Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Un estudio empírico sobre la eficacia de las herramientas de reparación automática de código para aplicaciones Android

Tesis de Licenciatura en Ciencias de la Computación

Joaquín Arribas
Nicolás Walter

Director: Lic. Iván Arcuschin Moreno

Buenos Aires, 2023

Índice general

1. INTRODUCCIÓN

Los errores de software, comúnmente conocidos como *bugs*, representan un desafío persistente en el campo de la ingeniería de software. Su frecuencia, y la inversión de tiempo y recursos necesarios para su identificación y corrección son considerables. Tradicionalmente esta labor recae en los programadores y se ejecuta de manera manual. Por esta razón, desde hace años se investigan técnicas de localización de fallas y reparación de programas, buscando la automatización de este proceso crítico.

El objetivo de aplicar técnicas de localización de fallas consiste en la identificación de elementos de código sospechosos de ser defectuosos. Esto permite a los desarrolladores ahorrar tiempo significativo durante el proceso de análisis y corrección de fallas. Además, estos elementos pueden servir como entrada para herramientas de reparación automática de programas como las que se evalúan en esta tesis.

La plataforma Android se destaca como una de las plataformas líderes en el ámbito de dispositivos móviles, abarcando desde teléfonos celulares hasta tablets, y cuenta con una adopción masiva a nivel global[1]. Es por esto que resulta de mucho interés poder evaluar las distintas técnicas de localización de fallas y reparación de programas orientadas a este tipo de aplicaciones.

Para llevar a cabo una evaluación efectiva y rigurosa de estas técnicas es fundamental contar con un conjunto de datos auténtico. Por lo tanto, optamos por utilizar *bugs* reales extraídos de aplicaciones de código abierto, con el objetivo de acercar nuestras evaluaciones a los desafíos que se enfrentan en el mundo real.

Es importante señalar que la eficacia de cualquier técnica de localización y reparación de fallas depende de la calidad y cantidad de *tests* de la aplicación en cuestión. Este factor introduce variables adicionales que deben considerarse al seleccionar o construir el conjunto de datos de evaluación, añadiendo así una capa adicional de complejidad en la preparación del mismo.

El objetivo de este trabajo es evaluar cómo el paso del tiempo influye en la efectividad de las técnicas de localización de fallas y reparación de programas. Se desarrolla en dos fases:

- Primero elaboramos un benchmark compuesto por 21 *bugs* y describimos el proceso seguido para su construcción.
- Luego, utilizando el benchmark construido y la herramienta Astor4Android[2], evaluamos tanto las técnicas de localización y reparación de errores que implementa, como la herramienta en sí.

Finalmente, logramos ejecutar Astor4Android correctamente para 8 bugs, logrando reparar 2 de ellos automáticamente.

2. PRELIMINARES

2.1. Android

Android es un sistema operativo móvil basado en el kernel de Linux y desarrollado por Google. Es el más popular del mundo y se utiliza en una variedad de dispositivos, desde teléfonos inteligentes y tabletas hasta relojes inteligentes, televisores y automóviles[3].

Los lenguajes de programación más comunes para el desarrollo de aplicaciones Android son Java y Kotlin, aunque también es posible utilizar C++ y otros lenguajes a través de la NDK (Native Development Kit).

La arquitectura de Android se compone de las siguientes capas:

- **Kernel de Linux**: Capa más baja, proporciona servicios esenciales como el manejo de dispositivos, gestión de memoria y control de procesos.
- **Librerías Nativas**: Incluyen bibliotecas para tareas como gráficos, bases de datos y conectividad de red.
- **Android Runtime**: Android Runtime (ART) es el entorno de ejecución utilizado por el sistema operativo Android para ejecutar aplicaciones. Reemplazó a Dalvik a partir de Android 5.0 y mejoró significativamente el rendimiento y la eficiencia. La función principal de ART es convertir el bytecode de una aplicación en código nativo, que luego se ejecuta en el sistema operativo subyacente.
- **Framework de Aplicaciones**: Proporciona clases y servicios esenciales para el desarrollo de aplicaciones.
- **Aplicaciones**: Capa más alta, donde se ejecutan las aplicaciones del usuario y del sistema.

Desde su lanzamiento inicial en 2008 con la versión 1.0, el sistema operativo Android experimentó numerosas actualizaciones, llegando a la versión 13 en el presente. Cada iteración introdujo nuevas características y mejoras, lo que llevó a cambios en el paradigma de programación. Esto se traduce en que algunas aplicaciones pueden necesitar una versión específica del sistema operativo para funcionar correctamente, y aplicaciones más antiguas podrían no ser compatibles con versiones más recientes de Android. Aunque el Android Runtime (ART) de las versiones más actuales sigue siendo capaz de interpretar el bytecode Dalvik original, esto no garantiza que todas las aplicaciones sean compatibles, especialmente si dependen de características del entorno de ejecución que han sido descontinuadas o declaradas obsoletas.

2.2. Gradle

Gradle[1] es una herramienta de gestión de proyectos y ensamblado de código que ofrece un marco para la automatización del ciclo de vida del desarrollo de software. Se especializa en facilitar la compilación, prueba y despliegue de aplicaciones y bibliotecas, pro-

[1] https://gradle.org/

porcionando un lenguaje de configuración declarativo y una extensa interfaz para tareas personalizadas. Es la herramienta oficial de compilación de Android[4].

En el contexto de Android, Gradle actúa como un orquestador para diversas tareas, encargadas de:

- Compilar el código fuente en bytecode de Dalvik o ART.
- Empaquetar archivos que no son parte del código fuente, tales como imágenes, archivos de audio, etc.
- Gestionar y resolver dependencias de bibliotecas.
- Crear el archivo APK o AAB para distribuir la aplicación. El formato APK (Android Package Kit) es una extensión del formato JAR y el formato AAB (Android App Bundle) está diseñado para empaquetar y distribuir de manera eficiente aplicaciones a través de la Google Play Store, la tienda oficial de aplicaciones de Google.
- Ejecutar pruebas unitarias y de instrumentación.
- Generar informes y documentación.

Un proyecto Android está compuesto por un conjunto de elementos necesarios para crear una aplicación, incluyendo código, recursos y configuraciones. Un módulo es una parte separable del proyecto que se puede compilar y probar de forma autónoma. Los módulos pueden ser desde la aplicación principal hasta bibliotecas o funcionalidades adicionales, y son reutilizables entre diferentes proyectos.

Los proyectos Android típicamente contiene un archivo build.gradle en el nivel del proyecto y en el nivel del módulo. Estos archivos contienen la configuración necesaria para construir la aplicación, como las versiones del SDK (Software Development Kit) de Android, las dependencias de bibliotecas y las configuraciones específicas del módulo.

Otro concepto relevante es el de las *tareas* de Gradle: Una tarea es una unidad atómica de trabajo dentro de Gradle. Son los bloques de construcción básicos que definen qué se debe hacer durante el proceso de construcción de un proyecto. Pueden realizar una amplia variedad de operaciones, desde compilar clases hasta ejecutar pruebas y desplegar aplicaciones.

2.3. Bibliotecas

En el desarrollo de aplicaciones Android, las bibliotecas pueden distribuirse en dos formatos principales: JAR (Java Archive) y AAR (Android Archive). Ambos formatos se utilizan para empaquetar clases de código, pero difieren en los tipos de archivos que pueden contener. La explicación de cada uno es la siguiente:

- **JAR**: Es un formato de archivo de biblioteca estándar en Java, extensión del formato .zip, y se utiliza para agrupar múltiples archivos .class en un único archivo .jar. No es específico de Android.
- **AAR**: Es específico de Android y se utiliza para contener bibliotecas de código reutilizables para esta plataforma. Es más versátil que el formato jar cuando se trata de empaquetar componentes para Android, ya que permite empaquetar no solo código, sino también imágenes y otros recursos.

2.4. Tests

Los tests de unidad y de instrumentación son dos tipos de pruebas que se utilizan en el desarrollo de aplicaciones Android para intentar garantizar que el código es robusto, fiable y libre de errores. Hay dos categorías:

- **Tests de unidad:** Se centran en probar pequeñas piezas de código de forma aislada, como métodos o clases individuales, para asegurarse de que se comportan como se espera. Sus principales ventajas son su rapidez y facilidad de ejecución, al no depender de componentes externos.
- **Tests de instrumentación:** Son pruebas que involucran interacción con la interfaz de usuario de la aplicación y requieren un dispositivo o emulador para ejecutarse. Estos tests evalúan el comportamiento de una aplicación en condiciones cercanas a las reales.

2.5. Emulador de Android

Es una herramienta de software que permite simular el funcionamiento de un dispositivo Android en una computadora. Forma parte del entorno de desarrollo de Android (Android SDK) y se utiliza ampliamente para fines de desarrollo, pruebas y depuración de aplicaciones Android.

2.6. Estructura de directorios estándar de una aplicación Android

Un proyecto de Android, puede contener uno o más módulos. Un módulo es una colección de archivos de código fuente y parámetros de configuración de compilación que permiten dividir el proyecto en unidades de funcionalidad discretas. El módulo principal, es denominado *app*. La jerarquía de directorios estándar sigue la siguiente estructura[5]:

Directorio raíz del proyecto

- settings.gradle: Archivo que define qué módulos están incluidos en el proyecto.
- build.gradle: Archivo que define la configuración de compilación que se aplica a todos los módulos

Módulos

- nombre-del-módulo/
  - build/: Contiene resultados de compilación.
  - libs/: Contiene bibliotecas privadas.
  - src/: Contiene todos los archivos de código y recursos para el módulo en los siguientes subdirectorios:
    - main/: Contiene el conjunto de archivos de código fuente “principales”: el código y los recursos de Android compartidos por todas las variantes de compilación (los archivos para otras variantes de compilación residen en directorios del mismo nivel, como src/debug/ para el tipo de compilación de depuración).

- java/: Contiene fuentes de código Kotlin o Java, o ambos, si la aplicación tiene código fuente Kotlin y Java.
- kotlin/: Contiene solo fuentes de código Kotlin.
- res/: Contiene recursos de aplicación, como archivos de elementos de diseño y de strings de interfaz de usuario.
- assets/: Contiene archivos que se compilarán en un archivo APK tal como están.
- AndroidManifest.xml: Describe la naturaleza de la aplicación y cada uno de sus componentes.
- build.gradle: Define las configuraciones de compilación específicas para el módulo.

- test/: Contiene código para pruebas unitarias que se ejecutan localmente.
- androidTest/: Contiene código para las pruebas de instrumentación que se ejecutan en un dispositivo Android.
- cpp/: Contiene código C o C++ nativo en el cual se usa la NDK.

2.7. Google Play Store

Es la tienda de aplicaciones oficial de Android y sirve como plataforma principal para distribuir y descargar aplicaciones y contenido digital. Desarrollada y mantenida por Google, esta plataforma proporciona un ecosistema seguro y centralizado que conecta a desarrolladores de aplicaciones con usuarios finales.[2]

2.8. F-Droid

F-Droid[3] es una alternativa a las tiendas de aplicaciones convencionales para Android, especializada en alojar aplicaciones de código abierto y software libre.

[2] https://play.google.com
[3] https://f-droid.org/

3. CONSTRUCCIÓN DEL BENCHMARK

3.1. Trabajo relacionado

Existen conjuntos de datos de bugs previamente publicados, como DroidBugs[2], DroixBench[6] y Defects4J[7]. Tanto DroidBugs como DroixBench son del año 2018, y Defects4J es un benchmark de bugs de Java, no de Android.

- **DroidBugs**[1]: Contiene 13 bugs de 5 aplicaciones distintas. Todas las aplicaciones tenían al menos 5.000 descargas en Google Play Store, y al menos 10 tests.
- **DroixBench**[2]: Contiene 24 *crashes* de 15 aplicaciones distintas. Un *crash* es la finalización inesperada del proceso de una aplicación, efectuada por el sistema operativo, debido a una señal o excepción no controlada[8].

3.2. Motivación

Aunque existen conjuntos de datos de bugs previamente publicados, como DroidBugs[2], DroixBench[6] y Defects4J[7], identificamos la necesidad de crear uno nuevo. La principal razón de esta iniciativa es que estos trabajos anteriores han estado disponibles durante varios años.

Nuestro objetivo es evaluar cómo el paso del tiempo influye en la efectividad de las técnicas de localización de fallas y reparación de programas. Al incorporar bugs más recientes, aspiramos a proporcionar una evaluación más actualizada y relevante para el entorno moderno de aplicaciones Android. En este mismo sentido, optamos por centrarnos exclusivamente en *bugs* originados en aplicaciones reales, que creemos representan adecuadamente los desafíos enfrentados en entornos de producción.

3.3. Introducción

El primer objetivo que nos planteamos fue armar un benchmark de bugs a partir de los repositorios de aplicaciones de código abierto y software libre. Cada elemento de este conjunto de datos es una tupla de commits ($commit_{fix}$, $commit_{bug}$) que cumplen ciertas características:

- Tener tests, ya sea de unidad o de instrumentación.
- Que los tests demuestren que $commit_{fix}$ arregla un bug. Es decir, que en $commit_{bug}$ hayan tests fallidos que se solucionen en $commit_{fix}$.
- $commit_{fix}$ debe ser el commit inmediatamente siguiente a $commit_{bug}$. Esto es así para poder garantizar que los cambios realizados en $commit_{fix}$ son los que efectivamente solucionan el bug.

[1] https://github.com/I4Soft/DroidBugs
[2] https://github.com/stan6/droixbench

- Que en $commit_{fix}$ se modifiquen pocas líneas de código. Esto es porque consideramos que mientras menos líneas tenga el commit es más probable que sea algo simple de ser reparado automáticamente.
- Que el código modificado sea en lenguaje Java. Varias aplicaciones de Android utilizan también Kotlin, pero nos enfocamos en el primer lenguaje porque las herramientas de reparación automática funcionan solo con Java.[2]

Cada $commit_{bug}$ será utilizado como input para las herramientas de reparación de programas.

En este capítulo contaremos cómo fue el proceso de construcción del benchmark. Para esto pasamos por distintas etapas de selección y filtrado de repositorios que contenían posibles tuplas candidatas a ser utilizadas para la etapa de reparación de programas.

Podemos ver en el siguiente gráfico cómo fue este proceso, y cómo se fue achicando el espacio de búsqueda hasta obtener los repositorios con las tuplas candidatas:

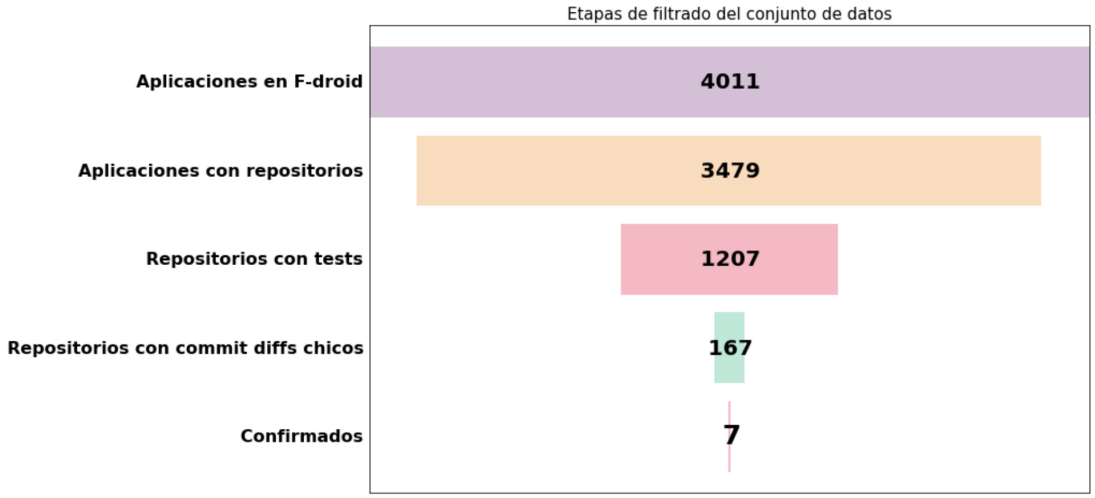

Fig. 3.1: Cantidad de repositorios por cada etapa de filtrado del conjunto de datos

Luego de todo este proceso obtuvimos 9 tuplas ($commit_{fix}$, $commit_{bug}$) de 7 repositorios de manera automática. Además sintetizamos manualmente tests para 12 tuplas, consiguiendo un benchmark de 21 tuplas pertenecientes a 11 repositorios. A continuación, listamos el proceso.

3.4. Búsqueda de aplicaciones

Partimos de F-droid para construir el benchmark. Al momento de realizar este análisis, el catálogo contaba con 4.011 aplicaciones. Cada aplicación pertenece a una categoría determinada por su funcionalidad (juegos, billeteras virtuales, etc).

Luego de analizar el sitio vimos que cada aplicación, de existir, tenía un hipervínculo a su repositorio de control de versiones. Nos propusimos desarrollar un *web scraper*[3] que

[3] https://es.wikipedia.org/wiki/Web_scraping

devolviera como resultado un listado de hipervínculos de repositorios Git[4] La metodología implementada en el scraper es la siguiente:

- Recorrer el listado de categorías de F-droid.
- Para cada categoría, recorrer las páginas. Una página es una lista de items de longitud fija. Un item es información sobre una aplicación (título, descripción, links útiles, etc.). Las categorías estaban paginadas por tener muchos items.
- Para cada item dentro de una página, conseguir su URL de F-Droid.
- Dentro de esa sección se consigue el hipervínculo al repositorio de esa aplicación en particular.

No todas las aplicaciones tenían un hipervínculo a su repositorio. Además, el código de algunas estaba alojado en repositorios muy poco usados[5], por lo que decidimos descartarlas. Nos quedamos solo con las que tenían repositorios en los sitios más populares: GitHub[6] (3.064 aplicaciones), GitLab[7] (376 aplicaciones) y BitBucket[8] (39 aplicaciones).

3.5. Obtención de aplicaciones

Una vez conseguidos los 3.479 repositorios, necesitabamos corroborar cuantos de estos tenían tests. Para eso desarrollamos un programa que, dado un repositorio, devuelve cuantos tests de unidad y de instrumentación tiene. Para realizar la detección y conteo de tests hay que recorrer la estructura del repositorio. Comenzamos por los repositorios alojados en *GitHub* dado que representaban el mayor porcentaje del total.

Nuestro primer acercamiento fue utilizar la API REST de *GitHub*[9] para ir navegando por cada repositorio. Buscamos detectar los repositorios que tuvieran los directorios *androidTest* o *test*, dado que solo nos interesaban las aplicaciones que tuvieran casos de prueba. Uno de los problemas encontrados en esta etapa es que esta API limita la cantidad de peticiones que se pueden realizar en un período de tiempo. Como queríamos minimizarlas decidimos realizar la búsqueda de estos directorios utilizando *breadth-first search*[10]. Este algoritmo recorre el grafo a lo ancho, característica deseada en este caso porque los directorios buscados usualmente se encuentran cerca de la raíz. Luego de probar esta opción decidimos descartarla porque evidenciamos que el tiempo para obtener todos los datos iba a ser muy superior a lo estimado. Esto se debe a que el volumen de peticiones a realizar superaba ampliamente el límite de 5000 por hora impuesto por *GitHub*[9].

Al vernos limitados por este enfoque, decidimos hacer un programa para clonar todos los repositorios y realizar el análisis de forma local. De esta forma la solución sería compatible tanto para *GitHub*, como para *GitLab* y *BitBucket*. De la otra manera, tendríamos que haber modificado el código para adaptarnos a las distintas APIs de las otras plataformas.

[4] https://git-scm.com
[5] Algunos ejemplos de esto son: https://code.google.com/archive/, https://codeberg.org/, o mismo repositorios SVN.
[6] https://github.com/
[7] https://about.gitlab.com/
[8] https://bitbucket.org/
[9] https://docs.github.com/en/rest/repos/contents?apiVersion=2022-11-28
[10] https://es.wikipedia.org/wiki/Busqueda_en_anchura

Es importante notar las ventajas y desventajas de haber clonado todos los repositorios. Si bien nos permitió no depender de la red ni de condiciones de las distintas compañías, tener los 3.479 repositorios descargados representa más de 250 gigabytes de disco ocupados.

Necesitábamos determinar cuantos tests tenía cada uno, por lo que desarrollamos un programa que contara la cantidad de tests de unidad y de instrumentación.

3.6. Conteo de tests

3.6.1. Algoritmo de conteo de tests

A continuación se describe el funcionamiento del algoritmo:

- Recorrer todos los archivos de tipo *Java* o *Kotlin* en el repositorio y leer su contenido.
- Para cada archivo el algoritmo identifica los tests:
  - **Test de instrumentación**: se identifica por la presencia de la anotación **@Test** seguida por la cadena "**onView**" en algún lugar dentro del método.
  - **Tests de unidad**: se identifica simplemente por la presencia de la anotación **@Test**.

  Se ignoran las líneas precedidas por "//", ya que son líneas comentadas.
- Mantiene un recuento de los distintos tipos de test en cada archivo y suma estos valores para obtener el total de tests en el repositorio.

3.6.2. Ejecución del programa

Consideramos que en el proceso de desarrollo de software los programadores especifican la correctitud de una aplicación a partir de los tests. Una buena práctica es acompañar cada nueva funcionalidad de un proyecto con un conjunto de tests que prueben con cierto grado de certeza que el código nuevo es libre de errores. Es muy poco común que se eliminen tests de una aplicación si estos eran correctos. Por estas razones decidimos descartar los repositorios que no tuvieran tests en el commit más reciente de la rama principal, porque probablemente nunca los hubo. Por eso este programa fue ejecutado para el último commit de cada repositorio de la rama principal: *main* o *master* según correspondiera.

Al finalizar la ejecución descartamos más del 60 % de repositorios. Nos quedamos solo con los que tenían al menos un test, ya sea de unidad o de instrumentación. Luego de esta etapa nos quedamos con **1.207** repositorios que sumaban 278.536 commits.

3.7. Obtención de metadata de los commits

Buscamos encontrar las tuplas de commits que cumplieran las condiciones esbozadas en la sección **3.3**.

En un principio, enfrentamos desafíos para desarrollar un programa capaz de automatizar la ejecución de los tests en todos los commits. Estos desafíos surgían de la diversidad de configuraciones y distintas condiciones para compilar las aplicaciones. Entre los factores que añadían complejidad se incluían la selección de la versión adecuada de Java, la

necesidad de utilizar un emulador específico para las pruebas, y el manejo de aplicaciones con submódulos.

Dada la alta complejidad y el costo temporal asociado a esta tarea, optamos por restringir el espacio de nuestra búsqueda. Para hacerla más manejable, clasificamos los commits según ciertas variables que podrían indicarnos su idoneidad como candidatos. Este enfoque se justifica aún más si consideramos que la validación de un commit como candidato requiere un tiempo considerable, ya que implica compilar la aplicación y ejecutar sus tests para $commit_{bug}$, así como para $commit_{fix}$. Este proceso puede tardar desde un tiempo despreciable, hasta más de 15 minutos. Buscamos específicamente casos donde al menos un test falle en $commit_{bug}$ y pase en $commit_{fix}$. Con este fin, ideamos una función de ranking que nos permitiera priorizar de forma efectiva la selección de commits candidatos.

La información que consideramos necesaria obtener fue:

- El hash de cada commit.
- El mensaje del commit. Ciertos mensajes hacen alusión a arreglar un bug en particular.
- Cantidad de líneas insertadas y borradas del commit. Buscamos quedarnos inicialmente con los cambios que sean de una línea modificada.
- Los archivos modificados por el commit. Si el archivo modificado no era Java, no nos sirve como candidato. Algunos ejemplos de archivos que no sirven son los que tienen las siguientes extensiones: .kt, .txt, .MD, .yml, etc.
- La cantidad de tests por commit.
- Para cada commit, el inmediatamente anterior.

Intentamos utilizar herramientas ya desarrolladas para obtener la metadata asociada a los commits de los repositorios de GitHub. Encontramos *PyDriller*[10], una aplicación desarrollada en Python para recorrer la estructura de versionado de un repositorio y obtener información sobre los commits. Parecía cumplir todos nuestros requerimientos, pero luego de utilizarla con nuestro conjunto de datos, decidimos descartarla porque consumía mucho tiempo para obtener la información sobre cada repositorio.

Finalmente desarrollamos un programa capaz de obtener la información necesitada en un menor tiempo utilizando *GitPython*[11]. Esta herramienta nos permitió listar los commits de un repositorio, pero necesitamos agregar funcionalidad para obtener información más detallada de cada uno. La metodología fue la siguiente:

1. Obtener el listado de commits de la rama principal, *main* o *master* según correspondiera.
2. Recorrer de a tuplas ($commit_i$, $commit_{i+1}$) para calcular la diferencia entre cada commit consecutivo, para esto utilizamos el comando *git diff*[12].
3. Obtener la información asociada al diff del punto anterior.

[11] https://gitpython.readthedocs.io/en/stable/tutorial.html
[12] https://git-scm.com/docs/git-diff

Luego, aplicamos filtros para quedarnos con los commits que cumplieran las siguientes condiciones necesarias para ser candidatos:

- Tener tests: Cada repositorio presentaba variaciones en la cantidad de tests disponibles al examinarlo commit por commit. Adaptamos el programa originalmente diseñado para contar los tests en el commit más reciente de cada repositorio. La versión modificada realiza un conteo de tests para cada commit.
- Que los archivos modificados tengan extensión Java y que no fuera un archivo de test. Asumimos que todos los archivos que tenían *test* en el nombre contenían exclusivamente métodos relacionados al testing. De todas maneras, encontramos falsos negativos ya que habían casos de tests dentro de implementaciones de clases.
- Tener *bug*, *fix* o *test* como palabras claves en el mensaje del commit. Las primeras dos palabras son las utilizadas en DroidBugs[2], y agregamos la última ya que estábamos particularmente interesados en cambios que implicaran la reparación de un test. Tomamos esto como punto de partida para achicar el espacio de búsqueda.
- Que se modificara una sola línea. Esta condición está relacionada a la decisión de luego utilizar este benchmark para las herramientas de reparación automática de programas.

Realizamos dos etapas de filtrado:

- En la primera, seleccionamos los commits que tenían tests y que incluían palabras clave en sus mensajes. Pasamos de 1.207 repositorios conteniendo 278.536 commits a 53.345 commits, provenientes de 226 repositorios distintos.
- La segunda fase se centró en commits que modificaban una sola línea: nos quedamos finalmente con 1.198 commits de 100 repositorios.

3.8. Búsqueda manual en el conjunto de datos

En una primer instancia recorrimos manualmente varios de los 1.198 commits buscando candidatos. Para elegir qué commits revisar, realizamos un ranking contemplando:

- Primero, los repositorios con muchos commits. Nos pareció mejor empezar por las aplicaciones más abarcativas del conjunto de datos. Pensamos que de no poder correr los tests podíamos descartar mayor cantidad de commits en una menor cantidad de tiempo.
- Luego, los que no tenían una gran cantidad de tests. El proceso de ejecución de tests de estas aplicaciones debería ser más rápido.
- Finalmente, priorizamos los commits más nuevos ya que eran de mayor interés para nosotros.

Una vez establecido el ranking, examinamos manualmente más de 100 commits, una tarea que nos demandó alrededor de 80 horas. Este proceso consistió en:

1. Resolver desafíos asociados con la compilación de las aplicaciones y la ejecución de sus tests:

   - Cambiar la versión del Android SDK.
   - Modificaciones en los repositorios de dependencias[13].
   - Configuraciones de Linting que eran bloqueantes para la ejecución de los tests.
   - Asignación de permisos de ejecución a archivos de Gradle.
   - Configurar la JDK (8, 11, etc) a utilizar.
   - Cambios para que no se ejecuten ciertas tareas de Gradle que conflictuaban con el proceso de compilación.
   - Modificar versiones de dependencias.
   - Utilizar un emulador para correr los tests de instrumentación.

2. Analizar los resultados de ejecución. Los caracterizamos de la siguiente manera:

   - Alguno de los commits de la tupla candidata no compilaba a pesar de los esfuerzos dedicados.
   - Pasaban todos los tests para ambos commits de la tupla candidata ($commit_{fix}$, $commit_{bug}$).
   - Habían tests que fallaban, pero eran exactamente los mismos en ambos commits.
   - Habían errores de sintaxis en el código de algunos de los commits.

Luego de este proceso no encontramos ninguna tupla de commits que cumpliera la condicion necesaria de tener un test fallido en $commit_{bug}$ que pasara en $commit_{fix}$.

Descartamos una gran cantidad de commits antiguos. Notamos que usualmente, a partir de cierto momento en el tiempo, no es posible resolver las dependencias necesarias para compilar la aplicación. En el siguiente gráfico podemos ver la distribución por año de nuestro conjunto de datos:

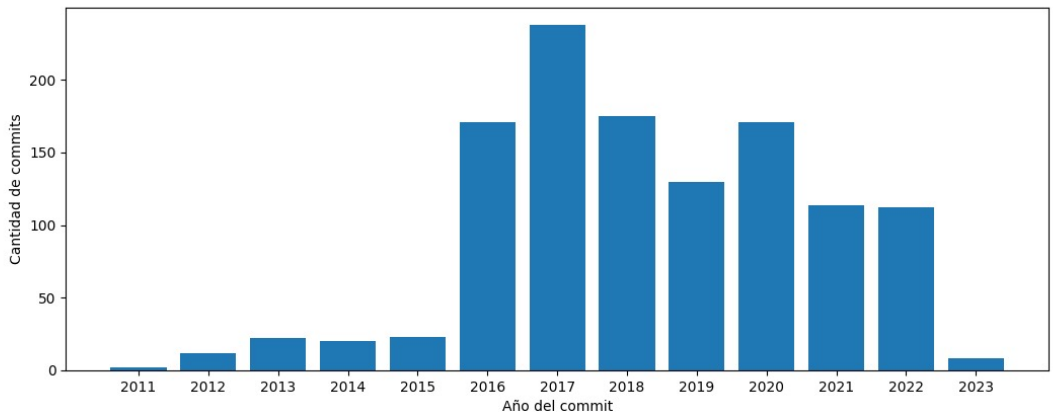

Fig. 3.2: Commits por año

[13] MavenCentral (https://mvnrepository.com/repos/central), JCenter, etc.

Como hasta el momento no habíamos obtenido resultados satisfactorios, nos propusimos hacer un análisis sobre los mensajes de los commits buscando que se mencione explicitamente el arreglo de algún test. De esta manera encontramos los primeros 2 casos confirmados del dataset final:

1. El bug ***Nextcloud_deck-Regex-1***[14] cuyo mensaje de commit es "*#707 fixed test*".
2. El bug ***Niccokunzmann-Operator-1***[15] cuyo mensaje de commit es "*Make connected components test run*" .

Como no encontramos más casos confirmados buscando de esta manera, decidimos automatizar el proceso de búsqueda.

3.9. Automatización de la búsqueda

3.9.1. Desarrollo del programa

Aprovechamos todo el aprendizaje de la etapa anterior para desarrollar un programa que automatice la búsqueda. Algo importante a tener en cuenta es el tiempo que conlleva compilar una aplicación y ejecutar sus tests. Observamos que, en promedio, este proceso suele tardar entre 2 y 15 minutos por commit. Toda la ejecución se realizó en computadoras MacBook Pro 2020, con 16GB de memoria RAM y procesador Intel Core i5 2 GHz Quad-Core.

El programa que desarrollamos opera de la siguiente manera:

- Parte de una lista de tuplas de commits candidatas.
- Modifica temporalmente los archivos *build.gradle* para que la aplicación compile a pesar de tener advertencias del linter[11]. Particularmente añadir el siguiente bloque de código en la cláusula *android* en caso de no existir:

```
android {
    ...
    lintOptions {
        abortOnError false
    }
    ...
}
```

- Verifica qué versión de JDK es necesaria para cada ejecución y la configura adecuadamente.
- Ejecuta los tests para $commit_{bug}$ utilizando **./gradlew test**. La ejecución sigue en base al resultado:
    - Si $commit_{bug}$ no compila, o todos los tests ejecutan correctamente, se descarta esta tupla y se pasa a la siguiente.

[14] https://github.com/niccokunzmann/coloring-book/commit/42a3418b6842ba7fc73aa639c80a138fff33e41e
[15] https://github.com/stefan-niedermann/nextcloud-deck/commit/559681f7c58380d2e4aeeabcfa03a2b8940503c9

- De haber fallado al menos un test en $commit_{bug}$, corre los tests para $commit_{fix}$. Si por el cambio introducido pasa alguno de los tests previamente fallidos entonces esta tupla será un caso confirmado. Si un test pasaba en $commit_{bug}$ no puede fallar en $commit_{fix}$.

- Se avanza con la siguiente tupla candidata hasta finalizar la lista.

Guarda el resultado de la ejecución de **./gradlew test** para cada commit. Esto nos permitió detectar automáticamente cuantos tests fallan por commit, y también hacer algunos análisis de errores más comunes, entre otros aspectos analizados.

3.9.2. Ejecución

Este programa fue ejecutado para las 1.198 tuplas candidatas, encontrando 6 confirmadas: las 2 que habíamos encontrado manualmente y 4 adicionales. La ejecución para el total del conjunto de datos transcurrió a lo largo de más de 120 horas, dado que el proceso es lento: además del costo computacional de correr los tests y compilar la aplicación, muchas veces se necesitaban dependencias pesadas que no estaban descargadas.

Como información adicional, destacamos que luego de haber ejecutado el programa para todos el conjunto de datos, el directorio .gradle/caches, donde se guardan todas las dependencias de las aplicaciones, ocupaba 50 gigabytes en disco.

Al finalizar esta parte del proceso contamos con un benchmark de 6 tuplas confirmadas. Consideramos que esta cantidad no era suficiente por lo que decidimos agrandar el espacio de búsqueda.

Decidimos relajar el requerimiento de cantidad de líneas modificadas para el $commit_{fix}$. Además de considerar candidatos a los commits con una sola línea modificada, es decir, una inserción y una eliminación en el *diff* de git, agregamos los siguientes casos:

- 2 inserciones y 2 eliminaciones
- 2 inserciones y 1 eliminación
- 2 inserciones y 0 eliminaciones
- 1 inserción y 0 eliminaciones
- 0 inserciones y 1 eliminación
- 0 inserciones y 2 eliminaciones

Esto nos permitió contar con 1.066 tuplas candidatas más. Ejecutamos el programa para este nuevo conjunto de datos y encontramos una tupla confirmada más.

Buscamos seguir agregando tuplas a nuestro benchmark. El primer filtro que aplicamos sobre el conjunto de commits inicial fue por palabras claves en el mensaje del commit. De esta manera habíamos pasado de 278.536 commits a 53.345, dejando de lado 225.191 commits. Dado que ya teníamos una herramienta en funcionamiento para determinar de manera automatizada si una tupla de commits podía ser parte de nuestro benchmark, decidimos agrandar el espacio de búsqueda utilizando este último conjunto de datos sin explorar. Nos quedamos con todos los commits que cumplieran la condición de tener entre 0 y 2 adiciones/eliminaciones. Esta expansión fue particularmente útil porque descubrimos casos en los que el mensaje del commit incluía el número del issue de GitHub relacionado

con la corrección del error, pero carecía de otras palabras clave que lo hubieran incluido en el conjunto de datos donde realizamos la primer búsqueda. Al aplicar este criterio más relajado, incorporamos 1.747 candidatos adicionales y, tras ejecutar nuestro programa, encontramos 2 tuplas confirmadas más. Con esta nueva iteración, llegamos a un total de 9 tuplas confirmadas.

Luego de todas estas ejecuciones, notamos que más del 90 % de ellas habían terminado en error, resultado que nos llamó mucho la atención. Como teníamos todos los resultados guardados en distintos archivos, realizamos un análisis y las principales causas detectadas fueron las siguientes:

- La mayoría fallaba por no poder resolver alguna dependencia. En general, mientras más viejo era el commit, más probable era este escenario. Una de las razones, es la discontinuación de JCenter[12], un popular repositorio de artefactos.
- Algunos no compilaban por errores de sintáxis en el código.
- Otros habían fallado por quedarse sin memoria RAM disponible. Esto se debía a una configuración errónea de la máquina virtual de java.
- Repositorios que contenían código en otros lenguajes que generalmente correspondía código de servidores o de páginas web.
- Aplicaciones que no usaban Gradle para compilar.
- Aplicaciones complejas con submódulos que no respetaban una estructura tradicional de proyecto.

Para el punto referido a la memoria RAM, realizamos un cambio en el programa para que modifique temporalmente el archivo *gradle.properties* de cada commit, asignándole más memoria dinámica a la aplicación. Puntualmente, esta asignación se realiza agregando la siguiente línea: *org.gradle.jvmargs=-Xmx4g*. En este ejemplo, se asignan 4 gigabytes al heap de la aplicación. Volvimos a ejecutar el programa para todos los commits que fallaron por este motivo, pero no encontramos más tuplas confirmadas.

Finalmente, sumando todas las tuplas confirmadas, llegamos a 9. Dado que el principal interés de la construcción de este benchmark está relacionado a utilizar herramientas de reparación de programas decidimos no modificar las restriccion del tamaño del diff de Git. Habiendo exhaustado el conjunto de datos buscamos otras formas de añadir más tuplas.

3.10. Sintetización de tests

Buscamos poder incluir más tuplas al benchmark. Para esto, evaluamos dos estrategias posibles: introducir bugs de forma manual mediante mutaciones en el código testeado o buscar bugs reales para los cuales pudiéramos escribir tests que cumplieran las condiciones de fallar en $commit_{bug}$, y pasar en $commit_{fix}$. Elegimos la segunda opción, ya que la consideramos más representativa de escenarios reales porque no implicaba alterar el código fuera del ámbito de los tests.

Todas las tuplas que incluimos cumplen con la condición de contener commits inmediatos. Para esta etapa usamos solo los commits que no habían tenido problemas de compilación en la etapa anterior. Luego de intentar escribir tests para 357 commits dedicando proximadamente 70 horas, logramos sintetizar 12 tuplas más, escribiendo un test que falle en $commit_{bug}$, y pase en $commit_{fix}$ para cada una.

Para escribir tests, nos apoyamos en las siguientes bibliotecas, muy utilizadas en la comunidad:

- **JUnit**: Es una biblioteca para pruebas unitarias en Java que permite a los desarrolladores validar la funcionalidad de su código. Ofrece anotaciones y aserciones para facilitar la escritura y ejecución de pruebas.
- **Mockito**: Permite la creación de *mocks*[16] para distintas clases. Los mocks se utilizan para imitar el comportamiento de las dependencias complejas y verificar cómo interactúa con ellas el código a prueba. No puede mockear clases del SDK de Android.
- **Robolectric**: Es un *framework* de tests unitarios especialmente diseñado para Android. Permite ejecutar tests directamente en la Java Virtual Machine, en lugar de en un emulador o dispositivo físico, debido a que simula el SDK de Android. Esto significa que los tests se pueden ejecutar rápidamente y en paralelo.

A pesar de contar con estas bibliotecas, nos encontramos con varios problemas a la hora de realizar esta tarea, los más comunes fueron:

- Muchos commits no modifican comportamiento, sino comentarios del código, cuestiones relacionadas al linting, etc. Todos estos fueron descartados ya que no hay test posible para escribir. Por ejemplo:

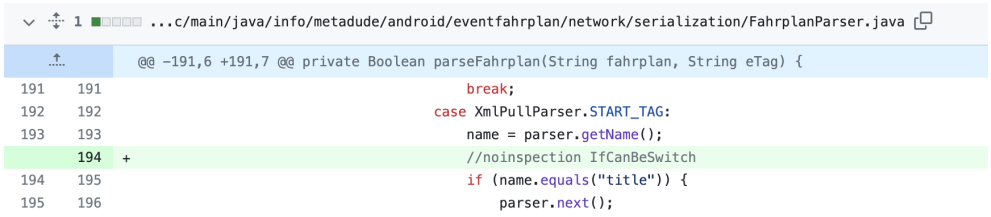

Fig. 3.3: Commit que agrega un comentario.

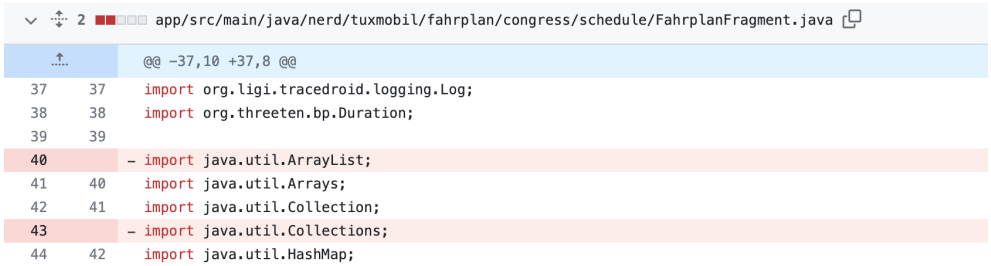

Fig. 3.4: Commit que elimina dependencias no utilizadas.

- El cambio en $commit_{fix}$ se realizó sobre un método u objeto que no era testeable sin modificar el código fuente. Por ejemplo, por no permitir inyección de dependencias.
- El código era muy díficil de entender, por lo que escribir un test válido podía requerir muchas horas de trabajo: el tiempo que llevara entender el código

[16] https://en.wikipedia.org/wiki/Mock_object

- Jerarquías de clases muy complejas.
- Funciones con muchas líneas de código y que no siguen buenas prácticas.
- Necesidad de conocer en detalle el dominio de la aplicación. Por ejemplo: El siguiente bloque de código contiene un cambio de una sola línea, pero es dificil entender qué implica para poder escribir un test que valide su comportamiento.

```
2 app/src/main/java/me/ccrama/redditslide/ImageFlairs.java
@@ -40,8 +40,8 @@ public static void syncFlairs(final Context context, final String subreddit) {
40 40            @Override
41 41            protected void onPostExecute(FlairStylesheet flairStylesheet) {
42 42                super.onPostExecute(flairStylesheet);
   43 +              d.dismiss();
43 44                if(flairStylesheet != null) {
44    -               d.dismiss();
45 45                    flairs.edit().putBoolean(subreddit.toLowerCase(), true).commit();
46 46                    d = new AlertDialogWrapper.Builder(context).setTitle("Subreddit flairs synced")
47 47                            .setMessage("Slide found and synced " + flairStylesheet.count + " image flairs")
```

- Instanciar objetos muy complejos para escribir el test.

```
2 app/src/main/java/org/secuso/privacyfriendlysudoku/ui/GameActivity.java
@@ -306,7 +306,7 @@ public void onClick(DialogInterface dialog, int id) {
306 306                   finish();
307 307                   overridePendingTransition(0, 0);
308 308               }
309    -              gameSolved = savedInstanceState.getInt("gameSolved") == 1;
   309 +              gameSolved = savedInstanceState.getBoolean("gameSolved");
310 310           }
```

Fig. 3.5: Es costoso construir una instancia de *savedInstanceState* con el atributo *gameSolved*.

- Testear una función privada, a través de funciones públicas. Configurar todo lo necesario para realizar el llamado a la función privada, se tornaba realmente complejo.

```
2 opentasks/src/main/java/org/dmfs/tasks/TaskListFragment.java
@@ -503,7 +503,7 @@ private void selectChildView(ExpandableListView expandLV, int groupPosition, int
503 503           }
504 504
505 505           // TODO For now we get the id of the task, not the instance, once we support recurrence we'll have to change that, use instance URI
506    -          Uri taskUri = ContentUris.withAppendedId(Tasks.getContentUri(mAuthority), (long) TaskFieldAdapters.TASK_ID.get(cursor));
   506 +          Uri taskUri = ContentUris.withAppendedId(Tasks.getContentUri(mAuthority), (long) TaskFieldAdapters.INSTANCE_TASK_ID.get(cursor));
507 507           Color taskListColor = new ValueColor(TaskFieldAdapters.LIST_COLOR.get(cursor));
508 508           mCallbacks.onItemSelected(taskUri, taskListColor, force, mInstancePosition);
```

Fig. 3.6: Llamar a la función *selectChildView* para testearla era muy complejo.

Estos desafíos exponen la complejidad de la tarea y son un factor clave en la baja tasa de éxito obtenida, con solo 12 tuplas sintetizadas de un total de 357 intentos. En la siguiente sección, presentaremos los distintos bugs que conforman nuestro benchmark.

4. BENCHMARK

4.1. Características de las aplicaciones

El benchmark cuenta con 21 commits de 11 aplicaciones distintas. 9 de ellos fueron encontrados semi-automáticamente, y 12 incluyeron tests sintetizados por nosotros.

Las aplicaciones son:

App	Repositorio
Coloring-book	https://github.com/niccokunzmann/coloring-book/
Nextcloud-deck	https://github.com/stefan-niedermann/nextcloud-deck/
Hydra	https://github.com/ZeusWPI/hydra-android/
Notes	https://github.com/nextcloud/notes/
GPSLogger	https://github.com/mendhak/gpslogger/
Markor	https://github.com/gsantner/markor/
Dicio assistant	https://github.com/Stypox/dicio-android/
Catima	https://github.com/CatimaLoyalty/Android/
OctoDroid	https://github.com/slapperwan/gh4a/
Tutanota	https://github.com/tutao/tutanota/
FinanceManager	https://github.com/SecUSo/privacy-friendly-finance-manager/

Tab. 4.1: Aplicaciones.

Las estadísticas de las aplicaciones son:

App	Descripción	Líneas	Tests	Stars	Commits	Descargas
Coloring-book	Libro de colorear	51.352	40	25	312	N/A
Nextcloud-deck	Planificación	98.243	74	443	4.258	10K+
Hydra	App universitaria	88.835	202	20	3.032	10K+
Notes	Notas	44.230	89	545	2.480	10K+
GPSLogger	Información del GPS	122.211	162	1.800	2.811	50+
Markor	Editor markdown	114.702	163	2.900	1.924	N/A
Dicio assistant	Asistente de voz	17.471	19	509	880	N/A
Catima	Tarjetas de membresía	58.019	92	563	4.374	50K+
OctoDroid	Cliente de GitHub	61.052	73	1600	3.037	N/A
Tutanota	E-mails encriptados	826.661	19	5.500	8.840	500K+
FinanceManager	Finanzas	17.298	1	35	551	1K+

Tab. 4.2: Estadísticas de las aplicaciones. El campo N/A se utiliza para cuando no se encuentra la aplicación en la Google Play Store.

4.2. Lista de bugs

A continuación listamos los bugs. Los nombres fueron construidos en base al nombre del repositorio y una categorización arbitraria del tipo de error:

Para cada bug listamos la cantidad de líneas que tiene la función defectuosa:

Nombre	$commit_{fix}$	Sintetizado
Nextcloud_deck-Regex-1	559681f7c58380d2e4aeeabcfa03a2b8940503c9	No
Nextcloud_deck-Regex-2	99a5c11fc1b16ec71a006f9dae4b9d8877f73ae5	No
Gsantner_markor-Regex-1	1ad8150755c4b886d6d6b1f6aab542310bf50bae	Sí
Stypox_dicio_android-Regex-1	400582cabf268a38ac083ee2e2da89086f4c6e12	Sí
Nextcloud_notes-NPE-1	00f06e930b64dbc262ba50c726b7c5d1581c53bf	No
Nextcloud_deck-NPE-1	2b1f44bf13d33ede572ee4f7a2239f26a3f0f275	Sí
Woheller_weather-Operator-1	485157a90f8e9222f0c49fbea028fe64bb681a93	Sí
Nextcloud_deck-Operator-1	9e4050a1b2fb12599e273415929449cdf18928af	Sí
Niccokunzmann-Operator-1	42a3418b6842ba7fc73aa639c80a138fff33e41e	No
Niccokunzmann-Operator-2	b5942315aaf8f1fc024652a17bf3230288909f45	Sí
CatimaLoyalty-Miscelánea-1	e1f4ed65bb0c53d1a6c17de36daf67e870cc7245	No
CatimaLoyalty-Miscelánea-2	24061dae974311f95c666eb8921d3a08debbe8e7	Sí
CatimaLoyalty-Miscelánea-3	401fc98b4dc770499185cdee110681af1173453d	No
CatimaLoyalty-Miscelánea-4	184af4d272d663d200688bd73d4bbb9ad1302536	Sí
Gsantner_markor-Miscelánea-1	2849cb42d651cab9fe37a27be265d498bff99d7e	Sí
Gsantner_markor-Miscelánea-2	934c3852597f802423c5ccd79814f553564f1c7c	Sí
ZeusWPI-Miscelánea-1	7d1aca20a8001763eb9f60e4f7313ef4962091e0	No
Mendhak_gpslogger-Miscelánea-1	5d145ce4e018e84eea0f5e28be73984cb5b6473a	No
Slapperwan-Miscelánea-1	e9e604ba8383585a838692394e5e9f3d86cff6e9	Sí
Tutao_tutanota-Miscelánea-1	3aad19ab276ea9d70d500464ff44e5aabda885c4	No
SecUso-Miscelánea-1	13997b144310c4080d50633e266b6ba112c3f344	Sí

Tab. 4.3: Listado de bugs del benchmark.

Nombre	Cantidad de líneas
Nextcloud_deck-Regex-1	45
Nextcloud_deck-Regex-2	45
Gsantner_markor-Regex-1	1
Stypox_dicio_android-Regex-1	1
Nextcloud_notes-NPE-1	20
Nextcloud_deck-NPE-1	36
Woheller_weather-Operator-1	13
Nextcloud_deck-Operator-1	5
Niccokunzmann-Operator-1	51
Niccokunzmann-Operator-2	4
CatimaLoyalty-Miscelánea-1	194
CatimaLoyalty-Miscelánea-2	194
CatimaLoyalty-Miscelánea-3	37
CatimaLoyalty-Miscelánea-4	49
Gsantner_markor-Miscelánea-1	20
Gsantner_markor-Miscelánea-2	18
ZeusWPI-Miscelánea-1	23
Mendhak_gpslogger-Miscelánea-1	5
Slapperwan-Miscelánea-1	1
Tutao_tutanota-Miscelánea-1	12
SecUso-Miscelánea-1	4

Tab. 4.4: Cantidad de líneas por función defectuosa.

4.3. Características de los bugs

A continuación hacemos una descripción de los distintos bugs que componen el benchmark, agrupados por tipo de error.

4.3.1. Expresiones regulares

Podemos tomar de ejemplo al bug ***Nextcloud_deck-Regex-1***:

2 app/src/main/java/it/niedermann/nextcloud/deck/util/ProjectUtil.java

```
@@ -50,7 +50,7 @@ public static long[] extractBoardIdAndCardIdFromUrl(@Nullable String url) throws
50  50          throw new IllegalArgumentException("trimmed url is empty");
51  51      }
52  52      // extract important part
53      -   String[] splitByPrefix = url.split(".*index\\.php/apps/deck/#/board/");
    53  +   String[] splitByPrefix = url.split(".*index\\.php/apps/deck(/#)?/board/");
54  54      // split into board- and card part
55  55      if (splitByPrefix.length < 2) {
```

Fig. 4.1: Bug utilizando un parámetro opcional al escribir una expresión regular.

La expresión regular fallaba al no considerar el caracter # como opcional y ser una opción válida de URL.

La aserción que fallaba era la siguiente:

```
assertArrayEquals(ProjectUtil.extractBoardIdAndCardIdFromUrl("index.php/apps/deck/board/4"), new long[]{4});
```

Fig. 4.2: Línea del test que falla en ***Nextcloud_deck-Regex-1***

Hay 3 bugs más análogos:

- ***Nextcloud_deck-Regex-2***.
- ***Gsantner_markor-Regex-1***.
- ***Stypox_dicio_android-Regex-1***.

4.3.2. Null pointer exception

Podemos tomar de ejemplo al bug ***Nextcloud_notes-NPE-1***:

2 markdown/src/main/java/it/niedermann/android/markdown/MarkdownUtil.java

```
@@ -95,7 +95,7 @@ private static String getCheckboxEmoji(boolean checked) {
95  95          final String[] uncheckedEmojis;
96  96          // Seriously what the fuck, Samsung?
97  97          // https://emojipedia.org/ballot-box-with-x/
98      -       if(Build.MANUFACTURER.toLowerCase().contains("samsung")) {
    98  +       if(Build.MANUFACTURER != null && Build.MANUFACTURER.toLowerCase().contains("samsung")) {
99  99              checkedEmojis = new String[]{"✅", "☑", "✔"};
100 100             uncheckedEmojis = new String[]{"❌", "\uD83D\uDD32", "☐"};
```

Fig. 4.3: Bug por obviar que el objeto invocado puede ser *null*.

Se le aplican al objeto *Build.Manufacturer* funciones que fallan si el parámetro recibido es *null*. El test que fallaba era el siguiente:

```
public void testIsEmptyLine() {
    assertTrue(NoteUtil.isEmptyLine(" "));
    assertTrue(NoteUtil.isEmptyLine("\n"));
    assertTrue(NoteUtil.isEmptyLine("\n "));
    assertTrue(NoteUtil.isEmptyLine(" \n"));
    assertTrue(NoteUtil.isEmptyLine(" \n "));
    assertFalse(NoteUtil.isEmptyLine("a \n "));
}
```

Hay 1 bug más análogo:

- ***Nextcloud_deck-NPE-1***.

4.3.3. Operadores aritméticos-lógicos

Woheller_weather-Operator-1:

2 app/src/main/java/org/woheller69/weather/widget/WeatherWidget.java

```
@@ -55,7 +55,7 @@ public static int getWidgetCityID(Context context) {
55 55          for (int i = 0; i < cities.size(); i++) {   //find cityID for first city to watch = lowest
56 56              CityToWatch city = cities.get(i);
57 57              //Log.d("debugtag",Integer.toString(city.getRank()));
58    -           if (city.getRank() < rank ){
   58 +           if (city.getRank() <= rank ){
59 59                  rank=city.getRank();
60 60                  cityID = city.getCityId();
```

Fig. 4.4: Bug por aplicar de manera incorrecta un operador.

Se quiere obtener el ID mínimo de una lista y por aplicar mal el operador no se consigue el resultado esperado cuando el elemento a buscar es el primero de la lista. Se sintetizo un test que evidencia este mismo caso.

Niccokunzmann-Operator-1:

En el siguiente bug se utiliza de manera equivocada el índice de acceso a un array.

2 src/main/java/eu/quelltext/clustering/ConnectedComponents.java

```
@@ -31,7 +31,7 @@ public Result compute() {
31 31                  if (x > 0 && classified[x-1 + y * width] == thisValue) {
32 32                      if (y > 0 && classified[x + (y-1) * width] == thisValue) {
33 33                          /* pixel is equal to left and top */
34    -                     int leftLabel = area[x + y * width];
   34 +                     int leftLabel = area[x-1 + y * width];
35 35                          int topLabel = area[x + (y-1) * width];
36 36                          area[x + y * width] = leftLabel;
37 37                          if (leftLabel == topLabel) {
```

Fig. 4.5: Bug de acceso a un array.

El test correspondiente a este bug consiste en generar dos matrices de elementos y verificar que sean iguales al aplicarles la función *compute*.

Niccokunzmann-Operator-2:

En el siguiente bug no se tiene en cuenta la posibilidad de causar un overflow sobre una variables de tipo *Integer*.

2 src/main/java/org/androidsoft/coloring/ui/widget/PaintArea.java

```
@@ -131,7 +131,7 @@ public void setPaintColor(int color)
131 131     {
132 132         paintColor = color;
133 133         if (paintColor == FloodFill.BORDER_COLOR) {
134     -           paintColor ++;
    134 +           paintColor = paintColor ^ 1;
135 135         }
136 136     }
```

Fig. 4.6: Bug de overflow.

El test correspondiente a este bug fue sintetizado y busca simular un *overflow* sobre la variable *paintColor*.

Hay 1 bug más análogo:

- ***Nextcloud_deck-Operator-1***: El fix consiste en cambiar un *and* por un *or*.

4.3.4. Miscelánea

Enumeramos el resto de los bugs que no pudimos categorizar, es decir, lograr una generalización de ese caso particular.

ZeusWPI-Miscelánea-1:

En el siguiente bug no se verifica que el valor de una variable sea positivo para retornarlo. El error causa que fechas pasadas se vean como futuras

4 app/src/main/java/be/ugent/zeus/hydra/utils/DateUtils.java

```
@@ -73,9 +73,9 @@ public static String getFriendlyDate(Context context, @NonNull LocalDate date, F
73 73              return context.getString(R.string.date_tomorrow);
74 74          } else if (daysBetween == 2 && context.getResources().getBoolean(R.bool.date_supports_overmorrow)) {
75 75              return context.getString(R.string.date_overmorrow);
76     -        } else if (daysBetween < ONE_WEEK_DAYS) {
    76 +        } else if (0 <= daysBetween && daysBetween < ONE_WEEK_DAYS) {
77 77              return date.getDayOfWeek().getDisplayName(TextStyle.FULL_STANDALONE, locale);
78     -        } else if (daysBetween < TWO_WEEKS_DAYS && context.getResources().getBoolean(R.bool.date_supports_next)) {
    78 +        } else if (0 <= daysBetween && daysBetween < TWO_WEEKS_DAYS && context.getResources().getBoolean(R.bool.date_supports_next)) {
79 79              return context.getString(R.string.date_next_x, date.getDayOfWeek().getDisplayName(TextStyle.FULL, locale));
```

Fig. 4.7: Bug de verificación de posibles valores de una variable.

El test correspondiente a este bug consiste en verificar que las fechas pasadas no sean mostradas como fechas futuras.

CatimaLoyalty-Miscelánea-1:

En el siguiente bug no se cumple la condición de una guarda por no convertir el valor que devuelve una función al tipo de dato *String*.

2 app/src/main/java/protect/card_locker/LoyaltyCardEditActivity.java

```
@@ -231,7 +231,7 @@ else if(importLoyaltyCardUri != null)
231 231
232 232          if(cardIdFieldView.getText().length() > 0 && barcodeTypeField.getText().length() > 0)
233 233          {
234     -            if(barcodeTypeField.getText().equals(NO_BARCODE))
    234 +            if(barcodeTypeField.getText().toString().equals(NO_BARCODE))
235 235              {
236 236                  hideBarcode();
```

Fig. 4.8: Bug de conversión de variable.

El test correspondiente a este bug consiste en generar el objeto *barcodeType* y verificar que su valor sea igual a uno obtenido en una base de datos.

CatimaLoyalty-Miscelánea-2:

En el siguiente commit se arregló la falla que había en la aplicación, al mismo tiempo que se agregó el test correspondiente al error.

1 app/src/main/java/protect/card_locker/LoyaltyCardEditActivity.java

```
@@ -208,6 +208,7 @@ else if(importLoyaltyCardUri != null)
208 208          else
209 209          {
210 210              setTitle(R.string.addCardTitle);
    211 +            hideBarcode();
211 212          }
212 213
213 214          if(headingColorValue == null)
```

1 app/src/test/java/protect/card_locker/LoyaltyCardViewActivityTest.java

```
@@ -257,6 +257,7 @@ public void startWithoutParametersCheckFieldsAvailable()
257 257          Activity activity = (Activity)activityController.get();
258 258
259 259          checkAllFields(activity, ViewMode.ADD_CARD, "", "", "");
    260 +        assertEquals(View.GONE, activity.findViewById(R.id.barcodeTypeTableRow).getVisibility());
260 261      }
261 262
262 263      @Test
```

Fig. 4.9: Bug de funcionalidad

CatimaLoyalty-Miscelánea-3:

En el siguiente bug se hace el llamado a la función equivocada, en este caso un constructor:

2 app/src/main/java/protect/card_locker/importexport/CatimaImporter.java

```
@@ -48,7 +48,7 @@ public void importData(Context context, DBHelper db, InputStream input, char[] p
48 48          bufferedInputStream.mark(100);
49 49
50 50          // First, check if this is a zip file
51    -       ZipInputStream zipInputStream = new ZipInputStream(bufferedInputStream);
   51 +       ZipInputStream zipInputStream = new ZipInputStream(bufferedInputStream,password);
52 52
53 53          boolean isZipFile = false;
```

Fig. 4.10: Bug por aplicar de manera incorrecta un constructor.

El test que fallaba se fijaba que los valores generados fueran los correctos.

CatimaLoyalty-Miscelánea-4:

En el siguiente bug no se utiliza el atributo correcto de un objeto:

```
@@ -68,7 +68,7 @@ public void bindView(View view, Context context, Cursor cursor)
68          expiryField.setVisibility(View.VISIBLE);
69          int expiryString = R.string.expiryStateSentence;
70          if(Utils.hasExpired(loyaltyCard.expiry)) {
-               expiryString = R.string.expiryStateSentence;
71 +             expiryString = R.string.expiryStateSentenceExpired;
72              expiryField.setTextColor(context.getResources().getColor(R.color.alert));
73          }
74          expiryField.setText(context.getString(expiryString, DateFormat.getDateInstance(DateFormat.LONG).format(loyaltyCard.expiry)));
```

El atributo a utilizar era *R.string.expiryStateSentenceExpired* en vez de *R.string.expiryStateSentence*.

Mendhak_gpslogger-Miscelánea-1:

El siguiente bug es introducido por un error al cambiar el operador '=' por el '+ ='. Esto se puede evidenciar al ver los cambios introducidos en $commit_{bug}$ que luego fueron arreglados en $commit_{fix}$.

2 gpslogger/src/main/java/com/mendhak/gpslogger/common/Locations.java

```
@@ -50,7 +50,7 @@ public static Location getLocationAdjustedForGPSWeekRollover(Location loc) {
50 50          long recordedTime = loc.getTime();
51 51          //If the date is before April 6, 23:59:59, there's a GPS week rollover problem
52 52          if(recordedTime < 1554595199000L){
53    -           recordedTime += recordedTime;  //add 1024 weeks
   53 +           recordedTime += 619315200000L;  //add 1024 weeks
54 54              loc.setTime(recordedTime);
55 55          }
```

Fig. 4.11: Arreglo del bug introducido por el cambio de un operador.

En el commit inmediato anterior se realiza la modificación del operador y se introduce el bug:

2 gpslogger/src/main/java/com/mendhak/gpslogger/common/Locations.java

```
@@ -50,7 +50,7 @@ public static Location getLocationAdjustedForGPSWeekRollover(Location loc) {
50  50          long recordedTime = loc.getTime();
51  51          //If the date is before April 6, 23:59:59, there's a GPS week rollover problem
52  52          if(recordedTime < 1554595199000L){
53      -           recordedTime = recordedTime + 619315200000L;  //add 1024 weeks
    53  +           recordedTime += recordedTime;  //add 1024 weeks
54  54              loc.setTime(recordedTime);
55  55          }
```

Fig. 4.12: Bug previo que introduce el error.

El test correspondiente a este bug consiste en verificar que al *recordedTime* se le agreguen 1024 semanas al cumplirse la guarda.

SecUso-Miscelánea-1:

En el siguiente bug no se convierte correctamente un valor de tipo *String* a *Double*. No es posible mostrar correctamente el valor '*2.30*'

2 app/src/main/java/org/secuso/privacyfriendlyfinance/helpers/CurrencyHelper.java

```
@@ -72,7 +72,7 @@ private static String digitsOf(String input) {
72  72
73  73      public static Long convertToLong(String text) {
74  74          try {
75      -           return ((Double) (Double.parseDouble(text) * 100)).longValue();
    75  +           return Math.round(Double.parseDouble(text) * 100);
76  76          } catch (NumberFormatException e) {
77  77              return null;
78  78          }
```

Fig. 4.13: Bug de conversión de variable.

El test correspondiente a este bug fue sintetizado y busca mostrar correctamente la conversión de '2.30' a '230'.

Gsantner_markor-Miscelánea-1:

En el siguiente bug no se tiene en cuenta la última posición de un *string* como un valor posible para la variable.

2 app/src/main/java/net/gsantner/opoc/util/StringUtils.java

```
@@ -155,7 +155,7 @@ public static int getIndexFromLineOffset(final CharSequence s, final int l, fina
155 155                     }
156 156                 }
157 157             }
158     -           if (i < s.length() - 1) {
    158 +           if (i < s.length()) {
159 159                 final int start = (l == 0) ? 0 : i + 1;
160 160                 final int end = getLineEnd(s, start);
```

Fig. 4.14: Bug de valor posible de una variable.

El test correspondiente a este bug fue sintetizado y busca obtener el índice del último caracter de un *string*.

Gsantner_markor-Miscelánea-2:

En el siguiente bug no se respeta el orden que debe tener la aplicación al listar los directorios y archivos.

2 app/src/main/java/net/gsantner/opoc/util/GsFileUtils.java

```
@@ -695,7 +695,7 @@ public static String getFilenameWithTimestamp(final String... A0prefixA1postfixA
695 695         */
696 696        private static List<String> makeSortKey(final String sortBy, final File file, final boolean dirFirst) {
697 697            // If we want directories first we prefix with a 0 to increase priority
698     -          final String dirPrefix = file.isDirectory() ? "0" : "1";
    698 +          final String dirPrefix = (dirFirst && file.isDirectory()) ? "0" : "1";
699 699            // All sort conflicts resolved by name
700 700            final String name = file.getName().toLowerCase();
```

Fig. 4.15: Bug de listar valores.

El test correspondiente a este bug fue sintetizado y busca que se muestren primero los directorios por sobre los archivos.

Slapperwan-Miscelánea-1:

En el siguiente bug no se tiene en cuenta que los archivos con extensión '*.mjs*' deben ser reconocibles como archivos de texto en Android 11+.

1 app/src/main/java/com/gh4a/utils/FileUtils.java

```
@@ -33,6 +33,7 @@ public class FileUtils {
33 33           // JavaScript can be resolved to both text/javascript and application/javascript,
34 34           // for our purposes it's text in any case
35 35           MIME_TYPE_OVERRIDES.put("js", "text/javascript");
    36 +         MIME_TYPE_OVERRIDES.put("mjs", "text/javascript");
36 37           // Same for Ruby, LaTeX, SQL, JSON
37 38           MIME_TYPE_OVERRIDES.put("rb", "text/x-ruby");
38 39           MIME_TYPE_OVERRIDES.put("latex", "text/x-latex");
```

Fig. 4.16: Bug de formato de archivo.

El test correspondiente a este bug fue sintetizado y busca reconocer que un archivo con extensión '*.mjs*' sea de tipo '*test/javascript*'.

Tutao_tutanota-Miscelánea-1:

En el siguiente bug se realiza una equivocación sobre la verificación del formato hexadecimal de un tipo de color.

2 app-android/app/src/main/java/de/tutao/tutanota/Utils.java

```
@@ -145,7 +145,7 @@ public static Map<String, String> jsonObjectToMap(JSONObject jsonObject) throws
145 145         }
146 146
147 147         static boolean isColorLight(String c) {
148     -           if (c.charAt(0) != '#' || c.length() != 6) {
    148 +           if (c.charAt(0) != '#' || c.length() != 7) {
149 149                 throw new IllegalArgumentException("Invalid color format: " + c);
150 150             }
```

Fig. 4.17: Bug de verificación del valor de una variable.

El test correspondiente a este bug consiste en verificar que ciertos números hexadecimales sean colores claros.

5. ASTOR4ANDROID

5.1. Trabajo relacionado

En 3.1 mencionamos la existencia de **DroidBugs** y **DroixBench**. En esos trabajos, además de presentar conjuntos de datos, se presentan dos herramientas: **Astor4Android**[2] y **Droix**[6].

Astor[13]: Es una herramienta que realiza reparación automática de código Java. Usa técnicas del estilo *generate-and-validate*. El nombre de esta categoría de técnicas está directamente ligado a su funcionamiento: producen variantes en el código, buscando repararlo automáticamente. Cada variante es validada ejecutando la suite de tests y esperando que todos pasen. Todas están implementadas en Java, y basadas en técnicas implementadas en C.

Astor4Android: Según lo que pudimos investigar, es la única herramienta *open-source* que realiza reparación automática de aplicaciones Android. Es una adaptación de Astor. Explicaremos más en profundidad el funcionamiento de esta herramienta en la sección **5.5**.

Droix: Realiza reparación automática de *crashes* en aplicaciones Android. A diferencia de la mayoría de las herramientas de reparación de programas basadas en los resultados de ejecución de tests, solo necesita un test fallido, y no una *test-suite* completa. No encontramos esta herramienta publicada.

5.2. Introducción

Utilizando los $commit_{bug}$ de nuestro benchmark, evaluamos tanto **Astor4Android**[1] como la eficacia de las distintas técnicas de localización de fallas y reparación de programas que implementa.

Ejecutamos la herramienta para los 21 bugs del benchmark que describimos en **4.3**, pero logramos ejecutarla de manera exitosa solo para 8. En este capítulo enumeramos los inconvenientes que tuvimos al usarla, y las modificaciones que hicimos en su código fuente para resolverlos.

5.3. Localización de fallas

Las técnicas de localización de fallas buscan identificar automáticamente elementos defectuosos de un programa. Estas técnicas toman como entrada un programa defectuoso y un conjunto de tests, con al menos uno no exitoso. Generan como resultado una lista de elementos sospechosos del programa que suelen ser líneas de código, métodos o archivos.

Estas técnicas utilizan heurísticas para determinar qué partes del programa son las más sospechosas, es decir, las más propensas a ser defectuosas y estar asociadas con el *bug*. Las técnicas examinadas en este trabajo, e implementadas en Astor4Android, pertenecen a la categoría de enfoques *spectrum-based*. En este paradigma, la probabilidad de que un determinado elemento del código sea defectuoso se estima en función de los resultados de los tests ejecutados. Específicamente, un elemento se considera altamente sospechoso si se

[1] https://github.com/I4Soft/Astor4Android

encuentra frecuentemente involucrado en tests que fallan, mientras que su participación en tests exitosos es mínima.

Se define $S(s)$ como la función que cuantifica la probabilidad de sospecha relacionada con el elemento s en el código. Un mayor valor para $S(s)$ sugiere una probabilidad más alta de que s contenga un defecto. Denominamos *totalpassed* al número total de tests que finalizan exitosamente y *passed(s)* a aquellos tests exitosos que implican la ejecución del elemento s. Análogamente, definimos *totalfailed* y *failed(s)* para representar el número total de tests que fallan y los que involucran el elemento s, respectivamente.

Evaluaremos 5 técnicas distintas de localización de fallas, las implementadas en Astor4Android. Así define S(s) cada una de ellas:

Ochiai[14]	$S(s) = \frac{\text{failed}(s)}{\sqrt{\text{totalfailed} \times (\text{failed}(s) + \text{passed}(s))}}$
Op2[15]	$S(s) = \text{failed}(s) - \frac{\text{passed(s)}}{\text{totalpassed}+1}$
D*[16]	$S(s) = \frac{\text{failed(s)}^*}{\text{passed(s)} + (\text{totalfailed} - \text{failed(s)})}$
Tarantula[17]	$S(s) = \frac{\frac{\text{failed}(s)}{\text{totalFailed}}}{\frac{\text{failed}(s)}{\text{totalfailed}} + \frac{\text{passed}(s)}{\text{totalpassed}}}$
Barinel[18]	$S(s) = 1 - \frac{\text{passed(s)}}{\text{passed(s)} + \text{failed(s)}}$

A continuación, describimos la intuición detrás de cada técnica:

- **Ochiai**: Mide qué tan frecuentemente una línea de código está relacionada con tests fallidos en comparación con su presencia en todos los tests. Utiliza la media geométrica para equilibrar la influencia de cada test fallido en el cálculo final.
- **Op2**: En lugar de simplemente contar los tests que fallan o pasan, esta técnica evalúa su impacto en relación al número total de tests en el proyecto. Al hacerlo, Op2 evita que el análisis se distorsione, lo cual podría ser especialmente problemático en proyectos con una gran cantidad de tests.
- **D***: Prioriza la información proveniente de tests que fallan sobre la de los tests exitosos. La sospecha hacia una línea de código se incrementa con más tests fallidos y disminuye con más tests exitosos.
- **Tarantula**: Se centra en identificar las líneas de código que son más a menudo ejecutadas por tests fallidos que por los exitosos. A diferencia de otras técnicas, Tarantula es más tolerante si el código en cuestión también se ejecuta en tests que pasan.
- **Barinel**: Considera que un bug podría ser el resultado de interacciones entre múltiples líneas de código. Utiliza un enfoque probabilístico para estimar qué conjunto de líneas de código es más probable que haya causado los bugs observados, en lugar de señalar una sola línea como la culpable.

5.4. Reparación de programas

La reparación automática de programas es un subcampo de la ingeniería de software que se enfoca en corregir defectos en el código fuente sin intervención humana. Este área de investigación engloba una serie de técnicas que generalmente toman como entrada un programa defectuoso y una serie de tests que el programa no pasa. El objetivo es producir un nuevo programa que sea semánticamente similar al original pero que pase todos los tests.

Diferentes enfoques de reparación utilizan distintas heurísticas y métodos para llevar a cabo esta tarea. Para este trabajo evaluaremos tres técnicas de reparación de programas introducidas en Astor.

Las tres están implementadas en Java, y basadas en técnicas implementadas en C. El funcionamiento de cada una se describe a continuación:

- **jKali:** Basada en *Kali*[19]. Tomando las líneas sospechosas obtenidas por alguna técnica de localización de fallas, en orden descendente según el nivel de sospecha, aplica las siguientes estrategias:

  1. Eliminación de sentencias.
  2. Reemplazo de las guardas de las estructuras *if*, por verdadero o falso. Es decir, modificar *if(condicion)* por *if(true)* o *if(false)*.
  3. Inserción de sentencias *return*, con valores por defecto según el tipo de dato que retorna el método. Por ejemplo, 0 y -1 si es un entero, o null si es un puntero.

- **jMutRepair:** Basada en *MutRepair*[20]. Aplica operadores de mutación en las guardas sospechosas de estructuras *if*, realizando una sola modificación a la misma. Esta implementación cuenta con 3 tipos de operadores de mutación:

  1. Relacionales: $>$, $\geq$, $<$, $\leq$, $==$ y $\neq$
  2. Lógicos: *AND* y *OR*.
  3. Unarios: negación, y eliminación del operador de negación.

- **jGenProg2:** Basada en *GenProg*[21]. Utiliza programación genética con el objetivo de encontrar una variante que corrija los bugs. Parte de la hipótesis de que un programa que contiene un bug, probablemente implemente el comportamiento correcto en otro área, por lo que solo utiliza el código existente de la aplicación para sintetizar soluciones candidatas, no inventa código. Para GenProg, la unidad primitiva son las sentencias de código, por lo que sus operadores de mutación son: eliminación/inserción de una sentencia o intercambio con otra.

5.5. Funcionamiento

Astor4Android busca aplicar técnicas de localización de fallas y reparación de programas a una aplicación, esperando poder localizar las líneas defectuosas y solucionar los bugs. Permite ejecutar los distintos algoritmos de localización de fallas y de reparación de programas previamente descritos.

Para ejecutar la herramienta debemos configurar varios parámetros. Estos están especificados en la sección **5.6.2**.

Astor4Android lleva a cabo varios pasos para la reparación de aplicaciones. Primero compila la aplicación a reparar, luego ejecuta todos los tests de la misma y obtiene datos de cobertura[2], las líneas de código alcanzadas por los distintos tests. Es importante notar que utiliza un método de compilación que difiere con el que se usa al compilar con *./gradlew build*. Un ejemplo de esta diferencia es que, en el contexto de Astor4Android, los tipos "Integer" e "int" se consideran incompatibles.

Con la información de cobertura, la técnica de localización de fallas elegida puede calcular S(s). Luego, la herramienta analiza la lista de sentencias sospechosas, y descarta las que estén debajo de cierto umbral de sospecha, definido al ejecutar la herramienta. Las que superen el umbral, ordenadas de mayor a menor por sospecha, se utilizan como entrada para el algoritmo de reparación de programas seleccionado. Este algoritmo intentará generar variantes candidatas, y si alguna logra que pasen todos los tests, la herramienta la considerará como una solución al bug. La figura **5.1** ilustra este proceso.

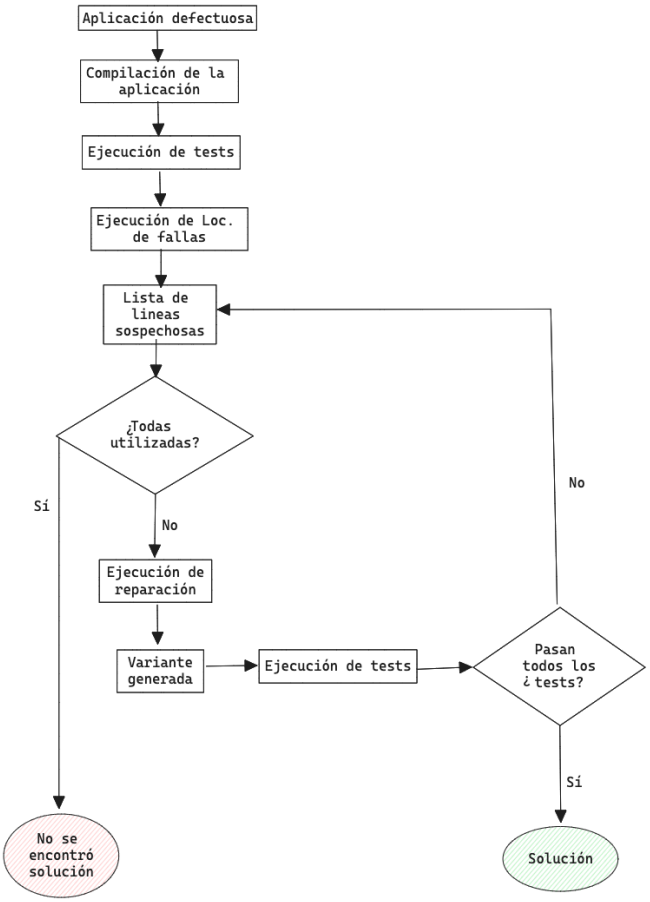

Fig. 5.1: Flujo de ejecución.

[2] Se utiliza la herramienta JaCoCo: https://www.jacoco.org/jacoco/

5.6. Ejecución

5.6.1. Requisitos de ejecución

Para ejecutar Astor4Android, hay que asegurarse de que tanto el ambiente de desarrollo como las aplicaciones Android a reparar cumplan con un conjunto de requisitos específicos. Los requisitos fundamentales se listan a continuación:

- Contar con Maven[3], JDK 8 y Android SDK instaladas.
- Tener instalado un emulador de Android, junto con alguna imagen del sistema operativo Android, con versiones compatibles a los requisitos de las aplicaciones a reparar.
- Que el código a reparar pueda ejecutarse utilizando JDK 8. Por limitaciones de Astor, Astor4Android no funciona con aplicaciones que exijan una versión superior [22]. Como a partir de la versión 7.0 del Android Gradle Plugin (AGP) se exige utilizar JDK 11 o superior, las aplicaciones a reparar deben utilizar una versión de AGP menor [23].
- Que la aplicación siga cierta estructura de directorios y archivos: que el código a reparar esté dentro de un directorio denominado *app*, dentro del directorio raíz, y tener al menos dos archivos *build.gradle*, uno en el directorio raíz, otro en *app*.

5.6.2. Cómo ejecutar Astor4Android

Esta herramienta consta de dos ejecutables: Astor4Android[4] y AstorWorker[5]. Ambos están desarrollados en Java y utilizan Maven.

Para reparar una aplicación, necesitamos ejecutarlos a ambos. Primero, AstorWorker, que recibe los siguientes parámetros por línea de comandos:

- **hostip**: La dirección IP desde donde se ejecutará Astor4Android. Usualmente se elige la dirección 127.0.0.1, correspondiente a *localhost*.
- **hostport**: El puerto desde donde se ejecutará Astor4Android.
- **workerip**: La dirección IP desde donde se ejecutará este worker. Usualmente se elige la dirección 127.0.0.1, correspondiente a *localhost*.
- **workerport**: El puerto desde donde se ejecutará este worker.
- **androidsdk**: Ruta local donde está instalada la SDK de Android

Por lo que, el comando para ejecutarlo es:

```
mvn exec:java -Dexec.mainClass=br.ufg.inf.astorworker.main.AstorWorker
-Dexec.args="<arguments>"
```

[3] https://maven.apache.org/
[4] https://github.com/I4Soft/Astor4Android
[5] https://github.com/kayqueteixeira/astorworker

donde $< arguments >$ corresponde a los parámetros previamente listados.

Por otro lado, Astor4Android cuenta con los siguientes parámetros configurables por línea de comandos:

- **mode**: Algoritmo de reparación de programas. Se debe elegir entre *statement*, *mutation* y *statement-remove*. Representan *JGenProg*, *JMutRepair* y *Jkali* respectivamente.
- **flmode**: Algoritmo de localización de fallas. Se debe elegir entre *ochiai*, *op2*, *dstar*, *tarantula* y *barinel*.
- **location**: Ruta local del directorio que contiene el código de la aplicación a reparar.
- **unitfailing**: Nombre de los tests fallidos.
- **flthreshold**: Límite inferior para considerar una línea como sospechosa. Las técnicas de localización de fallas asignan probabilidades a distintas líneas sospechosas. Para reparar el programa, solo se tendrán en cuenta las que tengan probabilidad mayor a este límite.
- **stopfirst**: Valor booleano que indica si al encontrar la primera variante que haga que la ejecución de todos los tests sea exitosa, se finaliza la ejecución, o si se debe seguir procesando para buscar más candidatos a soluciones.
- **androidsdk**: Ruta local donde está instalada la SDK de Android.
- **jvm4testexecution**: Ruta local donde está instalada la JDK de Java.
- **javacompliancelevel**: Versión de JDK a ejecutar, siendo 8 el máximo valor permitido.
- **port**: Puerto al que se conectará AstorWorker.

El comando para ejecutarlo es:

```
mvn exec:java
-Dexec.mainClass=br.ufg.inf.astor4android.main.evolution.Astor4AndroidMain
-Dexec.args="<arguments>"
```

donde $< arguments >$ corresponde a los parámetros previamente listados.

5.6.3. Modificaciones realizadas a Astor4Android

Ejecutamos la herramienta para todos los $commit_{bug}$ de nuestro benchmark, con la intención de repararlos. En todos encontramos inconvenientes, a pesar de ir variando los parámetros. Investigando la herramienta, dedujimos la siguiente información:

Astor4Android se desarrolló durante los años 2017 y 2018, como se evidencia en el historial de commits de su repositorio Git[24]. En junio de 2017, la versión más reciente del plugin de Gradle para Android era 2.3.3[25], lo que nos llevó a suponer que las aplicaciones evaluadas con esta herramienta utilizaran dicha versión o alguna anterior. Realizamos

una revisión de todos los bugs documentados en Droidbugs[26], corroborando nuestra suposición: la máxima versión utilizada de AGP es 2.3.3.

Analizando el código de Astor4Android[27], observamos que busca las clases compiladas de la aplicación a reparar en un directorio que se discontinuó en la versión 3.0 de AGP. Por lo tanto, concluimos que no soportaba reparar aplicaciones que utilizaran versiones superiores a 2.3.3 de AGP.

Astor4Android no podía compilar ninguna de las aplicaciones ya que todas utilizan versiones de AGP no soportadas por la herramienta. Por lo tanto, hicimos las siguientes modificaciones:

- Astor4Android utiliza Maven para la gestión de dependencias, las mismas están configuradas en un archivo llamado *pom.xml*[6]. Modificamos la versión de *spoon-core*, en lugar de utilizar 5.4.0-SNAPSHOT, usar 5.5.0. De otra manera, el proyecto no compila.

- Adaptamos el código de Astor4Android para buscar las clases compiladas de la aplicación a reparar en otros directorios. Originalmente, se utilizaba la ruta /build/intermediates/classes/debug, pero con aplicaciones más modernas es necesario utilizar build/intermediates/javac/debug/classes.

- El equivalente al punto anterior, pero para la clase R[7], automáticamente generada.

- Cambiamos la forma de obtención de dependencias externas: Astor4android no utiliza un gestor de dependencias específico para la aplicación que se está reparando, por lo que es necesario contar con todas las dependencias externas descargadas y agregadas al *classpath* que utiliza la herramienta para su ejecución. En la versión original de Astor4Android, esto se hacía con una tarea de Gradle, que generaba un directorio denominado *localrepo*, conteniendo todas las dependencias. Esta tarea dejó de funcionar para AGP 3, por modificaciones en el sistema de permisos de Gradle.

  Inicialmente intentamos reescribir la tarea con el fin de identificar todas las versiones de las dependencias externas, tanto directa como transitivamente. Sin embargo, no pudimos realizarlo debido a las limitaciones de permisos mencionadas previamente. Por lo tanto, decidimos avanzar por una solución que no se viera limitada por los cambios de permisos.

  Notamos que con el objetivo de mejorar la performance de compilación Gradle guarda todos los archivos .jar y .aar de las dependencias en un directorio denominado *caches*. Por lo que, si compilamos con *./gradlew build* una aplicación, sabemos que todas sus dependencias estarán en ese directorio.

  Luego, basta con ejecutar *./gradlew build* en la aplicación a reparar, para descargar todas las dependencias necesarias y luego copiarlas al directorio *localrepo*. El directorio *caches* es compartido por todas las aplicaciones que se compilan en la misma computadora. Para aislar y obtener las dependencias de cada aplicación, eliminamos este directorio antes de buscar las dependencias para una aplicación en particular. Con esto nos aseguramos que luego de compilarla, todos los elementos en el directorio corresponderán a dependencias de la aplicación.

[6] https://maven.apache.org/pom.html
[7] https://developer.android.com/reference/android/R

Resumiendo, los pasos a seguir para conseguir los archivos necesarios de las dependencias directas y transitivas:

1. De existir, eliminar el directorio *caches* dentro de *.gradle*.
2. Compilar la aplicación ejecutando *./gradlew build*
3. Copiar recursivamente todos los archivos *.aar* y *.jar* dentro del directorio *caches* a un directorio especifico para esta aplicación, llamado *localrepo*.
4. Agregar ese directorio al *classpath*, para que Astor4Android pueda compilar la aplicación correctamente.

Luego de estos cambios logramos que Astor4Android compilara 2 commits de manera exitosa. Sin embargo, el proceso de localización de fallas no funcionó, por lo que procedimos a hacer modificaciones sobre la herramienta:

- Para este proceso es necesario ejecutar los tests. En Android, estos se ejecutan utilizando tareas de Gradle. El nombre de estas tareas puede cambiar por la versión de AGP, o por cuestiones particulares de la aplicación, como los *flavors*[8] y los *buildtypes*. Astor4Android necesita determinar, del conjunto de tareas de la aplicación, cuál usar para ejecutar los tests. Para esto utiliza expresiones regulares que realizan un filtro sobre el conjunto de tareas disponibles. Las expresiones regulares estaban acopladas a la versión 2 de AGP. Hicimos modificaciones para que funcionen con AGP superior a 3.0.

- Astor4Android ejecuta tareas de Gradle para determinar qué líneas son cubiertas por los tests utilizando JaCoCo. Para poder reparar aplicaciones que utilizan la versión de Gradle mayor a 5.1.1, realizamos cambios como el uso de *setFrom* para definir el valor de ciertas propiedades. Esto se debe a que algunas funcionalidades son requeridas a partir de Gradle 5.X[28]. Notar que a partir de AGP 3.4.0, es obligatorio el uso de gradle 5.1.1 o superior[29].

Luego de estas modificaciones, logramos ejecutar Astor4Android exitosamente para los 2 commits iniciales, utilizando el algoritmo de reparación **JGenProg2**. Para lograr ejecutar los algoritmos de reparación **JMutRepair** y **JKali**, realizamos además las siguientes modificaciones:

- El comando de compilación inicialmente utilizado por la herramienta excluía las clases de tests, lo que generaba un error durante la ejecución de estas dos técnicas. El sistema informaba que no podía encontrar las clases de tests necesarias para su ejecución. Resolvimos este problema modificando el comando de compilación para que incluyera dichas clases.

- A su vez, requerían tener en el classpath dónde estaban las clases compiladas de test. Tuvimos que modificar el código para agregar estas directorios.

- Leyendo el código fuente detectamos que la herramienta tenía dos flujos de ejecución, uno **JGenProg2**, y otro para **JMutRepair** y **JKali**. En el flujo de ejecución de las dos últimas, los parámetros utilizados para ejecutarlas no eran correctos. Particularmente, cómo obtener el test fallido y su clase asociada. Modificamos el código para la correcta propagación de los parámetros.

[8] https://developer.android.com/studio/build/build-variants?hl=es-419#product-flavors

Con las modificaciones que realizamos a la herramienta resolvimos varios problemas. Sin embargo, no pudimos resolver todos, ya que requerían modificar las aplicaciones. Por ejemplo, no poder ejecutar la herramienta porque la aplicación usaba AGP mayor o igual a 7.0, el cual requiere JDK 11 o superior.

La siguiente tabla expresa el avance alcanzado hasta ese momento:

Estado	Cantidad
No poder ejecutar, AGP $\geq$ 7.0	5
Error al compilar	14
Error al localizar fallas	0
Reparación de programas ejecutada	2
Total	21

5.6.4. Modificaciones realizadas a las aplicaciones

Luego de haber agotado las modificaciones posibles en el código de Astor4Android, y motivados por algunos mensajes de error más específicos, procedimos a realizar modificaciones en las aplicaciones.

- Comenzamos por los 5 commits que utilizaban AGP mayor o igual a 7.0. Modificamos las aplicaciones para que usen la versión inmediatamente inferior: 4.2.0 [30]. Solo en un caso la aplicación siguió funcionando a pesar del cambio, en los otros 4 casos surgieron errores relacionados a sus dependencias. Alguna dependencia directa o transitiva requería utilizar JDK 11 o superior, y bajar la versión implicaba grandes cambios en el código de la aplicación. No logramos arreglar las dependencias de los siguientes commits:
  - ***Slapperwan-Miscelánea-1***.
  - ***Nextcloud_deck-NPE-1***.
  - ***Gsantner_markor-Miscelánea-2***.
  - ***CatimaLoyalty-Miscelánea-3***.
- Había 3 commits que no respetaban la estructura estándar de directorios de Android:
  - ***Niccokunzmann-Operator-1***.
  - ***Niccokunzmann-Operator-2***.
  - ***Gsantner_markor-Miscelánea-1***.

  No cumplían con el requisito de tener un directorio llamado *app*, y tenían un solo archivo *build.gradle*. Hicimos los siguientes cambios:

  1. Creamos el directorio *app*, y movimos *src* dentro del mismo.
  2. Separamos el archivo *build.gradle* en dos, uno que contiene las directivas de compilación de la aplicación, el otro las del módulo. Los guardamos en los directorios correspondientes según el estándar.
  3. En *settings.gradle* incluimos la siguiente linea: "**include ':app'** ", indicándole a Gradle que incluya el módulo app a la hora de compilar.

Con estos cambios logramos que estos 3 commits pasen la fase de compilación de Astor4Android. Al ejecutar el proceso la localización de fallas, falló solamente para ***Gsantner_markor-Miscelánea-1***, por no utilizar AndroidX.

- Muchas aplicaciones que usaban bibliotecas de compatibilidad[9] no compilaban al usar Astor4Android. Hicimos dos tipos de cambios:
  - Migrar a AndroidX[10]. Para esto utilizamos la funcionalidad de migración[11] que ofrece Android Studio.
  - Actualizamos cada dependencia que siguiera usando bibiliotecas de compatibilidad, a su versión mínima que usara AndroidX. En algunos casos funcionó y en otros no.

  Luego de estos cambios logramos hacer que Astor4Android compile 3 aplicaciones más:
  - ***CatimaLoyalty-Miscelánea-1***.
  - ***CatimaLoyalty-Miscelánea-2***.
  - ***Gsantner_markor-Miscelánea-1***.
- Commits donde se usaba una versión de AGP superior a 3.5: en estos casos, agregamos *android.generateRJava=true* a las propiedades de Gradle, ya que en la versión 3.6 de AGP se cambió la forma en la que se genera y guarda la clase R. Con esa configuración, forzamos a que se mantenga el comportamiento anterior.

  Para los casos donde la versión de AGP era superior a 4.0 e inferior a 7.0, primero tuvimos que cambiar la versión de AGP a 4.0, y luego agregar esa configuración, debido a que se deprecó en la versión 4.1.
- Commits con versiones de AGP menores a 3.4.2: En estos escenarios, actualizamos la versión de AGP del proyecto a 3.4.2, que es el número de versión más bajo en el que conseguimos que la localización de fallas operara adecuadamente.
- Notamos que ciertas versiones de AGP y de Gradle, eran incompatibles con la versión de JaCoCo que usaba Astor4Android. En algunos casos tuvimos que cambiar tanto las versiones de AGP como la de JaCoco.
- Si la aplicación tenía mas de un flavor[12], borramos todos menos uno. Aplicaciones con más de un flavor traen problemas a la hora de elegir la tarea para correr los tests. Esto se debe a que se crean tantas tareas como combinaciones de flavors y variantes de compilación hayan.

En la siguiente tabla, se ve el estado alcanzado luego de las modificaciones a las aplicaciones:

[9] https://developer.android.com/topic/libraries/support-library/packages?hl=es-419
[10] https://developer.android.com/jetpack/androidx?hl=es-419
[11] https://developer.android.com/jetpack/androidx/migrate?hl=es-419
[12] https://developer.android.com/studio/build/build-variants?hl=es-419#product-flavors

Estado	Cantidad
Commits que utilizan JDK 11	4
Error al compilar	9
Error al localizar fallas	7
Reparación de programas ejecutada	1
Total	21

5.6.5. Errores específicos

Para esta instancia, ya no encontramos cambios que pudiéramos hacer de manera general sobre el conjunto de datos, por lo que nos enfocamos en intentar resolver errores específicos. Listamos los cambios realizados a continuación:

- Forzar a que todas las dependencias transitivas usen una versión en particular. Encontramos casos donde, al resolver todas las dependencias, había distintas versiones para la misma dependencia dentro del *classpath*. A veces, generaba incompatibilidades, por lo que especificamos en *build.gradle* que versión necesitábamos para todas las dependencias transitivas.

- Borramos tareas de Gradle que ejecutaban a la hora de correr *./gradlew build*, y que no eran realmente necesarias, como linters, o herramientas de análisis de código estático.

- Incompatibilidades entre compiladores: Como mencionamos en la sección **5.5**, Astor4Android usa un método de compilación distinto al que se usa al compilar con *./gradlew build*. Por ejemplo, para Astor4Android, Integer e int son tipos incompatibles. Lo resolvimos cambiando el código fuente de la aplicación para las líneas conflictivas, y haciendo que siempre se use el mismo tipo.

- Notamos que Astor4Android asumía que había una sola clase definida por archivo .java, y no funcionaba en caso contrario. En un commit en particular, fallaba la compilación por no encontrar una clase que se encontraba definida junto con otra en un mismo archivo. Las tuvimos que separar para que funcione.

Luego de todos los cambios, el resultado fue:

Estado	Cantidad
Commits que utilizan JDK 11	4
Error al compilar	4
Error al localizar fallas	4
Reparación de programas ejecutada	9
Total	21

Intentamos realizar otras modificaciones para los 8 commits que seguían en estado *error al compilar* o *error al localizar fallas*. Luego de haber dedicado un promedio de 10 horas a cada uno sin poder registrar ningún avance, decidimos descartarlos.

Luego de analizar los resultados preliminares de la ejecución de Astor4Android sobre el benchmark, notamos que las técnicas de localización de fallas no generaban líneas sospechosas para el bug **Woheller_weather-Operator-1**. Identificamos que esto se debe

a que el test fallido utiliza *Powermock*, librería que tiene problemas de compatibilidad con JaCoCo[31]. Por lo tanto, al no poder obtener información de cobertura, no se puede calcular S(s) para ninguna de las 5 técnicas.

En la siguiente sección, analizaremos los resultados de cada técnica aplicada sobre estos 8 bugs:

- ***Niccokunzmann-Operator-1***.
- ***Niccokunzmann-Operator-2***.
- ***Gsantner_markor-Miscelánea-1***.
- ***CatimaLoyalty-Miscelánea-1***.
- ***CatimaLoyalty-Miscelánea-2***.
- ***CatimaLoyalty-Miscelánea-4***.
- ***Stypox_dicio_android-Regex-1***.
- ***Tutao_tutanota-Miscelánea-1***.

6. EVALUACIÓN DE LAS TÉCNICAS

6.1. Introducción

En esta sección buscamos evaluar como funcionan las distintas técnicas de localización de fallas y reparación de programas usando los 8 $commit_{bug}$ que pudimos ejecutar sin errores.

En general, notamos un buen funcionamiento de las técnicas de localización de fallas, no así de las de reparación de programas.

En todos los casos alguna de las técnicas de localización de fallas pudo obtener un conjunto de líneas sospechosas. En un solo caso no se detectó como línea sospechosa la defectuosa. La herramienta propuso correcciones para 4 bugs, de los cuales 2 eran efectivamente una solución correcta. Para los otros 4, no propuso soluciones.

6.2. Metodología

Ejecutamos Astor4Android para cada $commit_{bug}$ con todas las posibles combinaciones de los algoritmos de localización de fallas y de reparación de programas. Variamos el parámetro **flthreshold** para cada una de las distintas combinaciones. Esto quiere decir que ejecutamos la herramienta con estos valores, para los distintos parámetros explicados en la sección **5.6.2**:

- **mode**: statement, statement-remove y mutation.
- **flmode**: ochiai, op2, dstar, barinel y tarantula.
- **flthreshold**: Este parámetro varía en un rango de 0 a 1. En el modo "statement", optamos por valores de 0.2, 0.4, 0.6 y 0.8 para evaluar cómo el tamaño del conjunto de líneas sospechosas afecta el rendimiento del algoritmo. Esto se justifica ya que, independientemente del tamaño del conjunto, se generan 500 variantes. Un umbral más elevado conlleva un conjunto más pequeño pero con elementos de mayor sospecha.

  Para los otros dos modos, establecimos un umbral único de 0.2. La razón es que estas técnicas exploran exhaustivamente el espacio de búsqueda, aplicando todas las modificaciones posibles a cada línea sospechosa del conjunto. Optamos por mantener un conjunto de líneas sospechosas lo más amplio posible, especialmente dado que estas técnicas son más eficientes en términos de tiempo de ejecución.
- **stopfirst**: *False*. Buscamos que la herramienta intentara encontrar la mayor cantidad de soluciones posibles. Notamos que muchas soluciones son incorrectas, a pesar de cumplir la condición de pasar todos los tests. Es por esto que nos pareció mejor generar todas las soluciones posibles para luego validar manualmente si son efectivamente correctas.

La ejecución del programa para los 8 commits nos llevó aproximadamente 200hs, realizada en computadoras MacBook Pro 2020, con 16GB de memoria RAM y procesador Intel Core i5 2 GHz Quad-Core. Esto se debe mayormente a que el proceso de localización de fallas era lo que más tardaba: En promedio, 25 minutos. Además, *JGenProg2* es la que

más tiempo consumía y la más ejecutada, dado que se ejecuta para 4 valores distintos de *flthreshold* por commit.

6.3. Modificaciones realizadas a las aplicaciones

Notamos que algunas aplicaciones tenían tests que fallaban en numerosos commits, incluyendo $commit_{bug}$ y $commit_{fix}$. Eliminamos estos tests antes de realizar las ejecuciones, ya que consideramos que introducen imprecisiones tanto en el proceso de localización de fallas como en el de reparación de programas.

- Localización de fallas: como todas las técnicas incluyen en sus cálculos los tests fallidos, estos tests influyen directamente en el resultado de este proceso.
- Reparación de programas: Los tests fallidos nada tenían que ver con el código defectuoso a reparar. Por lo que, de seguir fallando, harían que las técnicas de reparación de programas no puedan identificar ninguna variante como válida, ya que lo hacen cuando todos los tests pasan.

6.4. Resultados

Podemos categorizar los resultados de la siguiente manera:

No se encuentran líneas sospechosas	NS
No se proponen soluciones candidatas	NC
Se proponen soluciones incorrectas	SI
Se proponen soluciones correctas	SC

En las secciones que siguen, presentamos los resultados detallados para cada commit analizado. Fijamos el valor del parámetro **flthreshold** en 0.2 para las técnicas **JKali** y **JMutRepair**. Para **JGenProg2**, encontramos que los valores de 0.2 y 0.4 ofrecían resultados equivalentes. Al elevar el valor a 0.6 y 0.8, la eficacia disminuyó debido a la eliminación de un número significativo de líneas sospechosas. Por lo tanto, nos centramos en los resultados obtenidos con el valor 0.4 del parámetro. Podemos observar en el siguiente gráfico los valores de la función **S(s)** para las distintas técnicas de localización de fallas sobre el benchmark:

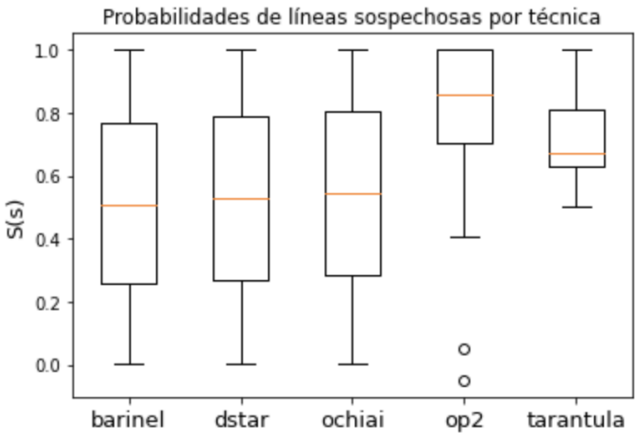

Fig. 6.1: Las probabilidades asignadas a las líneas sospechosas para cada método de localización de fallas. Elegimos el umbral 0.4 para visualizar los resultados en la siguiente sección.

6.4.1. Niccokunzmann-Operator-1

El siguiente bug no pudo ser reparado por Astor4Android:

```
@@ -31,7 +31,7 @@ public Result compute() {
31 31                    if (x > 0 && classified[x-1 + y * width] == thisValue) {
32 32                        if (y > 0 && classified[x + (y-1) * width] == thisValue) {
33 33                            /* pixel is equal to left and top */
34    -                          int leftLabel = area[x + y * width];
   34 +                          int leftLabel = area[x-1 + y * width];
35 35                            int topLabel = area[x + (y-1) * width];
36 36                            area[x + y * width] = leftLabel;
37 37                            if (leftLabel == topLabel) {
38 38                                /* both labels are equivalent */
39 39                            } else {
40 40                                /* record equivalence */
41 41                                Set<Integer> leftSet = labels.get(leftLabel);
42 42                                leftSet.addAll(labels.get(topLabel));
43 43                                labels.set(topLabel, leftSet);
44 44                            }
45 45                        } else {
46 46                            /* pixel is equal to left only */
47 47                            area[x + y * width] = area[x-1 + y * width];
48 48                        }
```

Podemos observar en la siguiente tabla como funcionaron las distintas técnicas de reparación de programas y localización de fallas:

-	Ochiai	Op2	DStar	Barinel	Tarantula
JGenProg2	NC	NC	NC	NC	NC
jMutRepair	NC	NC	NC	NC	NC
jKali	NC	NC	NC	NC	NC

Ninguna técnica propuso solución. Esperábamos que **JGenProg2** pudiera reparar este

bug, ya que en la línea 31 se puede ver código similar al necesario para hacerlo. Al observar el resultado del proceso de localización de fallas, observamos que las principales líneas sospechosas son las líneas 41, 42 y 43.

Por limitaciones teóricas mencionadas en la sección **5.4**, **JMutRepair** y **JKali** no podrían haberlo solucionado:

- **JMutRepair**: aplica operadores de mutación en las guardas sospechosas de estructuras *if*. En este caso, el bug no podría haberse reparado haciendo cambios sobre las condiciones de los *if*.
- **JKali**: borra líneas de código, cambia las condiciones de *if* por *true* o *false*, o inserta sentencias *return* con valores por defecto. Tampoco son estrategias útiles para este caso.

6.4.2. Niccokunzmann-Operator-2

El siguiente bug no pudo ser reparado por Astor4Android:

```
130  130      public void setPaintColor(int color)
131  131      {
132  132          paintColor = color;
133  133          if (paintColor == FloodFill.BORDER_COLOR) {
134       -          paintColor ++;
     134  +          paintColor = paintColor ^ 1;
135  135          }
136  136      }
137  137
```

Podemos observar en la siguiente tabla como funcionaron las distintas técnicas de reparación de programas y localización de fallas:

-	Ochiai	Op2	DStar	Barinel	Tarantula
JGenProg2	SI	SI	SI	SI	SI
jMutRepair	SI	SI	SI	SI	SI
jKali	SI	SI	SI	SI	SI

Todas los métodos de localización de fallas propusieron el mismo conjunto de líneas sospechosas. Estas son las que están en el extracto de código exhibido.

El funcionamiento de cada algoritmo de reparación de programas fue el siguiente:

JKali:

Para las 5 técnicas de localización de fallas propuso tres soluciones distintas.

1.

```
public void setPaintColor(int color) {
        paintColor = color;
-       if(paintColor == org.androidsoft.coloring.util.FloodFill.BORDER_COLOR){
+       if(false) {
                paintColor++;
        }
 }
```

2.

```
public void setPaintColor(int color) {
        paintColor = color;
-       if(paintColor == org.androidsoft.coloring.util.FloodFill.BORDER_COLOR){
-           paintColor++;
-       }
 }
```

3.

```
public void setPaintColor(int color) {
        paintColor = color;
        if(paintColor == org.androidsoft.coloring.util.FloodFill.BORDER_COLOR) {
-           paintColor++;
        }
 }
```

Podemos ver que las 3 son equivalentes, lo que hacen es no incrementar el valor de la variable *paintColor*. Ninguna de las tres es correcta porque lo que se buscaba originalmente era evitar que la variable *paintColor* fuera igual a BORDER_COLOR, y las variantes propuestas contradicen eso.

JMutRepair:

Este algoritmo propuso 4 soluciones distintas para las 5 técnicas de localización de fallas:

```
public void setPaintColor(int color) {
        paintColor = color;
-       if(paintColor == org.androidsoft.coloring.util.FloodFill.BORDER_COLOR) {
+       if(paintColor != org.androidsoft.coloring.util.FloodFill.BORDER_COLOR) {
            paintColor++;
        }
 }
```

Las otras tres son análogas, pero en lugar de utilizar el operador $!=$, se reemplaza por $>$, $<$, y la última niega toda la guarda del if. Tampoco son soluciones correctas porque ocasionan que la variable paintColor se modifique para casos no deseados. Por ejemplo, en lugar de hacerlo solo para un valor, lo hace para todos los valores menores al de la condición del if.

JGenProg2:

Propuso las mismas 3 soluciones que **JKali**. Ninguna de las soluciones propuestas repara correctamente la aplicación. Esto se debe a dos razones:

- El conjunto de tests resulta incompleto: No cubren todos los casos de uso posibles para el método *setPaintColor*, por ejemplo, el caso donde no es cierta la condición del *if*. Esto hace que variantes generadas por las técnicas parezcan correctas, aunque no lo sean.

- La única técnica que podría haber generado una variante que solucionara el problema es **JGenProg2**. Podemos notar que la solución original no es posible de generar con esta técnica porque el operador "^" no se utiliza en ninguna otra parte del código de la aplicación.

Escribimos los tests faltantes para que quedara especificado todo el comportamiento esperado de la función, y volvimos a ejecutar la herramienta. Estos son los tests que agregamos:

```
@Test
public void testSetPaintColorNormalValue() {
    setUp();

    int color = 123456;
    paintArea.setPaintColor(color);
    assertEquals(color, paintArea.getPaintColor());
}

@Test
public void testSetPaintColorEqualToBorderColor() {
    setUp();

    int originalBorderColor = FloodFill.BORDER_COLOR;
    paintArea.setPaintColor(originalBorderColor);
    assertNotEquals(originalBorderColor, paintArea.getPaintColor());
    assertEquals(originalBorderColor + 1, paintArea.getPaintColor());
}
```

El impacto en los resultados fue significativo: ninguna de las técnicas fue capaz de proponer una solución. En nuestra opinión, es preferible que la herramienta no ofrezca una solución a que ofrezca una incorrecta. Este caso resalta la importancia de tener un conjunto de tests bien definido para establecer el comportamiento esperado de la aplicación.

6.4.3. Gsantner_markor-Miscelánea-1

El siguiente bug fue reparado por Astor4Android:

```
145 145     public static int getIndexFromLineOffset(final CharSequence s, final int l, final int e) {
146 146         int i = 0, count = 0;
147 147         if (s != null) {
148 148             if (l > 0) {
149 149                 for (; i < s.length(); i++) {
150 150                     if (s.charAt(i) == '\n') {
151 151                         count++;
152 152                     }
153 153                     if (count == l) {
154 154                         break;
155 155                     }
156 156                 }
157 157             }
158     -           if (i < s.length() - 1) {
    158 +           if (i < s.length()) {
159 159                 final int start = (l == 0) ? 0 : i + 1;
160 160                 final int end = getLineEnd(s, start);
161 161                 // Prevent selection from moving to previous line
162 162                 return end - Math.min(e, end - start);
163 163             }
164 164         }
165 165         return i;
166 166     }
```

Podemos observar en la siguiente tabla como funcionaron las distintas técnicas de reparación de programas y localización de fallas:

-	Ochiai	Op2	DStar	Barinel	Tarantula
JGenProg2	NC	NC	NC	NC	NC
jMutRepair	SC	SC	SC	NC	SC
jKali	SI	SI	SI	SI	SI

Todas las técnicas de localización de fallas señalaron múltiples líneas como sospechosas y lograron incluir acertadamente la línea que contiene el bug.

La función calcula el índice de un caracter dentro de una secuencia de caracteres basado en un número de línea especificado y un desplazamiento desde el final de esa línea. Se recorre la secuencia de caracteres contando los caracteres de nueva línea hasta llegar a la línea deseada y, luego, calcula y devuelve el índice basado en el desplazamiento especificado.

En una primera instancia vimos que todas las soluciones propuestas eran incorrectas. Este es uno de los bugs para los cuales sintetizamos un test que evidenciara el caso borde que falla cuando $i == s.length() - 1$:

```
@Test
public void testGetIndexFromLineOffset_LastCharacter() {
    CharSequence s = "Hello\nWorld\n";
    int index = StringUtils.getIndexFromLineOffset(s, 2, 0);
    assertEquals(12, index);
}
```

Notamos que las soluciones propuestas no mantenían gran parte del comportamiento usual de la función. Deducimos que esto se debe a que el test que agregamos no contemplaba todos los casos posibles. Por eso procedimos a agregar tests para más casos:

```
@Test
public void testGetIndexFromLineOffset_SecondLine() {
    CharSequence s = "Hello\nWorld\nUniverse\nGalaxy\n";
    int index = StringUtils.getIndexFromLineOffset(s, 1, 0);
    assertEquals(11, index);
}
@Test
public void testGetIndexFromLineOffset_FirstLine() {
    CharSequence s = "Hello\nWorld\n";
    int index = StringUtils.getIndexFromLineOffset(s, 0, 0);
    assertEquals(5, index);
}
@Test
public void testGetIndexFromLineOffset_OneLine() {
    CharSequence s = "Hello\n";
    int index = StringUtils.getIndexFromLineOffset(s, 1, 0);
    assertEquals(5, index);
}
```

Los resultados de la ejecución de cada algoritmo de reparación de programas fueron:

JGenProg2:

El algoritmo que inicialmente consideramos como el más prometedor para corregir este defecto no cumplió con las expectativas. En la primera ejecución, en la que solo disponíamos del primer test, generó más de 60 variantes para cada técnica de localización de

errores. Dado que el conjunto de tests era incompleto, estas variantes introducían cambios que resultaban en nuevos bugs dentro de la funcionalidad general del programa. Por ejemplo:

```
 if(l > 0) {
    for(; i < (s.length()); i++) {
        if (s.charAt(i) == '\n') {
+           if (l > 0) {
+               for (; i < (s.length()); i++) {
+                   if ((s.charAt(i)) == '\n') {
+                       count++;
+                   }
+                   if (count == l) {
+                       break;
+                   }
+               }
+           }
        count++;
        }
        if(count == l) {
            break;
        }
    }
 }
```

Esta variante duplica la primer parte del código dentro de la misma guarda del *if*. Esto hace que no se pueda cumplir la segunda guarda $if(count == l)$ y que se termine el ciclo *for* con $i == s.length()$. Por ende, no se valida el comportamiento dentro de la guarda del if, no cubierto por el test inicial que escribimos.

Luego de agregar los tests correspondientes al comportamiento general de la función, la técnica no pudo proponer arreglo para el bug.

JKali:

Para las 5 técnicas de localización de fallas propuso las mismas 7 soluciones:

1.

```
if(l > 0) {
    for(; i < (s.length()); i++) {
        if (s.charAt(i) == '\n') {
            count++;
        }
        if(count == l) {
-           break;
        }
    }
}
```

2.

```
for (; i < (s.length()); i++) {
    if (s.charAt(i) == '\n') {
        count++;
    }
-   if (count == l) {
-       break;
```

```
-   }
}
```

3.

```
if(l > 0) {
    for(; i < (s.length()); i++) {
        if (s.charAt(i) == '\n') {
            count++;
        }
-       if(count == 1) {
+       if(false) {
            break;
        }
    }
}
```

4.

```
if(l > 0) {
    for(; i < (s.length()); i++) {
        if (s.charAt(i) == '\n') {
-           count++;
        }
        if(count == 1) {
            break;
        }
    }
}
```

5.

```
if(l > 0) {
    for(; i < (s.length()); i++) {
-       if(s.charAt(i) == '\n'){
+       if(false){
            count++;
        }
        if(count == 1) {
            break;
        }
    }
}
```

6.

```
-   if(i < (s.length() - 1)){
+   if(true) {
        final int start = (l == 0) ? 0 : i + 1;
        final int end = net.gsantner.opoc.util.StringUtils.getLineEnd(s,start);
        return end - java.lang.Math.min(e, (end - start));
    }
```

7.

```
if (l > 0) {
    for (; i < (s.length()); i++) {
```

```
-       if ((s.charAt(i)) == '\n') {
-           count++;
-       }
        if (count == 1) {
            break;
        }
    }
}
```

Podemos observar que **1**, **2** y **3** son equivalentes, así como también **4** y **5**. Ninguna de estas soluciones es correcta por quitar funcionalidad a la función. Al agregar los tests que evalúan más comportamiento solo genero la variante **6**.

JMutRepair:

En una primer instancia, para las 5 técnicas de localización de fallas, propuso las mismas 6 soluciones.

1.

```
for(; i < (s.length()); i++) {
    if (s.charAt(i) == '\n') {
        count++;
    }
-   if (count == 1) {
+   if (count > 1) {
        break;
    }
}
```

2.

```
for(; i < (s.length()); i++) {
-   if (s.charAt(i) == '\n') {
+   if (s.charAt(i) < '\n') {
        count++;
    }
    if (count == 1) {
    if (count > 1) {
        break;
    }
}
```

3.

```
-       if(i < (s.length() - 1)){
+       if(i == (s.length() - 1)) {
        final int start = (l == 0) ? 0 : i + 1;
        final int end = net.gsantner.opoc.util.StringUtils
  .getLineEnd(s, start);
          return end - java.lang.Math.min(e, (end - start));
    }
```

4.

```
-       if(i < (s.length() - 1)){
```

```
+       if(i <= (s.length() - 1)) {
        final int start = (l == 0) ? 0 : i + 1;
        final int end = net.gsantner.opoc.util.StringUtils.getLineEnd(s, start);
          return end - java.lang.Math.min(e, (end - start));
    }
```

5.

```
-       if(i < (s.length() - 1)){
+       if(i >= (s.length() - 1)) {
        final int start = (l == 0) ? 0 : i + 1;
        final int end = net.gsantner.opoc.util.StringUtils.getLineEnd(s, start);
          return end - java.lang.Math.min(e, (end - start));
    }
```

La única solución correcta y análoga al fix real es la solución número 4. Una vez agregado el resto de los tests esta fue la única solución propuesta por esta técnica.

6.4.4. CatimaLoyalty-Miscelánea-1

El siguiente bug no pudo ser reparado por Astor4Android:

```
231
232         if(cardIdFieldView.getText().length() > 0 && barcodeTypeField.getText().length() > 0)
233         {
        -       if(barcodeTypeField.getText().equals(NO_BARCODE))
234     +       if(barcodeTypeField.getText().toString().equals(NO_BARCODE))
235             {
236                 hideBarcode();
237             }
```

Podemos observar en la siguiente tabla como funcionaron las distintas técnicas de reparación de programas y localización de fallas:

-	Ochiai	Op2	DStar	Barinel	Tarantula
JGenProg2	NC	NC	NC	NC	NC
jMutRepair	NC	NC	NC	NC	NC
jKali	NC	NC	NC	NC	NC

Los métodos de localización de fallas propusieron varias líneas cercanas a dónde ocurre el fix, pero ninguna seleccionó como sospechosa la línea 234 que es la defectuosa. Una de las posibilidades por la que fallaron es que la función que contiene el bug tiene una longitud de casi 200 líneas de código. Esto podría dificultar el buen funcionamiento de las técnicas de localización de fallas.

Ninguno de los 3 algoritmos de reparación de programas pudo encontrar una solución. Era esperable que ni **JKali** ni **JMutRepair** lo hicieran porque no está dentro de sus posibilidades semejante mutación de código. **JGenProg2** es el candidato a poder solucionar este tipo de bugs, y hay numerosas líneas en el archivo a reparar que contienen el llamado a *toString* luego de *getText*. De todas maneras, el problema parece haber estado en la etapa de localización de fallas por no haber detectado correctamente la línea defectuosa.

6.4.5. CatimaLoyalty-Miscelánea-2

El siguiente bug no pudo ser reparado por Astor4Android:

```
@@ -208,6 +208,7 @@ else if(importLoyaltyCardUri != null)
208          else
209          {
210              setTitle(R.string.addCardTitle);
211 +            hideBarcode();
212          }
213
214          if(headingColorValue == null)
```

Podemos observar en la siguiente tabla como funcionaron las distintas técnicas de reparación de programas y localización de fallas:

-	Ochiai	Op2	DStar	Barinel	Tarantula
JGenProg2	NC	NC	NS	NS	NC
jMutRepair	NC	NC	NS	NS	NC
jKali	NC	NC	NS	NS	NC

Este caso es distinto a la mayoría porque, en vez de haber una línea defectuosa, faltaba realizar el llamado a una función. Algunas técnicas de localización de fallas no pudieron encontrar líneas sospechosas. Las que sí lo hicieron detectaron correctamente el entorno dónde se encontraba el problema, pero ninguna técnica de reparación de programas logró solucionarlo. **JKali** y **JMutRepair** no podrían haberlo reparado por limitaciones teóricas. **JGenProg2** era el candidato a solucionarlo, pero tampoco lo logró.

6.4.6. CatimaLoyalty-Miscelánea-4

El siguiente bug fue reparado por Astor4Android con algunos de los métodos de localización de fallas.

```
@@ -68,7 +68,7 @@ public void bindView(View view, Context context, Cursor cursor)
68              expiryField.setVisibility(View.VISIBLE);
69              int expiryString = R.string.expiryStateSentence;
70              if(Utils.hasExpired(loyaltyCard.expiry)) {
-                   expiryString = R.string.expiryStateSentence;
71 +                 expiryString = R.string.expiryStateSentenceExpired;
72                  expiryField.setTextColor(context.getResources().getColor(R.color.alert));
73              }
74              expiryField.setText(context.getString(expiryString, DateFormat.getDateInstance(DateFormat.LONG).format(loyaltyCard.expiry)));
```

Podemos observar en la siguiente tabla como funcionaron las distintas técnicas de reparación de programas y localización de fallas:

-	Ochiai	Op2	DStar	Barinel	Tarantula
JGenProg2	SC	NC	NS	SC	SC
jMutRepair	NC	NC	NS	NC	NC
jKali	NC	NC	NS	NC	NC

Op2 encontró una línea sospechosa distinta al resto de los métodos y *DStar* no encontró ninguna. Por esas razones no lograron solucionar el bug. La solución propuesta por

JGenProg2 es exactamente la misma que la solución original. No era posible que **JKali** y **JMutRepair** pudieran solucionar el bug.

6.4.7. Stypox_dicio_android-Regex-1

El siguiente bug no pudo ser reparado por Astor4Android:

```
2 app/src/main/java/org/dicio/dicio_android/util/StringUtils.java
@@ -9,7 +9,7 @@
 9   9
10  10  public class StringUtils {
11  11      private static final Pattern punctuationPattern = Pattern.compile("\\p{Punct}");
12      -   private static final Pattern wordDelimitersPattern = Pattern.compile("[^\\p{L}]");
    12  +   private static final Pattern wordDelimitersPattern = Pattern.compile("[^\\p{L}0-9]");
13  13
14  14      private StringUtils() {
15  15      }
```

Podemos observar en la siguiente tabla como funcionaron las distintas técnicas de reparación de programas y localización de fallas:

-	Ochiai	Op2	DStar	Barinel	Tarantula
JGenProg2	NC	NC	NS	NS	NC
jMutRepair	NC	NC	NS	NS	NC
jKali	NC	NC	NS	NS	NC

Dstar y *Barinel* no encontraron líneas sospechosas. El resto de los métodos de localización de fallas lograron identificar a la línea 12 como sospechosa. De todas maneras, la solución estaba fuera del alcance de **Jkali** y **JMutRepair**. En este caso tampoco era posible que **JGenProg2** pudiera reparar el programa porque en ningún lugar hay una solución similar.

6.4.8. Tutao_tutanota-Miscelánea-1

El siguiente bug no pudo ser reparado por Astor4Android:

```
2 app-android/app/src/main/java/de/tutao/tutanota/Utils.java
@@ -145,7 +145,7 @@ public static Map<String, String> jsonObjectToMap(JSONObject jsonObject) throws
145 145     }
146 146
147 147     static boolean isColorLight(String c) {
148     -       if (c.charAt(0) != '#' || c.length() != 6) {
    148 +       if (c.charAt(0) != '#' || c.length() != 7) {
149 149             throw new IllegalArgumentException("Invalid color format: " + c);
150 150         }
```

Podemos observar en la siguiente tabla como funcionaron las distintas técnicas de reparación de programas y localización de fallas:

-	Ochiai	Op2	DStar	Barinel	Tarantula
JGenProg2	SI	SI	SI	SI	SI
jMutRepair	SI	SI	SI	SI	SI
jKali	SI	SI	SI	SI	SI

Los métodos de localización de fallas propusieron correctamente el conjunto de líneas sospechosas. Estas son las que están en el extracto de código exhibido.

El funcionamiento de cada algoritmo de reparación de programas fue el siguiente:

JKali:

Para las 5 técnicas, propuso las mismas 3 soluciones.

1.

```
if (c.charAt(0) != '#' || c.length() != 6) {
-    throw new java.lang.IllegalArgumentException(("Invalid color format: " + c));
}
```

2.

```
-if (c.charAt(0) != '#' || c.length() != 6) {
-    throw new java.lang.IllegalArgumentException(("Invalid color format: " + c));
-}
```

3.

```
-if (c.charAt(0) != '#' || c.length() != 6) {
+if (false) {
    throw new java.lang.IllegalArgumentException(("Invalid color format: " + c));
}
```

Se puede observar que las 3 son equivalentes, lo que hacen es evitar que se arroje una excepción.

JGenProg2:

Propuso las mismas soluciones que JKali.

JMutRepair:

Para las 5 técnicas, propuso las mismas 6 soluciones.

1.

```
-if (c.charAt(0) != '#' || c.length() != 6) {
+if (c.charAt(0) != '#' && c.length() != 6) {
    throw new java.lang.IllegalArgumentException(("Invalid color format: " + c));
}
```

2.

```
-if (c.charAt(0) != '#' || c.length() != 6) {
+if (c.charAt(0) != '#' || c.length() == 6) {
    throw new java.lang.IllegalArgumentException(("Invalid color format: " + c));
}
```

3.

```
-if (c.charAt(0) != '#' || c.length() != 6) {
+if (c.charAt(0) != '#' || c.length() <= 6) {
    throw new java.lang.IllegalArgumentException(("Invalid color format: " + c));
}
```

4.

```
-if (c.charAt(0) != '#' || c.length() != 6) {
+if (c.charAt(0) != '#' || c.length() < 6) {
    throw new java.lang.IllegalArgumentException(("Invalid color format: " + c));
}
```

5.

```
-if (c.charAt(0) != '#' || c.length() != 6) {
+if (c.charAt(0) != '#' || !(c.length() != 6)) {
    throw new java.lang.IllegalArgumentException(("Invalid color format: " + c));
}
```

6.

```
-if (c.charAt(0) != '#' || c.length() != 6) {
+if (!(c.charAt(0) != '#' || c.length() != 6)) {
    throw new java.lang.IllegalArgumentException(("Invalid color format: " + c));
}
```

Todas las soluciones propuestas son incorrectas, ya que lo único que hacen es modificar o quitar incorrectamente la validación sobre el formato de un string. Este es otro ejemplo de lo que ocurre cuando el conjunto de tests es incompleto para especificar la funcionalidad de la aplicación. En este caso solo había dos tests que ejercitaran *isColorLight*, ambos fallaban para $commit_{bug}$ y pasaban para $commit_{fix}$.

```
@Test
public void testIsColorLightPinkDark() {
        assertFalse(Utils.isColorLight("#B73A9A"));
}


@Test
public void testIsColorLightBlueLight() {
        assertTrue(Utils.isColorLight("#3A9AFF"));
}
```

La excepción era arrojada para strings que no tuvieran longitud igual a 6, y estos tests verifican strings de longitud 7. Faltan más tests que especifiquen que se busca con la validación en la guarda.

6.4.9. Sumarización de resultados

Podemos observar en la siguiente tabla estadísticas sobre cómo funcionaron los distintos métodos de localización de fallas, sobre un total de 8 bugs:

-	Encontró líneas sospechosas	Encontró la línea sospechosa indicada
Ochiai	100 %	75 %
Op2	87.5 %	50 %
DStar	62.5 %	37.5 %
Barinel	87.5 %	62.5 %
Tarantula	100 %	75 %

Podemos concluir que en nuestro benchmark los métodos de localización de fallas *Ochiai* y *Tarantula* fueron los que mejor funcionaron a la hora de encontrar un conjunto no vacío de líneas sospechosas, y a su vez, la línea dónde efectivamente se encontraba el error.

La tabla a continuación presenta estadísticas que reflejan el rendimiento de los distintos métodos de reparación de programas. Estos datos se originan de aplicar estos métodos utilizando los 35 conjuntos de líneas sospechosas identificados por las técnicas de localización de fallas, de un total de 40 posibles (5 técnicas de localización de fallas aplicadas a 8 bugs):

-	Solución correcta	Solución incorrecta	Sin solución
JGenProg2	8.57 %	28.57 %	62.86 %
JMutRepair	11.43 %	28.57 %	60 %
JKali	0 %	42.86 %	57.14 %

Podemos concluir que en nuestro benchmark las técnicas de reparación de programas que mejor funcionaron fueron **JGenProg2** y **JMutRepair**. A su vez, son las que más versatilidad tienen a la hora de proponer variantes de código que puedan reparar un error. De todas maneras, no podemos afirmar que son las mejores, solo que fueron las que mejor se adaptaron a nuestro benchmark.

La variación en el rendimiento entre **JGenProg2** y **JMutRepair** se debe a que la técnica de localización *Op2* no identificó correctamente la línea defectuosa en el bug **CatimaLoyalty-Miscelánea-4**, perjudicando las estadísticas de **JGenProg2**. Por otro lado, **JKali** fue técnica que presentó mas soluciones incorrectas.

6.4.10. Conclusiones sobre Astor4Android

Pudimos observar que el paso del tiempo afectó negativamente a Astor4Android. No solo necesitamos implementar ajustes para que la herramienta pudiera llevar a cabo reparaciones, sino que también encontramos casos en los que fue imposible procesar commits debido a que utilizaban versiones de JDK más actuales que las soportadas por la herramienta.

Es importante hacer hincapié en la fragilidad que supone el acoplamiento tecnológico a herramientas que experimentan cambios constantes, como es el caso de Gradle. Al actualizarse las versiones de estas herramientas, se pueden producir alteraciones en el

comportamiento interno, como la configuración de directorios donde se guardan las clases compiladas.

Estas modificaciones no se especifican en las notas de lanzamiento, por lo que pueden pasar inadvertidas y resultar en incompatibilidades o errores en el funcionamiento de herramientas como Astor4Android.

A pesar de estos incovenientes, una vez realizadas las modificaciones pertinentes, las técnicas de localización de fallas implementadas por Astor4Android lograron detectar dónde se encontraban la mayoría de las fallas, y las de reparación de programas lograron solucionar 2 bugs exitosamente. Este resultado es interesante, teniendo en cuenta que en DroidBugs exponen que no pudieron reparar ningún bug.

7. CONCLUSIONES

Construir un benchmark robusto para el análisis y evaluación de herramientas de reparación de programas en aplicaciones Android ha demostrado ser una tarea compleja. Nos sorprendió descubrir que una cuarta parte de las aplicaciones en F-droid carecían completamente de tests, y aún entre las que sí los tenían, notamos áreas significativas sin cobertura. Otro factor que contribuye a la complejidad de construir un benchmark efectivo es la propia dinámica del desarrollo de software. Dado que es raro que los desarrolladores realicen commits que incluyan código sabiendo que romperán un test, es poco probable encontrar ejemplos de commits con fallas que sean corregidas en commits subsiguientes. Este desafío se intensifica considerando que las reparaciones de código que se estudian deben ser de un tamaño que coincida con lo que una herramienta de reparación automática podría razonablemente lograr.

Es posible que los desarrolladores prioricen con frecuencia agregar funcionalidad sobre otros factores como los atributos de calidad del código: la facilidad para ser probado, su extensibilidad, legibilidad, etc. Experimentamos dos tipos de dificultades al intentar mejorar la cobertura: la primera fue la complejidad de añadir tests a clases sin modificar el código existente, y la segunda fue el desafío de comprender el comportamiento esperado de la aplicación para escribir tests adecuados.

La ausencia de tests en muchas aplicaciones móviles es problemática, especialmente considerando que los errores pueden persistir en las versiones que los usuarios tienen instaladas si no se llevan a cabo actualizaciones. Estas circunstancias hacen evidente la necesidad de fomentar una cultura más fuerte en torno a la realización de tests durante el desarrollo.

Observamos que la eficacia de las técnicas de localización de fallas y reparación de programas está intrínsecamente ligada a la calidad del conjunto de tests disponibles. Un solo test defectuoso, o un caso de uso faltante, puede afectar negativamente todo el proceso. Esto quedó claro en nuestros resultados, donde la mejora del conjunto de tests resultó en reparaciones más precisas.

Mientras que las técnicas de localización de fallas generalmente se desempeñan adecuadamente, identificando la región de código problemática, las técnicas de reparación mostraron un rendimiento inferior. Esto se debe a la naturaleza más compleja del problema de reparación y a las limitaciones inherentes a las herramientas existentes como JKali, JMutRepair y JGenProg2. Aunque JGenProg2 posee una capacidad teórica para solucionar una amplia gama de problemas, no mostró una mejora radical en nuestro análisis comparativo.

Una de las estrategias que podría mitigar algunos de los desafíos que enfrentamos es el uso de *Test Driven Development* (TDD). Este enfoque de diseño de software no solo ayuda a evitar algunas de las malas prácticas en el desarrollo, sino que también contribuye a una mejor cobertura de tests y una especificación más clara del comportamiento esperado del programa. En un escenario donde TDD se aplica de manera rigurosa, esperamos que la etapa de síntesis de pruebas se simplifique debido a la existencia de un conjunto de pruebas más completo. Además, la etapa de reparación de programas se beneficiaría de tener especificaciones más claras, lo que podría mejorar la eficacia de las herramientas de reparación.

Concluimos que aquellos desarrolladores interesados en la utilización de técnicas de reparación y localización de fallas en aplicaciones Android deben prestar especial atención a la calidad del conjunto de tests. Si bien las técnicas evaluadas demuestran un rendimiento notable en la identificación de líneas defectuosas, es evidente que el ámbito de la reparación de programas requiere más avances para alcanzar un nivel de efectividad comparable. Además, debido a la constante evolución del ecosistema Android, observamos que es importante adaptar las herramientas para garantizar su correcto funcionamiento a lo largo del tiempo.

Bibliografía

[1] https://en.wikipedia.org/wiki/Usage_share_of_operating_systems#Mobile_devices.

[2] Larissa Azevedo, Altino Dantas, and Celso G. Camilo-Junior. Droidbugs: An android benchmark for automated program repair. 2018.

[3] https://www.android.com/what-is-android/.

[4] https://developer.android.com/studio/build?hl=es-419.

[5] https://developer.android.com/studio/projects?hl=es-419#ProjectView.

[6] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. Repairing crashes in android apps. In Proceedings of the 40th International Conference on Software Engineering, ICSE '18, page 187–198, New York, NY, USA, 2018. Association for Computing Machinery.

[7] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis, pages 437–440, San Jose, CA, USA, July 2014. Tool demo.

[8] https://developer.android.com/topic/performance/vitals/crash?hl=es-419.

[9] https://docs.github.com/en/rest/overview/resources-in-the-rest-api?apiVersion=2022-11-28#rate-limiting.

[10] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018, pages 908–911, New York, New York, USA, 2018. ACM Press.

[11] https://developer.android.com/studio/write/lint?hl=es-419#gradle.

[12] https://blog.gradle.org/jcenter-shutdown.

[13] Matias Martinez and Martin Monperrus. Astor: a program repair library for java. pages 441–444, 2016.

[14] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan Gemund. A practical evaluation of spectrum-based fault localization. Journal of Systems and Software, 82:1780–1792, 11 2009.

[15] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol., 20(3), aug 2011.

[16] W. Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. Reliability, IEEE Transactions on, 63:290–308, 03 2014.

[17] James Jones and Mary Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. pages 273–282, 11 2005.

[18] Rui Abreu, Peter Zoeteweij, and Arjan Gemund. Spectrum-based multiple fault localization. pages 88–99, 11 2009.

[19] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, page 24–36, New York, NY, USA, 2015. Association for Computing Machinery.

[20] Vidroha Debroy and W. Wong. Using mutation to automatically suggest fixes for faulty programs. pages 65–74, 01 2010.

[21] Claire Le Goues, Thanh Vu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. IEEE Transactions on Software Engineering, 38(1):54–72, 2012.

[22] https://github.com/SpoonLabs/astor/issues/307.

[23] https://developer.android.com/build/releases/past-releases/agp-7-0-0-release-notes#jdk-11.

[24] https://github.com/I4Soft/Astor4Android.

[25] https://developer.android.com/build/releases/past-releases/agp-2-3-0-release-notes.

[26] https://github.com/I4Soft/DroidBugs.

[27] https://github.com/I4Soft/Astor4Android/blame/master/src/main/java/br/ufg/inf/astor4android/entities/AndroidProject.java#L269.

[28] https://docs.gradle.org/5.1.1/release-notes.html.

[29] https://developer.android.com/studio/releases/gradle-plugin?hl=es-419.

[30] https://developer.android.com/build/releases/past-releases/.

[31] https://github.com/powermock/powermock/wiki/Code-coverage-with-JaCoCo#on-the-fly-instrumentation.