF 2016 co-located with Software Technologies: Applications and Foundations (STAF 2016), Vienna, Austria, July 4-7, 2016*, pages 65–71, 2016.

[43] T. E. J. Vos, P. Tonella, J. Wegener, M. Harman, W. Prasetya, E. Puoskari, and Y. Nir–Buchbinder. Future internet testing with fittest. In *15th European Conference on Software Maintenance and Reengineering*, pages 355–358, March 2011.

[44] J. Wang, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu. Combodroid: Generating high-quality test inputs for android apps via use case combinations. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 469–480, 2020.

[45] Y. Xiong, M. Xu, T. Su, J. Sun, J. Wang, H. Wen, G. Pu, J. He, and Z. Su. An empirical study of functional bugs in android apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, pages 1319–1331, 2023.