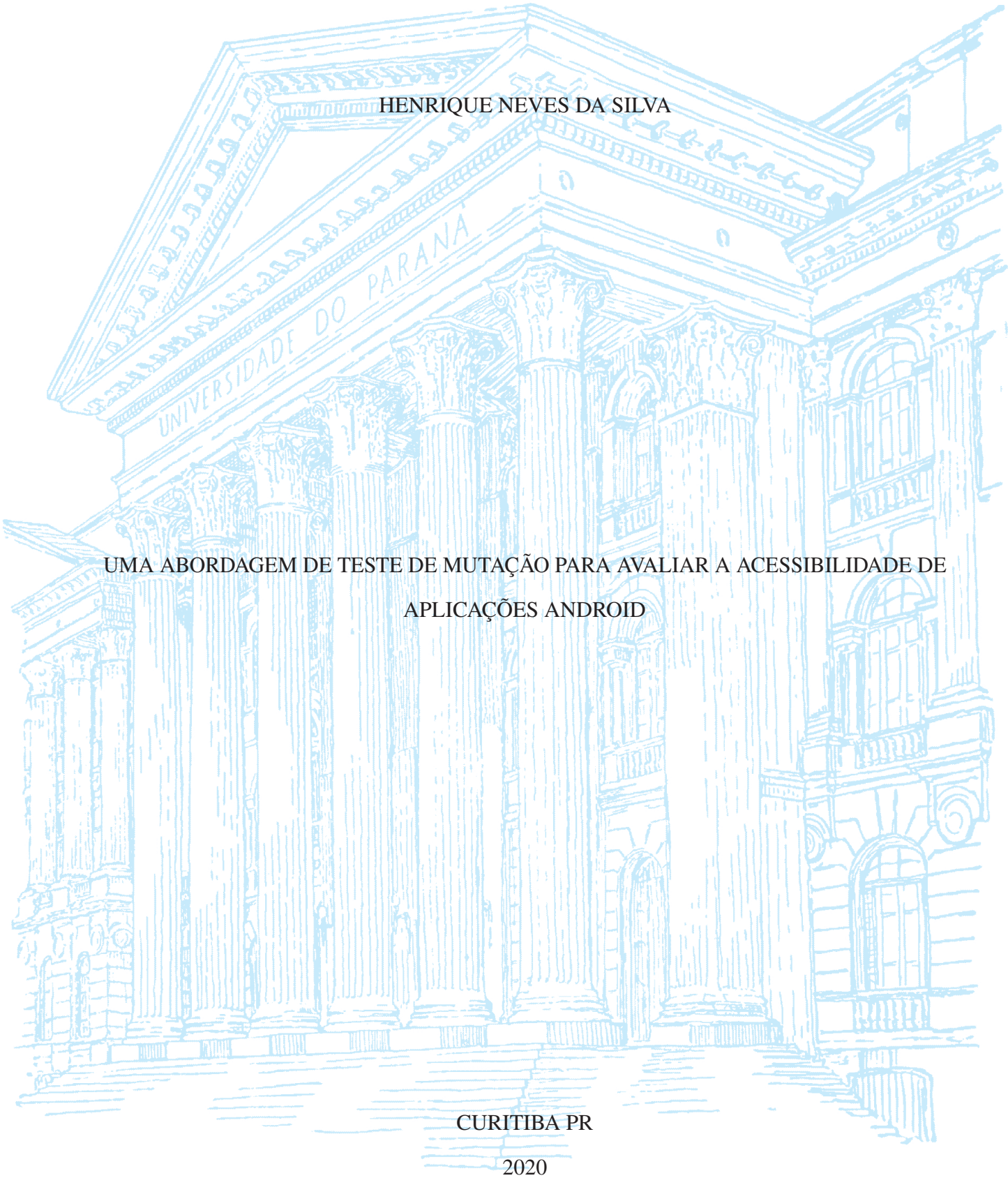
UNIVERSIDADE FEDERAL DO PARANÁ

HENRIQUE NEVES DA SILVA



UMA ABORDAGEM DE TESTE DE MUTAÇÃO PARA AVALIAR A ACESSIBILIDADE DE APLICAÇÕES ANDROID

CURITIBA PR

2020

HENRIQUE NEVES DA SILVA

UMA ABORDAGEM DE TESTE DE MUTAÇÃO PARA AVALIAR A ACESSIBILIDADE DE APLICAÇÕES ANDROID

Tese apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Silvia Regina Vergilio.

Coorientador: André Takeshi Endo.

CURITIBA PR

2020

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

S586a Silva, Henrique Neves da
Uma abordagem de teste de mutação para avaliar a acessibilidade de aplicações Android [recurso eletrônico] / Henrique Neves da Silva. – Curitiba, 2020.

Tese - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2020.

Orientadora: Silvia Regina Vergilio.
Coorientador: André Takeshi Endo.

1. Android (Recurso eletrônico). 2. Aplicativos móveis. I. Universidade Federal do Paraná. II. Vergilio, Silvia Regina. III. Endo, André Takeshi. IV. Título.

CDD: 005.3

Bibliotecária: Vanusa Maciel CRB- 9/1928

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **HENRIQUE NEVES DA SILVA** intitulada: **Uma abordagem de teste de mutação para avaliar a acessibilidade de aplicações Android**, sob orientação da Profa. Dra. SILVIA REGINA VERGILIO, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 17 de Dezembro de 2020.

Assinatura Eletrônica
21/12/2020 13:13:59.0
SILVIA REGINA VERGILIO
Presidente da Banca Examinadora (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
17/12/2020 22:33:29.0
NATASHA MALVEIRA COSTA VALENTIM
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
18/12/2020 11:11:44.0
ARILO CLAUDIO DIAS NETO
Avaliador Externo (UNIVERSIDADE FEDERAL DO AMAZONAS)

Dedico este trabalho à minha família, principalmente aos meus pais, que sempre apoiaram e incentivaram meus estudos, fornecendo todo e qualquer suporte necessário para que eu pudesse me dedicar exclusivamente à minha formação pessoal e profissional.

AGRADECIMENTOS

Em primeiro lugar, agradeço à minha família, especialmente meus pais Célia e Claudinei e minha vó Olinda, que sempre estiveram presentes e me incentivaram a percorrer e continuar essa jornada acadêmica.

Agradeço aos meus orientadores Profa. Dra. Silvia Regina Vergilio e Prof. Dr. André Takeshi Endo, pela orientação, paciência, apoio e confiança com que me guiaram nesta trajetória.

Meus agradecimentos aos meus amigos Guilherme, Fabian e Alex. Que foram irmãos na amizade e fizeram parte da minha formação e que vão continuar presentes em minha vida com certeza.

Agradeço a todos os professores por me proporcionar o conhecimento não apenas racional, mas a manifestação do caráter e afetividade da educação no processo de formação profissional.

Agradeço também aos meus colegas de trabalho da UTFPR, que estavam presentes no dia a dia desta jornada.

Enfim, a todos os que por algum motivo contribuíram para a realização deste trabalho.

RESUMO

Dispositivos móveis e suas aplicações estão presentes em grande parte das atividades cotidianas e desempenham um papel importante para pessoas com alguma deficiência. No entanto, tornar as aplicações mais acessíveis ainda é um desafio. Para tanto, os desenvolvedores podem recorrer a padrões e diretrizes que sintetizam as melhores práticas em relação ao desenvolvimento de produtos acessíveis. Dentre estes padrões, destaca-se o W3C's *Web Content Accessibility Guideline* (WCAG). O WCGA é um dos padrões mais populares e engloba várias diretrizes, cada uma relacionada a diferentes critérios de sucesso, agrupados em quatro princípios de acessibilidade. Em suma, WCAG abrange recomendações para tornar produtos digitais mais acessíveis para indivíduos com cegueira, baixa visão, surdez, perda auditiva, limitação de movimento, deficiência na fala, fotossensibilidade, dificuldades de aprendizagem, e limitações cognitivas. Ferramentas automáticas de teste de acessibilidade também podem ajudar na tarefa de tornar as aplicações mais acessíveis, mas apresentam algumas limitações. Elas produzem relatórios sobre falhas de acessibilidade que geralmente cobrem apenas um subconjunto da aplicação, porque são dependentes do conjunto de teste disponível. Com o objetivo de auxiliar no aprimoramento e/ou avaliação dos conjuntos de testes, bem como contribuir para aumentar o desempenho das ferramentas de teste de acessibilidade, este trabalho apresenta `AccessibilityMDroid`, uma abordagem de teste de mutação. A abordagem inclui: (i) um conjunto de operadores de mutação que descrevem defeitos derivados da negação dos princípios e critérios de sucesso do padrão WCAG; (ii) um processo para analisar os mutantes; e (iii) implementação de uma ferramenta para apoiar o processo proposto. Os resultados da avaliação com sete aplicações de código-aberto mostram que a abordagem é aplicável na prática. No processo de geração, 257 mutantes foram criados. A partir do processo proposto, os conjuntos de teste que acompanham os repositórios das aplicações melhoraram em média 932,8% o número de falhas de acessibilidade reveladas.

Palavras-chave: Aplicações Móveis. Teste de Mutação. Android. Acessibilidade.

ABSTRACT

Mobile devices and their applications are present in many everyday activities and play an important role for people with a disability. However, making applications more accessible is still a challenge. To this end, developers can resort to standards and guidelines that summarize best practices regarding the development of accessible products. Among these standards, W3C's Web Content Accessibility Guideline (WCAG) stands out. WCGA is one of the most popular standards, and encompasses several guidelines related to different success criteria, grouped into four principles. In short, WCAG covers recommendations to make digital products more accessible to individuals with blindness, low vision, deafness, hearing loss, movement limitation, speech impairment, photosensitivity, learning difficulties, and cognitive limitations. Automatic accessibility testing tools can also help with the task of making applications more accessible, but they have some limitations. They produce reports on accessibility failures that generally cover only a subset of the application because they are dependent on the available test suite. To assist in the improvement and/or evaluation of test suites and contribute to increasing the performance of the accessibility test tools, this work presents `AccessibilityMDroid`, a mutation testing approach. The approach includes: (i) a set of mutation operators that describe faults derived from the negation of WCAG principles and success criteria; (ii) a process for analyzing mutants; and (iii) a tool to support the proposed process. The results of the evaluation with seven open source applications show that the approach is applicable in practice. In the process of generation, 257 mutants were created. From the proposed process, the test suites that accompany the application improved on average 932.8 % the number of accessibility flaws revealed.

Keywords: Mobile Apps. Mutation Testing. Android. Accessibility.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Uma aplicação móvel é um programa de software executado em dispositivos móveis. Atualmente, este tipo de aplicação desempenha um papel essencial no dia a dia de bilhões de pessoas (Cisco, 2018), incluindo pessoas com deficiência. De acordo com a Organização Mundial de Saúde (OMS), estima-se que mais de um bilhão de pessoas são afetadas por alguma forma de deficiência (Hartley, 2011). Por exemplo, para pessoas com alguma restrição visual, dispositivos móveis podem ser um importante aliado para facilitar a independência na realização de diversas tarefas: entender estrutura de documentos de texto, comunicação ao longo de aplicações de mídia social, identificação de produtos em prateleiras de supermercado, movimentação entre obstáculos, entre outras (Acosta-Vargas et al., 2020). A necessidade de garantir que todas as vantagens da tecnologia são acessíveis para todo indivíduo é uma questão de fundamental importância. Consoante a ISO/IEC 25010 (2011), acessibilidade pode ser definida como a avaliação do grau em que o produto foi projetado para atender a usuários com necessidades especiais.

Segundo Ballantyne et al. (2018), grande parte das pesquisas em acessibilidade é dedicada para a Web. Acessibilidade no contexto móvel possui igual importância, mas é difícil aplicar os mesmos guias de acessibilidade da Web neste contexto, pelo fato de as aplicações móveis operarem em diversas plataformas (por exemplo Android, iOS, Windows Phone) e ainda possibilitar interação por diferentes meios como áudio, gestos e toque. Os poucos estudos relacionados à acessibilidade de aplicações móveis apontam a falta de ferramentas adequadas, guias e políticas para avaliar aplicações móveis (Acosta-Vargas et al., 2020).

Devido à relevância e demanda por aplicações móveis de alta qualidade no mercado, torna-se cada vez mais importante não só garantir a acessibilidade destas aplicações, mas também que elas possuam o comportamento esperado. Uma maneira de se avaliar o comportamento da aplicação móvel é com a realização de testes, mas ainda há limitações quanto à informação de teste gerada e preocupação quanto à efetiva melhora dos conjuntos de teste das aplicações (Linares-Vásquez et al., 2017).

1.1 MOTIVAÇÃO

É possível avaliar o conjunto de teste da aplicação móvel Android com a inserção de pequenos defeitos no código-fonte da aplicação. A versão com defeitos inseridos é chamada de mutante. Esta abordagem foi originalmente proposta por DeMillo et al. (1978) e é conhecida como Análise de Mutantes. A Análise de Mutantes é um critério de teste da técnica baseada em defeitos. A ideia é que a partir de operadores de mutação que provocam mudanças simples no programa em teste P, e que descrevam possíveis defeitos, sejam derivadas versões chamadas mutantes. O objetivo é gerar casos de teste capazes de distinguir P de seus mutantes, ou seja, que quando executado com cada mutante m produza uma saída diferente da saída de P. Se o resultado de P estiver correto, ele está potencialmente livre do defeito descrito por m. Se a saída é diferente, m é dito morto. Ao final tem-se uma medida chamada escore de mutação que é dada pela razão entre número de mutantes mortos e o número de mutantes gerados. O escore de mutação pode ser usado para avaliar um conjunto de casos de teste existente, para melhorar a atividade de teste e também para considerar se um programa foi testado o suficiente.

O teste de mutação provou ser eficaz em diferentes domínios e contextos (Jia e Harman, 2011). Mais recentemente, tem sido usado no teste de requisitos não-funcionais, como desempenho (Lisper et al., 2017) e consumo de energia (Jabbarvand e Malek, 2017).

Existem algumas iniciativas que exploram o teste de mutação de aplicações Android (Wei, 2015; Deng et al., 2015; Jabbarvand e Malek, 2017; Luna e El Ariss, 2018; Escobar-Velásquez e Linares-Vásquez, 2019; Liu et al., 2020), mas esses trabalhos não estão focados em testes de acessibilidade.

Existem ferramentas de teste de acessibilidade que geralmente são baseadas em guias existentes. Um dos padrões mais populares é WCAG (W3C's Web Content Accessibility Guideline). O guia WCAG cobre recomendações para pessoas com cegueira e baixa visão, surdez e perda auditiva, movimento limitado, limitações cognitivas, fala e dificuldades de aprendizagem. WCAG engloba várias diretrizes, cada uma relacionada a diferentes critérios de sucesso, agrupados em quatro princípios de acessibilidade. Algumas ferramentas produzem, a partir de um conjunto de teste executado, um relatório de violações de acessibilidade para a aplicação Android. Exemplos dessas ferramentas são Espresso (Google, 2019b), A11y (Toff, 2019) e MATE (Eler et al., 2018). Elas podem realizar análises estáticas ou dinâmicas (Silva et al., 2018).

Um número limitado de violações pode ser verificado por ferramentas estáticas, mas a análise dinâmica tende a ser mais cara. Outra limitação é que as violações de acessibilidade verificadas pelas ferramentas são limitadas pelo conjunto de teste usado. Ele cobre apenas um subconjunto da aplicação devido a *scripts* de teste fracos ou algoritmos de geração de dados de teste de entrada limitados (Silva et al., 2018). Ferramentas geralmente usadas para geração de dados de teste, como Monkey (Moher et al., 2009), Sapienz (Mao et al., 2016), Stoat (Su et al., 2017) e APE (Gu et al., 2019), são focadas no comportamento funcional, cobertura de código ou *crashes*. Nesse sentido, a hipótese deste trabalho é que uma abordagem de mutação específica para testes de acessibilidade pode auxiliar na melhoria e/ou avaliação de conjuntos de teste gerados, bem como contribuir para aumentar o desempenho das ferramentas de teste de acessibilidade.

1.2 OBJETIVOS

Dado o contexto e a motivação descritos acima, esta dissertação apresenta `AccessibilityMDroid`, uma abordagem de mutação para o teste de acessibilidade de aplicações Android. Para a abordagem, foi definido um modelo de defeitos que está relacionado ao não cumprimento dos princípios e critérios de sucesso do guia WCAG. Definiu-se um conjunto de 9 operadores que removem alguns atributos da API do Android (*code elements*), os mais comumente usados nas aplicações, e cuja ausência pode implicar em violações de acessibilidade. Também foi definido um processo de análise de mutantes que usa relatórios de acessibilidade de ferramentas para distinguir mutantes mortos. O processo é implementado usando os relatórios produzidos pela Espresso (Google, 2019b) e avaliados com 7 aplicações de código-aberto. Dessa maneira, a abordagem `AccessibilityMDroid` pode: (i) ser usada como um critério para geração e/ou avaliação de dados de teste, ajudando os desenvolvedores a medir a qualidade de seus conjuntos de testes sob uma perspectiva de acessibilidade; (ii) ser utilizada para avaliar as ferramentas de teste de acessibilidade disponíveis no mercado e na academia; e (iii) contribuir para a área emergente de teste de mutação para propriedades não-funcionais, representando um primeiro passo para tornar possível o teste de mutação de acessibilidade. O trabalho serve desta maneira como base para direcionar pesquisas futuras e estimular a comunidade acadêmica a criar ferramentas que explorem mais este campo de pesquisa.

1.3 ORGANIZAÇÃO DO TRABALHO

Este trabalho foi divido da seguinte forma. O Capítulo 2 apresenta o embasamento teórico necessário para ambientação com o tema, introduzindo a área de teste de software e aplicações móveis. O Capítulo 3 elenca os trabalhos relacionados e está separado em duas seções: Teste de Acessibilidade e Teste de Mutação para aplicações Android. O Capítulo 4 define a abordagem proposta por este trabalho, explicando os principais aspectos: geração dos mutantes e estratégia de análise dos mutantes. Capítulo 5 descreve como a abordagem foi avaliada. Os procedimentos envolvidos no estudo experimental são detalhados e os resultados são discutidos. Finalmente, no Capítulo 6, são apresentadas as contribuições e as oportunidades que podem ser exploradas em trabalhos futuros. O trabalho contém dois apêndices. O Apêndice A apresenta resultados de um mapeamento sistemático da área de teste de mutação para aplicações Android. O Apêndice B contém um artigo publicado com resultados de um estudo conduzido para entender a relação entre os recursos da API Android com questões de acessibilidade.

2 FUNDAMENTOS

Este capítulo apresenta conceitos sobre importantes temas desta dissertação: Teste de Software (Seção 2.1) e Aplicações Móveis (Seção 2.2).

2.1 TESTE DE SOFTWARE

"Teste de software é o processo de executar um programa com o objetivo de revelar defeitos" (Myers et al., 2011, Capítulo 1). O teste compreende um conjunto de atividades de verificação e validação do software que podem ser planejadas com antecedência e executadas sistematicamente (Pressman e Maxim, 2016), além de ser uma importante área de estudo dentro da Engenharia de Software. Miller (1977) diz que o teste de software está relacionado com garantia de qualidade: "a motivação que está por trás do teste de programas é a confirmação da qualidade do software com métodos que podem ser econômica e efetivamente aplicados a todos os sistemas, de grande e pequena escala".

Para que a qualidade do software seja assegurada, existe uma série de atividades, coletivamente chamadas de Validação, Verificação e Teste, ou VV&T (Delamaro et al., 2017, Capítulo 1). Ao longo da literatura existem diferentes terminologias envolvendo os conceitos de teste de software, Ammann e Offutt (2016, Capítulo 1) definem os termos Verificação e Validação como:

Definição 2.1.1 (Verificação)**.** *O processo de determinar se os produtos de uma fase do processo de desenvolvimento de software atendem aos requisitos estabelecidos na fase anterior.*

Definição 2.1.2 (Validação)**.** *O processo de avaliação de software no final do desenvolvimento do software para garantir a conformidade com o uso pretendido.*

As atividades VV&T não se restringem ao produto final, elas são importantes pois permitem a eliminação de erros desde as fases iniciais do processo de desenvolvimento, o que, em geral, representa uma economia significativa de recursos (Delamaro et al., 2017, Capítulo 1). É interessante também, a partir da literatura tradicional, distinguir os termos: defeito, engano, erro e falha (Ammann e Offutt, 2016, Capítulo 1).

- **Defeito**. Do inglês, *fault*. Um defeito estático no código-fonte do software.
- **Engano**. Do inglês, *mistake*. Uma ação humana que produz um defeito.
- **Erro**. Do inglês, *error*. Um estado interno incorreto que é causado por um defeito no software.
- **Falha**. Do inglês, *failure*. Externalização do erro, como sendo um comportamento incorreto do software com relação ao que é esperado.

Delamaro et al. (2017, Capítulo 1) complementam a definição de erro e falha, explicando o conceito de estado de um programa. Esse estado da execução de um programa, é dado pelo valor da memória (ou das variáveis do programa) e do apontador de instruções. Sendo assim, um estado inconsistente ou inesperado indica a existência de um erro. Tal estado pode levar a uma falha, ou seja, pode fazer com que o resultado produzido pela execução seja diferente do resultado esperado.

A execução de um programa com defeito em seu código-fonte pode não acarretar em um comportamento inesperado (falha). A fim de descrever o "gatilho" que implica em uma saída não esperada, Ammann e Offutt (2016, Capítulo 2) definem um conjunto com quatro condições, conhecido como modelo "RIPR" ou modelo de falha.

1. **Alcançabilidade**. Do inglês, ***R**eachability*. O local que contém defeitos do programa deve ser alcançado.

2. **Infecção**. Do inglês, ***I**nfection*. Após a execução do local que possui um defeito, o estado do programa deve ser incorreto.

3. **Propagação**. Do inglês, ***P**ropagation*. O estado infectado deve causar alguma saída incorreta ou estado final do programa deve ser incorreto.

4. **Revelabilidade**. Do inglês, ***R**evealability*. O testador deve ser capaz de observar a porção incorreta do estado final do programa.

Para melhor entender o cenário em que as quatro condições são satisfeitas, com objetivo de encontrar um defeito no programa, é necessário definir mais alguns conceitos importantes (Delamaro et al., 2017, Capítulo 1):

Definição 2.1.3 (Dado de Teste)**.** *Elemento do domínio de entrada do programa necessário para completar alguma execução do software em teste.*

Definição 2.1.4 (Resultado Esperado)**.** *É o resultado que será produzido durante a execução do dado de teste se, e somente se, o programa satisfaz seu comportamento pretendido.*

Definição 2.1.5 (Caso de Teste - CT)**.** *Par formado por um dado de teste mais o resultado esperado.*

Definição 2.1.6 (Oráculo de Teste)**.** *Determina se um caso de teste leva o programa ao resultado esperado.*

A composição do caso de teste e oráculo de teste procura revelar defeitos no programa, e assim formar a realização do modelo RIPR. A Figura 2.1 ilustra como o oráculo é utilizado. Primeiramente o programa quando executado com o dado de teste atinge o local do defeito do programa. A execução do local com defeito, infecta o programa e o leva a um estado intermediário incorreto. Este mesmo estado incorreto deve infectar o restante da execução e causar um estado final incorreto. A partir do oráculo, o testador observa o estado final incorreto do programa.

2.1.1 Critérios de Teste

O fluxo de trabalho do teste de software pode ser composto por mais de um engenheiro de teste. De acordo com Ammann e Offutt (2016), o engenheiro de teste é um cargo da Tecnologia da Informação (TI) encarregado das tarefas de projeto de dados de teste, implementação e execução dos dados de teste, e por fim, análise e entrega dos resultados para os desenvolvedores e gerentes. Existem diferentes formas de agrupar as atividades de teste. De forma genérica e aproximadamente cronológica os grupos podem ser (Tian, 2005, Capítulo 7):

1. **Planejamento de teste e preparação.** Define os objetivos do teste, seleciona uma estratégia geral de teste e prepara casos de teste específicos e o procedimento geral de teste.

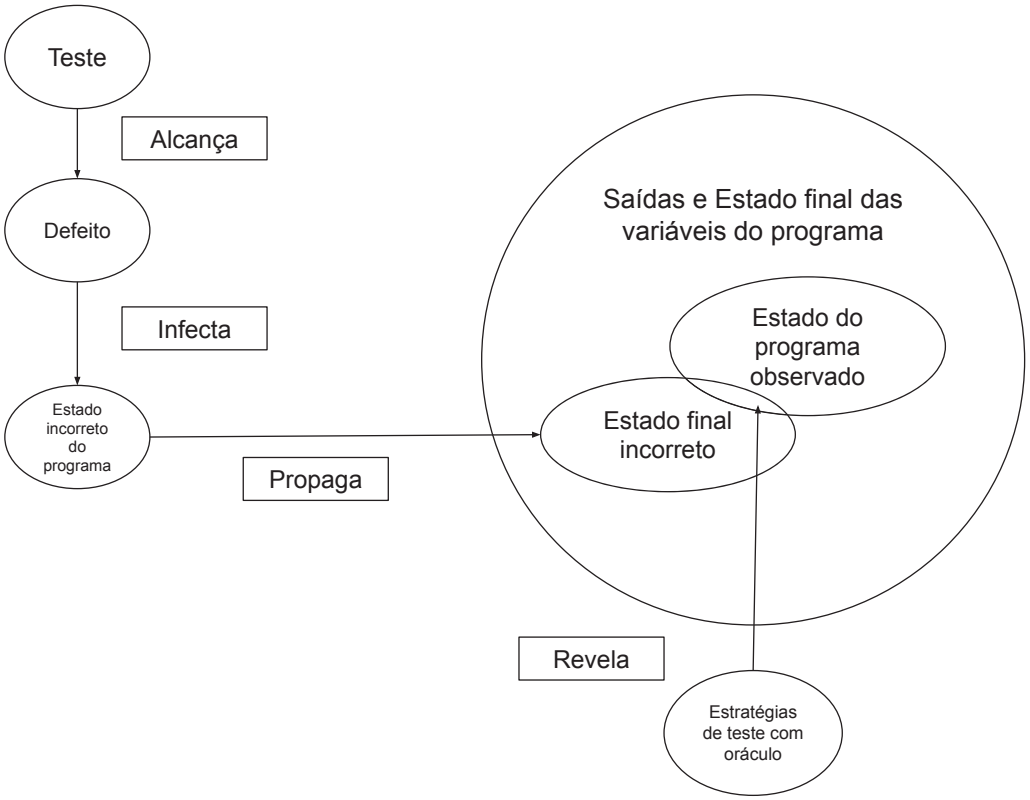


Figura 2.1: Modelo RIPR, adaptado de Li e Offutt (2016).

2. **Execução de teste e atividades relacionadas.** Inclui observação relacionada e medição do comportamento do produto.

3. **Análise e acompanhamento.** Inclui verificação e análise de resultados para determinar se uma falha foi observada e, em caso afirmativo, as atividades de acompanhamento são iniciadas e monitoradas para garantir a remoção das causas ou defeitos subjacentes que levaram às falhas observadas em primeiro lugar.

Durante a etapa de planejamento de teste, definem-se os critérios de cobertura a fim de fornecer uma maneira sistemática de projetar casos de teste com nível satisfatório de qualidade. Critério de cobertura pode ser definido como (Ammann e Offutt, 2016, Capítulo 2):

Definição 2.1.7 (Critério de Cobertura)**.** *Os critérios formais de cobertura fornecem aos engenheiros maneiras de decidir quais dados de teste usar durante o teste, aumentando a probabilidade de o testador encontrar problemas no programa e fornecendo maior garantia de que o software é de alta qualidade e confiabilidade. Os critérios de cobertura também fornecem regras de parada para que os engenheiros de teste considerem a atividade de teste encerrada.*

O critério escolhido depende da natureza da fonte da informação em que os dados de teste são derivados. Podem ser classificados em três tipos: funcional, estrutural e baseado em defeitos (Maldonado e Jino, 1991).

- **Teste Funcional**. "Técnica utilizada para se projetar casos de teste na qual o programa ou sistema é considerado uma caixa preta e, para testá-lo, são fornecidas entradas e avaliadas as saídas geradas para verificar se estão em conformidade com os objetivos especificados" (Delamaro et al., 2017, Capítulo 2). A visão que se tem do programa é a

visão do usuário, pois o testador não tem conhecimento do comportamento interno do programa (Myers et al., 2011, Capítulo 2). Alguns exemplos de critérios funcionais são: Partição em Classes de Equivalência, Análise do Valor Limite, e Grafo de Causa-Efeito.

- **Teste estrutural**. Também conhecido como teste de caixa branca, é uma técnica em que os testes são baseados na lógica interna do programa (Myers et al., 2011). A lógica interna do programa é extraída a partir de ramos, condições individuais e sentenças (Ammann e Offutt, 2016, Capítulo 2). Os critérios pertencentes à técnica estrutural são classificados com base na complexidade, no fluxo de controle e no fluxo de dados (Delamaro et al., 2017, Capítulo 4).

- **Teste Baseado em Defeito**. Além de testar com base na especificação, implementação e uso dos programas e outros artefatos, o teste pode ser baseado nos defeitos descobertos ou defeitos em potencial (Tian, 2005, Capítulo 12). Teste baseado em defeito utiliza o conhecimento de erros comuns ao longo do desenvolvimento de software para definir os requisitos de teste (Endo, 2013). Exemplos de critérios desta técnica são Semeadura de Erro e Análise de Mutantes ou Teste de Mutação.

O teste de mutação é de especial interesse para o presente trabalho e por isso é descrito em detalhes na próxima seção.

2.1.2 Teste de Mutação

Teste de mutação é um critério proposto por DeMillo et al. (1978) e que engloba um conjunto de atividades usadas para validação e verificação do software. É classificado como um critério da técnica baseada em defeitos, pois tem a premissa de que defeitos artificiais podem ajudar a encontrar defeitos reais. Trata-se de um critério de teste baseado na manipulação da sintaxe do programa. Seu desempenho foi comprovado empiricamente como eficaz para ajudar os testadores a gerar dados de teste de alta qualidade e avaliar conjuntos de teste pré-existentes projetados por outras técnicas de teste (Andrews et al., 2005; Just et al., 2014; Papadakis et al., 2018). O teste de mutação é fundamentado em dois pressupostos, a hipótese do programador competente e o efeito de acoplamento (DeMillo et al., 1978):

- **Hipótese do Programador Competente**. Parte do pressuposto de que o programador tem conhecimento e experiência o suficiente para deixar o código-fonte do programa próximo do ideal. Os erros cometidos no programa são relativos a pequenos detalhes como nomeação incorreta da variável ou troca de operadores.

- **Efeito de Acoplamento**. A partir desta hipótese, determina-se que um conjunto de teste capaz de detectar erros relativos a pequenos detalhes, também será capaz de detectar erros mais complexos.

A partir dessas duas premissas o testador deve avaliar quais defeitos serão inseridos no código original do programa. Quando o defeito, ou possível engano do programador, é inserido no código, gera-se uma versão alternativa do programa também conhecido como mutante (Delamaro et al., 2017, Capítulo 5):

Definição 2.1.8 (Mutante)**.** *Um mutante M é a versão modificada de um programa P com auxílio de um operador de mutação.*

Definição 2.1.9 (Operador de Mutação - OM)**.** *Representa um possível equívoco do programador, mas que ainda mantém a integridade sintática do programa.*

Uma vez que o operador de mutação é aplicado ao programa original, e este possui estrutura sintática que um dado operador consegue modificá-la então essas modificações são realizadas e um conjunto de mutantes é gerado (Fischer, 2015). Em geral, a aplicação de um operador de mutação gera mais de um mutante, pois, se o programa P contém várias entidades que estão no domínio do operador, então esse operador é aplicado a cada uma dessas entidades, uma de cada vez (Delamaro et al., 2017, Capítulo 5).

O operador de mutação pode ser definido e usado em diversos contextos e linguagens. Por exemplo, Agrawal et al. (1999) definem um conjunto de 77 operadores de mutação para a linguagem C, que foram separados em diferentes classes: *Statement Mutations*, *Operator Mutations*, *Variable Mutations* e *Constant Mutations*. Em Kim et al. (1999), foi proposto o uso da técnica *Hazard and Operability Studies* (HAZOP) para se chegar ao conjunto de 13 operadores de mutação em classes da linguagem Java. No contexto desta dissertação operadores de Java estão mais relacionados.

A critério de exemplo, considere o trabalho de Ma et al. (2005) que propõe a ferramenta muJava. Esta ferramenta explora os conceitos de OO dentro da linguagem Java. A ferramenta trabalha com 43 operadores, sendo 15 operadores no nível de método e 28 operadores no nível de classe. Os operadores do nível de método são classificados como: *Arithmetic Operators*, *Relational Operators*, *Conditional Operators*, *Shift Operators*, *Logical Operators*, *Assignment Operators* e *Deletion Operators*.

O Código 2.1 apresenta o caso de um dos possíveis operadores elencados por Ma et al. (2005). A função descrita tem como objetivo determinar se um número inteiro é par. O operador ROR (*Relational Operator Replacement*) atua sobre cada expressão lógica do programa, trocando os operadores relacionais.

Código 2.1: Programa original.

```
1  String isPar (int numero){
2     if (numero % 2 == 0) {
3        return "PAR";
4     } else
5        return "IMPAR";
6  }
```

Ao aplicar o operador ROR no programa original, os mutantes representados pelos Códigos 2.2 e 2.3 são gerados. O operador relacional em destaque nos códigos refere-se à alteração realizada no código-fonte do programa original.

Código 2.2: Mutante m_1.

```
1  String isPar (int numero){
2     if (numero % 2 != 0) {
3        return "PAR";
4     } else
5        return "IMPAR";
6  }
```

Código 2.3: Mutante m_2.

```
7  String isPar (int numero){
8     if (numero % 2 >= 0) {
9        return "PAR";
10    } else
11       return "IMPAR";
12 }
```

Com os mutantes gerados, o engenheiro de teste determina um conjunto de teste T que verifica se o comportamento do programa é o esperado (Definição 2.1.6). Suponha que o conjunto T é formado por $T = \{2, 5, 1\}$. Como o programa original avalia se um valor inteiro é par ou ímpar, a priori tem-se como oráculo o seguinte conjunto: {PAR, IMPAR, IMPAR}. Caso a execução do programa P confirme a saída prevista pelo oráculo, basta seguir para o próxima etapa de execução.

Para cada mutante, é aplicado o conjunto de teste original. Se a execução do caso de teste no mutante m resulta em uma saída diferente de P, isso significa que os casos de teste T conseguiram expor a diferença entre P e m (Ammann e Offutt, 2016, Capítulo 9):

Definição 2.1.10 (Mutante Morto)**.** *Quando a diferença entre P e m é evidenciada através de um caso de teste, considera-se m morto.*

Em certas situações, não é possível matar um mutante porque este, possui o mesmo comportamento do programa original, o que faz com que ambos não possam ser distinguidos por nenhum caso de teste. Neste caso, o mutante é dito equivalente (Souza Neto, 2019).

Definição 2.1.11 (Mutante Equivalente)**.** *Um mutante m para o qual não existe um caso de teste capaz de distingui-lo de P.*

Após a execução dos mutantes, é possível mensurar a eficiência do conjunto de teste utilizado. A sobrevivência de mutantes pode indicar fraqueza no conjunto de teste. Segundo Delamaro et al. (2017), dado um programa P e o conjunto de casos de teste T, calcula-se o escore de mutação $MS(P,T)$ da seguinte maneira:

Definição 2.1.12 (Escore de Mutação)**.** $MS(P,T) = \dfrac{DM(P,T)}{M(P) - EM(P)}$*, onde:*

- *DM(P, T) = número de mutantes mortos pelo conjunto T;*
- *M(P) = número total de mutantes gerados para P;*
- *EM(P) = número de mutantes gerados que são equivalentes a P.*

A partir do escore de mutação atingido, cabe ao testador decidir se a atividade de teste continua ou não. Quanto mais próximo o escore for de 1 melhor será o conjunto de teste T. Caso o valor de MS não seja satisfatório é necessário fazer melhorias no conjunto de teste e repetir o processo de execução do programa P com o conjunto de mutantes M.

2.2 APLICAÇÕES MÓVEIS

Nos últimos anos aplicações móveis se tornaram indispensáveis. Tarefas como: mandar e-mail, tirar foto, assistir um vídeo, relacionar-se com amigos e acessar algum conteúdo na Internet, estão intrinsecamente ligadas ao cotidiano das pessoas. Dada a gama de opções de uso que um aparelho móvel pode ter é natural pensar que este objeto é praticamente inseparável da maioria das pessoas (Lecheta, 2015, Capítulo 1). Segundo Deng et al. (2015), uma aplicação móvel pode ser definida da seguinte forma:

Definição 2.2.1 (Aplicação Móvel)**.** *É um programa de software que executa em um dispositivo móvel como um smartphone ou tablet.*

O aumento do uso de dispositivos móveis e da complexidade das aplicações trouxe novos desafios para os pesquisadores de Engenharia de Software (Salihu et al., 2019). Wasserman (2010) enumera os requisitos adicionais que se têm ao se desenvolver software para dispositivos móveis:

1. **Potencial de interação com outras aplicações**. A maioria dos dispositivos embutidos apenas possuem o software instalado de fábrica, mas os dispositivos móveis podem ter inúmeras aplicações de fontes variadas, podendo existir interação entre elas.

2. **Manipulação de sensores**. A maioria dos *smartphones* já inclui diferentes sensores: o acelerômetro, tela sensível ao toque, teclados reais e/ou virtuais, Sistema de Posicionamento Global (GPS), microfone, câmeras e vários protocolos de rede.

3. **Aplicações nativas e híbridas (web móvel)**. Ao desenvolver uma aplicação móvel considera-se que a instalação ocorre localmente no dispositivo móvel, mas as diferentes funcionalidades presentes na aplicação podem consumir dados e serviços da Internet ou da telefonia.

4. **Famílias de plataformas de *hardware* e *software***. Um desenvolvedor Android precisa decidir se deve construir um único aplicativo ou várias versões para executar nas diferentes versões de sistema operacional e de dispositivos Android.

5. **Segurança**. Instalação constante de diferentes aplicações móveis e a conexão com redes externas apresentam lacunas de segurança que podem ser exploradas em possíveis ataques.

6. **Telas de usuário**. Uma aplicação móvel deve compartilhar elementos comuns da interface do usuário com outras aplicações e deve aderir às diretrizes de interface de usuário desenvolvidas externamente.

7. **Complexidade no teste**. As aplicações web móveis são particularmente desafiadoras para o teste, pois além de possuírem os mesmos problemas encontrados no teste de aplicações da Web elas têm os problemas associados à transmissão através da rede telefônica.

8. **Consumo de energia**. O comportamento da aplicação móvel pode influenciar no nível de consumo de energia do dispositivo. Sendo assim, a utilização de diferentes recursos do dispositivo móvel interferem na autonomia. Lecheta (2015, Capítulo 22) adiciona que quanto mais novo o aparelho, mais sensores ele terá, ou seja, novas funcionalidades irão consumir a energia do dispositivo.

O sistema operacional Android domina o mercado junto com iPhone Operation System (iOS). No segundo quarto de 2020, os usuários fizeram *download* de 28,7 bilhões de aplicações Android e 9,1 bilhões iOS (Iqbal, 2020). O presente trabalho possui como foco aplicações Android, além de este sistema ser amplamente utilizado, ele é uma plataforma de código-aberto. Sendo assim é de interesse entender melhor sobre o sistema Android, assunto da próxima seção.

2.2.1 Plataforma Android

A Google lançou o Android em novembro de 2007, por meio do grupo Open Handset Alliance (OHA). O Android é um sistema operacional móvel de código aberto baseado no kernel do Linux e facilita os desenvolvedores a escreverem código gerenciado em Java usando as bibliotecas desenvolvidas pela própria Google (Grønli et al., 2014). O grupo OHA, composto por gigantes do mercado de telefonia de celulares, tem como propósito definir uma plataforma única e aberta. Os fabricantes podem customizar o sistema operacional Android sem precisar compartilhar o código-fonte, uma vez que sua licença baseia-se na Apache Software Foundation (ASF) (Lecheta, 2015, Capítulo 2).

O grau de abertura da plataforma estimula a rápida inovação. Ao contrário do iOS patenteado pela Apple, que só existe para dispositivos Apple, o Android está disponível em aparelhos de dezenas de fabricantes de equipamento original e em numerosas operadoras de

telecomunicações em todo o mundo. A intensa concorrência entre as fabricantes e as operadoras beneficia os consumidores (Deitel et al., 2015).

A pilha de camadas Android é composta por uma grande quantidade de características. De acordo com Ableson et al. (2012, Capítulo 1), as camadas podem ser separadas em:

- ***Kernel* Linux**. Um *kernel* Linux que fornece uma camada básica de abstração de *hardware*, além de serviços principais, como gerenciamento de processos, memória e sistema de arquivos. O *kernel* é onde os *drivers* específicos de *hardware* são implementados - recursos como Wi-Fi e *Bluetooth*. A pilha do Android foi projetada para ser flexível, com muitos componentes opcionais que dependem amplamente da disponibilidade de *hardware* específico em um determinado dispositivo. Esses componentes incluem recursos como telas sensíveis ao toque, câmeras, receptores GPS e acelerômetros.

- **Bibliotecas de código**. Tecnologias como navegador WebKit, banco de dados SQLite, suporte de áudio e vídeo com OpenCORE, entre outras.

- **Matriz de gerentes que fornecem serviços**. Os serviços podem ser: *Activities*, Janelas, Recursos, Serviço baseados em localização, Telefone, entre outros.

- **Android Runtime**. Para dispositivos com Android versão 5.0 (API nível 21) ou mais recente, cada aplicação executa o próprio processo com uma instância própria do Android Runtime (ART). O ART é projetado para executar várias máquinas virtuais em dispositivos de baixa memória executando arquivos DEX (Dalvik *Executable*)[1], um formato de *bytecode* projetado especialmente para Android, otimizado para oferecer consumo mínimo de memória, construir cadeias de ferramentas, e compilar fontes Java em *bytecode* DEX, que podem ser executados na plataforma Android (Google, 2019a).

Além de entender as camadas que compõem o sistema Android, é importante ter conhecimento sobre alguns elementos que pertencem ao cenário de desenvolvimento de aplicações Android. Dentro de cada aplicação, especificamente declarados no arquivo AndroidManifest.xml, existem quatro principais componentes: (i) *Activity*; (ii) *Service*; (iii) *BroadcastReceiver*; e (iv) *ContentProvider*. A classe *Activity* está relacionada a uma tela visível da aplicação móvel. Aplicações Android podem conter mais de uma *Activity* e cada uma é composta por elementos de interface de usuário (Ableson et al., 2012, Capítulo 1). A maioria dos elementos de GUI (*Graphical User Interface*) utilizados na *Activity* estão no pacote *widget*. A classe *Service* é para tarefas que serão executadas em segundo plano e não estarão visíveis na tela. Por exemplo, uma aplicação móvel de reprodução de música provavelmente seria implementada como um serviço para continuar a reproduzir músicas enquanto um usuário pode estar visualizando páginas da web (Mednieks et al., 2012). O componente *BroadcastReceiver* é utilizado para responder determinados eventos enviados por uma *Intent*, como executar determinada aplicação ao receber uma mensagem SMS, uma ligação, ou qualquer outra ação customizada (Lecheta, 2015, Capítulo 9). A classe *ContentProvider* é necessária para que a aplicação móvel transfira informação para outra aplicação no ambiente Android (Ableson et al., 2012, Capítulo 1).

2.3 CONSIDERAÇÕES FINAIS

Neste capítulo, foram apresentados os tópicos essenciais para contextualizar o trabalho desenvolvido nesta dissertação. Os conceitos de teste de software e teste de mutação foram

[1]Extensão do arquivo executável entendido pela máquina virtual Dalvik, do Android.

abordados, bem como as principais características trazidas pelo contexto de aplicações móveis da plataforma Android.

No próximo capítulo, apresentam-se as pesquisas na área de teste de mutação de aplicações móveis Android e também a vertente de verificação de acessibilidade, descrevendo-se os principais trabalhos relacionados ao tema desta dissertação.

3 TRABALHOS RELACIONADOS

Este capítulo expõe trabalhos que abordam os principais temas relacionados a esta dissertação. Na Seção 3.1, apresentam-se trabalhos sobre teste de acessibilidade em aplicações Android. Em seguida, na Seção 3.2 encontram-se pesquisas a fim de propor estratégias para geração de mutantes no teste de aplicações Android.

3.1 TESTE DE ACESSIBILIDADE PARA APLICAÇÕES ANDROID

Existem poucos trabalhos na literatura sobre como avaliar a acessibilidade de aplicações móveis. Segundo Acosta-Vargas et al. (2020), esta pouca quantidade de estudos é atribuída à falta de ferramentas adequadas, guias, e políticas para avaliar as aplicações, e ainda, os poucos guias que existem são ignorados com frequência pelos desenvolvedores (Eler et al., 2018). Os principais trabalhos de acessibilidade de aplicações Android da literatura são abordados a seguir.

O trabalho de Machado et al. (2014) investiga as recomendações de acessibilidade propostas pela Google para o desenvolvimento de aplicações Android em *tablets*, na perspectiva de usuários com deficiência visual. As pesquisas de Acosta-Vargas et al. (2019) e Acosta-Vargas et al. (2020) apresentam um experimento em que o guia de acessibilidade WCAG 2.1 (Web Content Accessibility Guidelines 2.1) e a aplicação Accessibility Scanner (Google, 2019d) são utilizados de forma manual. O objetivo de ambos os trabalhos é avaliar e sugerir melhorias para que as aplicações se tornem mais acessíveis. O guia WCAG cobre um vasto conjunto de recomendações para tornar o conteúdo Web mais acessível. Em sua versão mais recente, as sugestões levam em consideração o acesso web feito via dispositivo móvel. Pela definição do guia (Kirkpatrick et al., 2018), os critérios de sucesso são divididos em quatro tipos: (i) Perceptível; (ii) Operável; (iii) Entendível; (iv) Robusto. A Tabela 3.1 relaciona os princípios com sua respectiva descrição.

Tabela 3.1: Barreiras relacionadas ao WCAG 2.1.

Princípio	**Descrição**
Perceptível	O usuário deve ser capaz de distinguir o conteúdo de forma visual, tátil e sonora
Operável	O usuário deve navegar de forma clara e apropria usando os elementos de GUI
Entendível	Como a interface controla o conteúdo para o usuário entender e gerenciar
Robusto	O conteúdo deve se adequar e permanecer manuseável

De acordo com Ballantyne et al. (2018), enquanto alguns trabalhos replicam o guia WCAG, outros pesquisadores trazem inúmeros requisitos que não são cobertos por ele. Os autores compilam um superconjunto de guias. O resultado elenca 11 categorias de elementos testáveis de acessibilidade, são eles: Texto, Áudio, Video, Elementos de GUI, Controle de Usuário, Flexibilidade e Eficiência, Reconhecimento ao invés de Recordar, Gestos, Visibilidade do Sistema, Prevenção de Erro e Interação Tangível.

Em um mapeamento similar, Damaceno et al. (2018) identificaram 68 problemas associados a diferentes aspectos de interação de pessoas com limitação visual em dispositivos móveis. Estes problemas foram mapeados em 7 grupos: Botões, Entrada de Dados, Interação baseada em gesto, Tamanho de tela, Retorno do usuário e Comando por voz. O grupo com o maior número de problemas foi o grupo de defeitos relativo à interação feita de formal gestual.

Vendome et al. (2019) apresentam uma taxonomia de problemas de acessibilidade. A classificação é construída a partir da mineração de 13.817 aplicações Android do GitHub. As

aplicações analisadas tinham em seus repositórios: (i) APIs de acessibilidade; e (ii) presença/falta de conteúdo descritivo em seus elementos de GUI. Os autores observaram que 36,96% dos projetos não possuíam elementos com atributos descritivos de rótulo e ainda, apenas 2,08% importou pelo menos uma API de acessibilidade. As principais categorias elencadas na taxonomia foram: Suporte para limitação visual, Suporte para limitação motora, Limitação auditiva e outros aspectos de acessibilidade e Limitação não específica.

Em um trabalho mais recente, Alshayban et al. (2020) apresentam resultados de um estudo de grande escala para entender a acessibilidade a partir de três perspectivas complementares: Aplicações, Desenvolvedores e Usuários. Primeiro, analisam a prevalência de problemas de acessibilidade em mais de 1000 aplicações Android. Em seguida, investigam os sentimentos do desenvolvedor por meio de uma pesquisa. Por fim, verificam as avaliações dos usuários e a popularidade da aplicação. A análise revelou: os desenvolvedores geralmente desconhecem os princípios de acessibilidade; as ferramentas de análise existentes não são suficientemente sofisticadas para serem úteis (por exemplo, não conseguem priorizar questões de acessibilidade); as taxas de problemas de acessibilidades de aplicações desenvolvidas por grandes empresas são relativamente semelhantes às de outras aplicações.

O estudo de Eler et al. (2018) definiu um conjunto de critérios de acessibilidade, e apresentou a MATE (*Mobile Accessibility TEsting*), uma ferramenta que de forma automática explora e verifica a acessibilidade da aplicação móvel. A MATE preenche uma lacuna importante da literatura, pois até então os testes de acessibilidade eram feitos através dos guias mencionados acima ou por ferramenta de checagem estática. Tal checagem estática pode ser feita pela ferramenta Lint (Google, 2019c), mas a natureza dinâmica da plataforma Android faz com que algumas propriedades passem despercebidas. As checagens de acessibilidade feitas pela MATE estão separadas nas propriedades elencadas pela Tabela 3.2.

Tabela 3.2: Propriedades checadas pela MATE.

Propriedade	**Descrição**
Texto pronunciável	Os elementos de interação da tela devem possuir rótulos descritivos e únicos
Relação de contraste	Taxa de contraste entre o primeiro plano e o segundo plano é de pelo menos: (i) 3 para componentes visuais maiores (ii) 4,5 para componentes pequenos
Área de toque	A área de toque de um elemento da tela deve ter no mínio 48x48dp
Limite de toque duplicado	Dois elementos de tela clicáveis não podem dividir a mesma área
Spam clicável	Palavras ou expressões clicáveis não podem ser acionadas pelos serviços de acessibilidade

Os desenvolvedores também podem avaliar manualmente as propriedades de acessibilidade pela aplicação Accessibility Scanner (Google, 2019d). Ela permite testar as aplicações e obter sugestões sobre como melhorar e aprimorar a acessibilidade (para ajudar quem tem limitações de visão, fala ou movimentação). A Figura 3.1 mostra a utilização da Accessibility Scanner. Primeiro a aplicação é ativada, em seguida exibe as principais instruções de manuseio. Por fim, com a aplicação móvel em execução, Accessibility Scanner realça na tela o elemento de GUI e qual propriedade de acessibilidade ele não está cumprindo. No caso da figura, a área de toque em destaque não possui o tamanho recomendado.

A aplicação A11y, desenvolvida por Toff (2019), realiza de forma manual ou automática a checagem de acessibilidade da aplicação móvel em execução. A Figura 3.2 mostra a tela de configuração em que é possível observar os problemas de acessibilidade checados pela a11y. A partir de sua integração pela linha de comando, a11y gera no final de sua execução um arquivo JSON. Este arquivo contém a relação dos elementos de GUI e qual critério de acessibilidade foi violado.

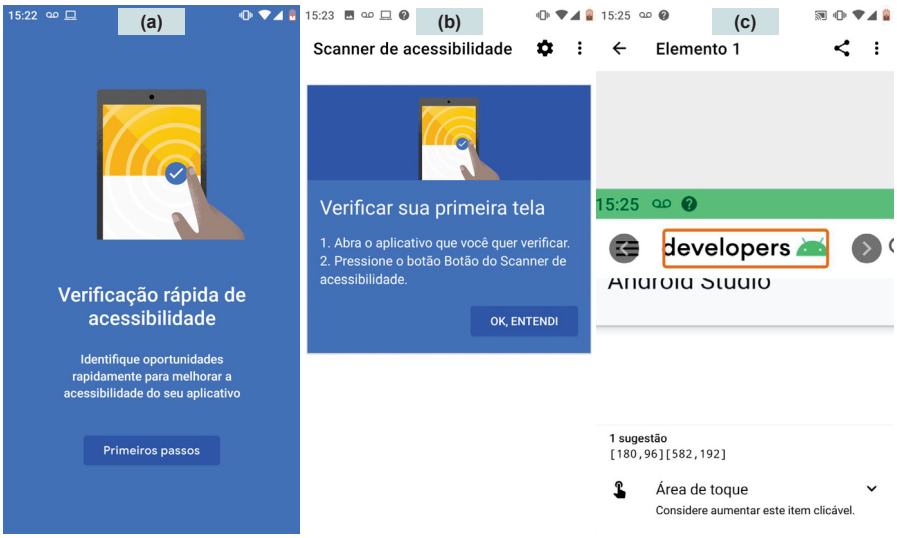


Figura 3.1: Fluxo de execução Accessibility Scanner.

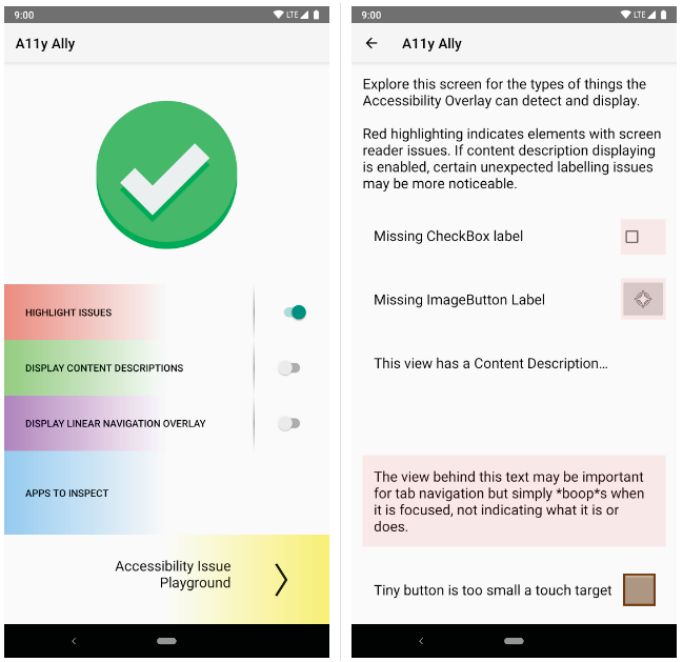


Figura 3.2: Tela de configuração da a11y.

O *framework* Espresso (Google, 2019b) permite a gravação de testes automatizados que avaliam a acessibilidade da aplicação móvel. A acessibilidade do elemento da GUI, ou simplesmente *widget*, será verificada caso a ação acione/interaja com o *widget* em questão. O Código 3.1 mostra como é a saída do *log* após a execução de um conjunto de teste implementado com Espresso. No caso, o *widget* clicável não apresentava área mínima de toque de 48x48dp.

Código 3.1: Exemplo de falha de acessibilidade no *log* após execução de ação implementada com Espresso.

```
AppCompatTextView{id=2131296368...} View falls below the minimum
    recommended size for touch targets. Minimum touch target size is 48x48dp.
     Actual size is 198.9x24.0dp (screen density is 2.6).
...
```

A Tabela 3.3 sumariza as ferramentas descritas nesta seção. Cada ferramenta elencada verifica critérios de acessibilidade durante a execução da aplicação Android. Ao apontar a necessidade destes critérios, não só a interação aplicação e usuário tem influência, mas também a interação da aplicação móvel com os recursos nativos Android de suporte à acessibilidade. O recurso TalkBack[1] é um dos mais famosos e utilizados; ele permite que usuários com limitações visuais possam interagir com o *smartphone* somente ouvindo o *feedback* de áudio emitido ao interagir com qualquer elemento da tela. Toda vez que um elemento é focado pelo TalkBack, um som é emitido descrevendo o componente seguindo a sintaxe: Estado + Texto do conteúdo componente + Tipo de componente.

Tabela 3.3: Critérios de acessibilidades avaliados pelas ferramentas MATE, Espresso, Accessibility Scanner e a11y.

	MATE	Espresso	Accessibility Scanner	a11y
Área de toque pequena	✓	✓	✓	✓
Ausência de rótulos	✓	✓	✓	✓
Contraste de texto	-	✓	✓	-
Contraste de imagem	✓	-	✓	-
Texto pronunciável duplicado	✓	✓	-	-
Descrição de conteúdo redundante	-	✓	-	-

Observa-se na Tabela 3.3 que a ferramenta Espresso é a que verifica um maior número de critérios entre todas as ferramentas.

3.2 TESTE DE MUTAÇÃO PARA APLICAÇÕES ANDROID

Para esta dissertação realizou-se um mapeamento sistemático da literatura sobre as técnicas de teste de mutação para aplicações Android. O objetivo do mapeamento realizado é prover informações sobre as principais características dos estudos, são elas: foco do estudo; tipo do estudo; classe de operador de mutação; automatização do processo do teste de mutação. Após aplicar os critérios de seleção, 16 estudos primários foram analisados. O primeiro estudo foi publicado no ano de 2015. 15 estudos (98%) foram publicados no período 2017-2020. Jeff Offut e Lin Deng, com 5 estudos sobre o tema, foram os pesquisadores que mais contribuíram até o momento. Dentro dos estudos analisados, 8 definem um conjunto de operador de mutação para Android. Ao todo 138 operadores foram definidos, eles podem ser classificados em: Configuração, Conectividade, GUI, *Intent*, Localização, Persistência, Sensor e Tradicional. Além da definição de operadores, alguns estudos propõem ferramentas que automatizam o processo de mutação. Apenas 3 estudos apresentam um ferramenta que automatiza todos os processos: Geração de Mutante, Execução do Mutante e Análise de Mutante. Mais detalhes do processo de mapeamento, resultados e oportunidades encontradas podem ser consultados no Apêndice A.

A partir dos 16 estudos investigados, percebe-se a existência de poucas propostas de abordagem para o teste de mutação de aplicações Android. O trabalho de Deng et al. (2015) define 4 classes de operadores de mutação específicas para o contexto Android: *Intent Mutation Operators*, *Event Handler Mutation Operators*, *Activity Lifecycle Mutation Operator* e *XML Mutation Operator*. O fluxo de trabalho proposto, representado na Figura 3.3, difere do processo tradicional do teste de mutação. Uma vez que os mutantes são gerados, é necessário realizar a instalação de cada mutante *m* no emulador Android. O conjunto de teste é implementado

[1]Ativação do recurso TalkBack: `https://support.google.com/accessibility/android/answer/6007100?hl=pt-BR`.

através dos *frameworks* Robotium (Reda, 2019) ou JUnit (Gamma e Beck, 2019). Depois da implementação, o conjunto de teste é executado no emulador e sua saída registrada para calcular o escore de mutação. O protótipo da ferramenta muDroid$_{\text{Deng}}$[2] não se encontra disponível.

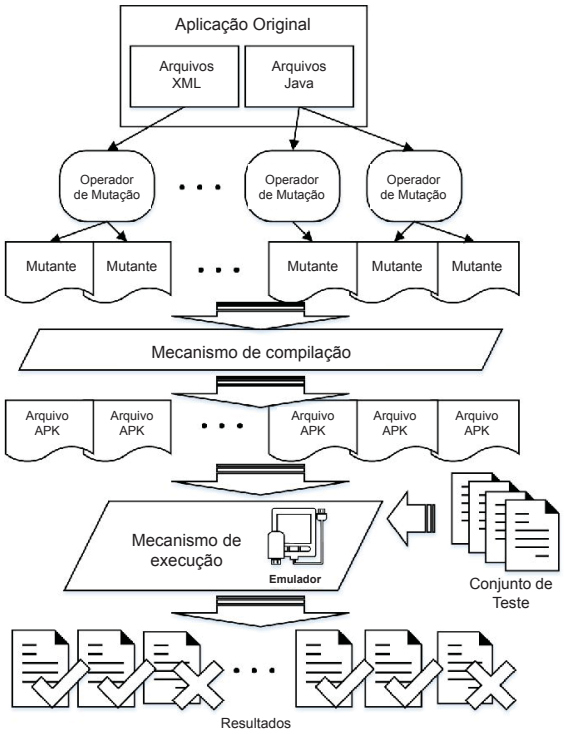


Figura 3.3: Análise de mutantes em aplicações Android; adaptado de Deng et al. (2015).

O processo de mutação da abordagem de Deng et al. (2015) depende do acesso ao código-fonte da aplicação móvel. Quando uma aplicação não é de código aberto, o testador tem acesso apenas ao seu executável, ou simplesmente arquivo na extensão Android Package (APK). Dada essa limitação, Wei (2015) propõe a ferramenta muDroid$_{\text{Wei}}$. Para sua entrada, fornece-se o executável da aplicação original. O fluxo de trabalho da ferramenta é representado pela Figura 3.4. Apesar da ferramenta muDroid$_{\text{Wei}}$ apresentar uma abordagem interessante, ela possui certas limitações. Na etapa de "Simulação de Interação", apenas eventos de clique em coordenadas são gerados. Se a ferramenta tiver como entrada uma aplicação que exige entrada de texto ou interações mais complexas, os casos de testes da ferramenta (representados por eventos de clique na tela) podem impactar o escore de mutação. Além disso, durante o processo de geração, os operadores de mutação utilizados estão relacionados a apenas defeitos clássicos de Java, não explorando aspectos específicos de aplicações Android.

A pesquisa de Linares-Vásquez et al. (2017), por meio de uma taxonomia de defeitos, buscou identificar os possíveis defeitos de implementação que uma aplicação Android pode ter. Eles investigaram seis fontes de informação: (i) Relatório de *Bug* de aplicações Android de código aberto; (ii) *Commit* de correção de *Bug*; (iii) Discussões na página *Stack Overflow*; (iv) Principais defeitos de lançamento de exceção elencados por Zhang e Elbaum (2012) e Coelho et al. (2015); (v) *Crashes/bugs* descritos em outros estudos de aplicações Android; e (vi) Revisões postadas por usuários de Android na página Google Play.

Consequentemente, por meio da taxonomia de defeitos Android, Linares-Vásquez et al. (2017) definiram uma lista de 38 operadores de mutação. Estes operadores foram implementados

[2]Os autores apenas nomeiam a abordagem em Deng e Offutt (2018).

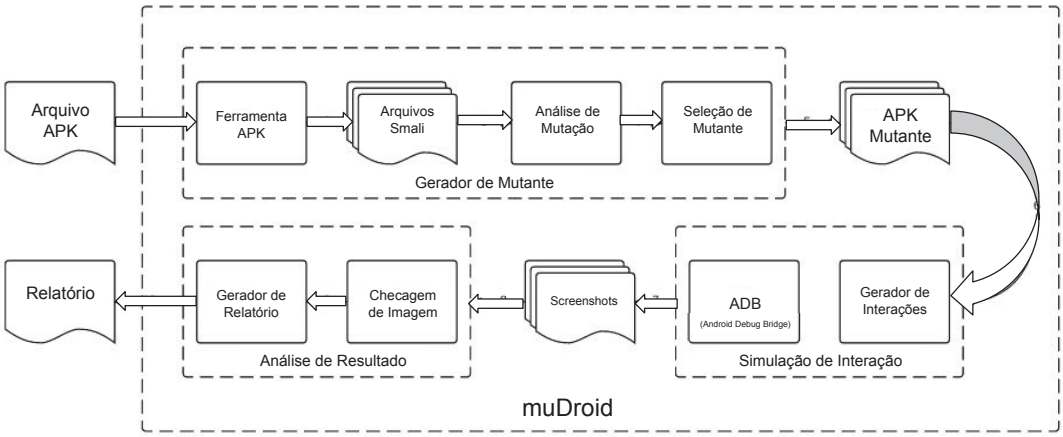


Figura 3.4: Diagrama do sistema muDroid; adaptado de Wei (2015).

pela ferramenta MDroid+ (Moran et al., 2018). O fluxo da MDroid+ (Figura 3.5) funciona da seguinte maneira. Primeiro, uma análise estática do código Java usando *Abstract Syntactic Trees* (AST) é realizada para encontrar um *Potential Fault Profile* (PFP) que descreve um local do código-fonte que pode ser alterado por um operador. Os PFPs são usados para aplicar a transformação correspondente a cada operador no código Java ou arquivo XML (representa as interfaces de usuário ou configurações da Aplicação). A MDroid+ cria um clone do projeto Android e aplica uma única mutação a um PFP especificado no projeto clonado, resultando em um projeto mutante para cada instância semeada de um operador de mutação. Por fim, um relatório é gerado associando o nome do clone criado com o operador de mutação nele aplicado. A ferramenta precisa ter como entrada o código-fonte da aplicação Android. Ela também não oferece uma maneira de compilar e executar os mutantes, nem calcular o escore de mutação.

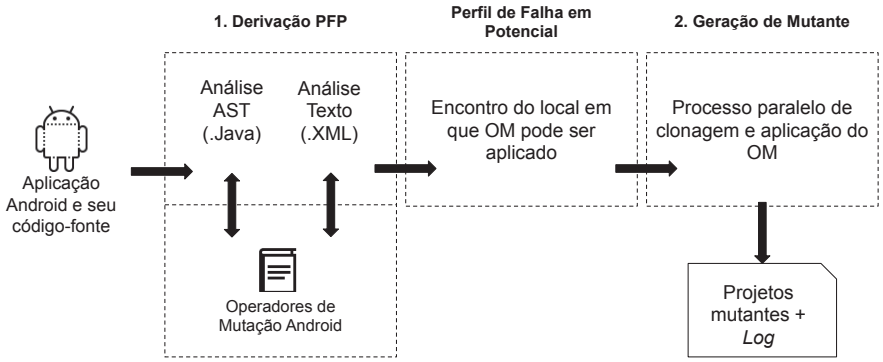


Figura 3.5: Fluxo de trabalho da MDroid+; adaptado de Moran et al. (2018).

Alguns trabalhos exploram aspectos de natureza específica dentro da plataforma Android. Luna e El Ariss (2018) propuseram Edroid, uma ferramenta que usa 10 operadores de mutação orientados para variar arquivos de configuração e elementos de interface do usuário (GUI). Para avaliar o uso da ferramenta, os autores conduziram um experimento com 5 aplicações móveis Android e 3 tipos de teste de geração de casos de teste: Teste Aleatório (com uso da ferramenta Monkey[3]), Teste Adhoc e Teste Caixa-preta. A análise do mutante foi feita de forma manual, caso

[3]Disponível em `https://developer.android.com/studio/test/monkey`.

os componentes da interface do mutante se distinguissem do original, o mutante era classificado como morto.

Jabbarvand e Malek (2017) implementam µDroid, uma ferramenta de teste de mutação orientada para identificar problemas relacionados a energia. A metodologia dos pesquisadores foi criar um modelo de defeitos a partir dos anti-padrões encontrados em Android Issue Tracker, fóruns e buscas por comentários com as palavras: *energy*, *power*, *battery*, *drain* e *consumption*. Os autores implementaram um total de 50 operadores de mutação que correspondem a 28 classes definidas como anti-padrão de consumo de energia. A ferramenta µDroid conta com um processo de teste de mutação totalmente automatizado. Enquanto o conjunto de teste *T* é executado na aplicação original, o gasto de energia é monitorado. Em seguida, no momento em que *T* é executado no mutante, compara-se o rastreio do gasto de energia da aplicação original com o do mutante. Caso o rastreio seja diferente o suficiente, o mutante é considerado morto. Seu fluxo de trabalho é representado pela Figura 3.6.

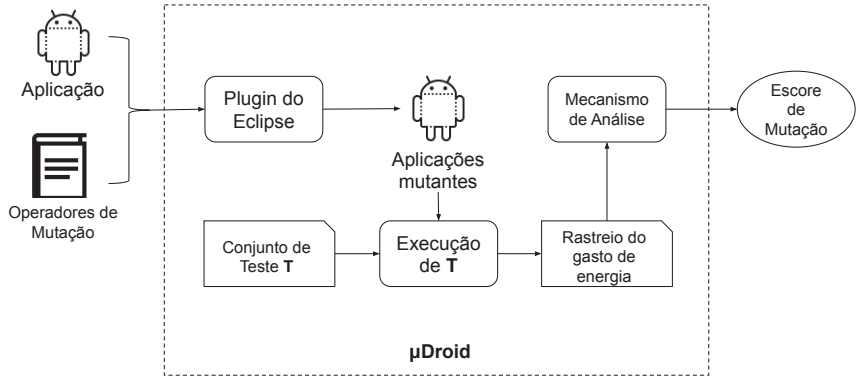


Figura 3.6: Fluxo de trabalho da µDroid; adaptado de Jabbarvand e Malek (2017).

Escobar-Velásquez e Linares-Vásquez (2019) introduziram MutAPK, uma ferramenta de código aberto que habilita o teste de mutação no nível de execução, ou seja, como entrada da ferramenta apenas o APK da aplicação Android é necessário. Os operadores implementados foram os mesmos propostos por Linares-Vásquez et al. (2017). Para utilizar os operadores da ferramenta MDroid+, eles traduziram a lógica da implementação original, que trabalhava com regras baseadas em código ou texto. A implementação correspondente considera a representação SMALI. O arquivo SMALI é criado no processo de descompilação dos arquivos DEX (Dalvik *Executable*). Para se chegar nesta representação SMALI, utilizaram o desempacotador APKTool[4].

Assim como MDroid+, a ferramenta MutAPK não inclui em seu fluxo de execução uma estratégia de análise de mutantes implementada (Figura 3.7). Ambas permitem a criação de um operador de mutação customizado.

Em um trabalho mais recente, Liu et al. (2020) apresentam DroidMutator. DroidMutator utiliza checagem de tipo para reduzir geração de mutantes natimortos[5]. DroidMutator implementa 32 operadores de mutação, sendo 4 específicos da linguagem Java e 28 da plataforma Android. O fluxo de execução (Figura 3.8) tem como entrada a aplicação Android junto com seu código-fonte.

Não foram definidos operadores de mutação de contexto XML. Dentro do estudo experimental, com 50 aplicações Android, DroidMutator gerou 48.703 mutantes e apenas 56 mutantes natimortos. A ferramenta não inclui processo de análise de mutante.

[4]Disponível em `https://ibotpeaches.github.io/Apktool/`.

[5]Mutantes que não puderam ser compilados (Deng et al., 2015).

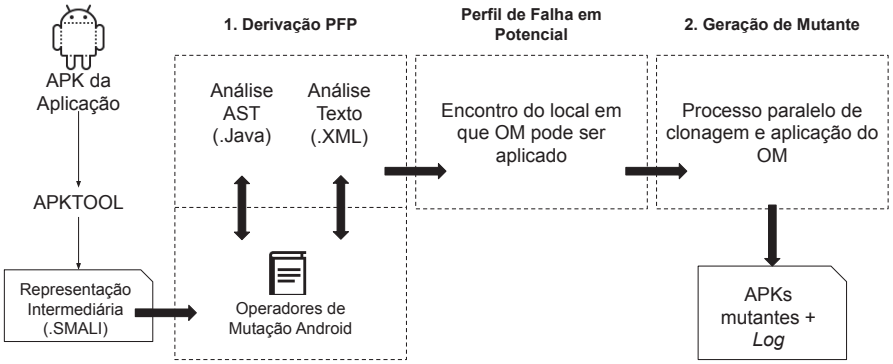


Figura 3.7: Fluxo de execução da MutAPK; adaptado de Escobar-Velásquez e Linares-Vásquez (2019).

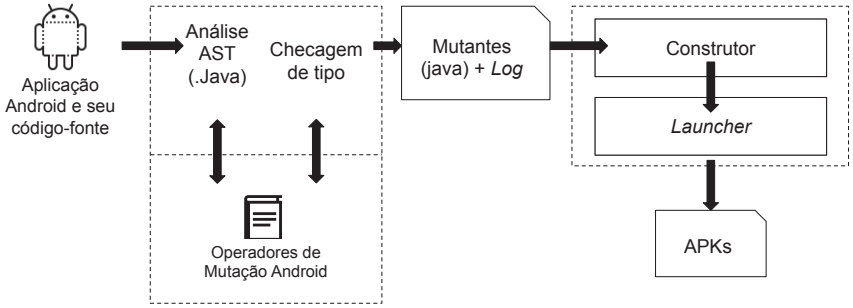


Figura 3.8: Fluxo de execução da DroidMutator; adaptado de Liu et al. (2020).

A Tabela 3.4 sumariza as principais características das ferramentas apresentadas neste capítulo. Note que não há uma ferramenta "superior" em todos os aspectos. Observa-se que a ferramenta MDroid+, MutAPK e DroidMutator possuem uma representatividade maior no universo de defeitos da plataforma Android. A primeira coluna, representada por "Geração de *T*", indica se a própria ferramenta de mutação gera de forma automática um conjunto de teste *T* para ser utilizado no processo de análise de mutantes. Se *T* é gerado por outro mecanismo o símbolo "-" está incluso na célula referente. A coluna "OM Android", indica se operadores de mutação da plataforma Android são explorados. A terceira coluna aponta se há algum mecanismo implementado para realizar a análise do mutante e consequentemente obter seu escore de mutação. A penúltima coluna "Item de entrada" evidencia se a ferramenta consegue trabalhar com a aplicação Android sem a necessidade do código-fonte. Por fim, a última coluna "Ferramenta disponível", indica se a ferramenta tem seu código disponível em algum repositório.

Tabela 3.4: As principais ferramentas de teste de mutação Android.

	Geração de *T*	**OM Android**	**Análise de Mutante**	**Item de entrada**	**Ferramenta disponível**
DroidMutator	-	✓	-	Código-fonte	✓
muDroid$_{Deng}$	-	✓	✓	Código-fonte	-
muDroid$_{Wei}$	✓	-	✓	APK	✓
MDroid+	-	✓	-	Código-fonte	✓
Edroid	-	✓	-	Código-fonte	-
µDroid	-	✓	✓	Código-fonte	-
MutAPK	-	✓	-	APK	✓

Vale a pena observar que nenhuma das ferramentas sugere operadores de mutação que explorem defeitos de acessibilidade de uma aplicação móvel Android e apenas duas ferramentas

implementaram um processo automático de teste capaz de realizar análise de mutante e calcular escore de mutação.

3.3 CONSIDERAÇÕES FINAIS

Neste capítulo, foram apresentados os resultados de um levantamento que abordou dois temas: (i) teste de acessibilidade em aplicações Android e (ii) mutação em aplicações Android. A Seção 3.1 sumarizou as principais limitações de teste de acessibilidade para a plataforma Android. Os trabalhos apresentados sobre mutação na Seção 3.2 mostraram possíveis fluxos de análise de mutante.

Esta dissertação define operadores de mutação que exploram defeitos de acessibilidade em uma aplicação Android. Os operadores de acessibilidades são implementados por meio de uma extensão da ferramenta MDroid+. Para a produção de relatório de acessibilidade, optou-se pela ferramenta Espresso. A análise de mutantes depende diretamente dos critérios de acessibilidade avaliados e expostos no *log* de cada ferramenta. A ferramenta MATE produz um relatório de acessibilidade interessante, mas como este é gerado a partir de ações aleatórias, não faria sentido determinar a morte do mutante através da comparação do *log* de acessibilidade da aplicação original com seus mutantes. Seria necessário alterar o código-fonte de MATE, para que ela salve ou reproduza os mesmos eventos utilizados na etapa de exploração da aplicação. A aplicação Accessibility Scanner avalia uma quantidade satisfatória de critérios de acessibilidade, mas não fornece uma forma de acessá-la e executá-la por linha de comando, o que dificultaria sua integração com a abordagem. A ferramenta a11y prevê o uso por linha de comando, entretanto avalia apenas dois critérios de acessibilidade.

Além da Espresso checar um conjunto maior de critérios de acessibilidade, seu processo de avaliação depende da interação do conjunto de teste com algum elemento de GUI da aplicação móvel. Dessa forma, os problemas de acessibilidade apontados dependem diretamente da qualidade do conjunto de testes. Ao propor operadores de mutação de acessibilidade e sua respectiva análise torna-se possível: (i) avaliar ferramentas de geração de conjunto de teste sob uma perspectiva de acessibilidade; (ii) avaliar ferramentas automatizadas de verificação de acessibilidade; e (iii) avaliar se a aplicação móvel Android possui atributos de acessibilidade em seus elementos de GUI.

O próximo capítulo apresenta a criação do modelo de defeitos; a definição dos operadores de mutação para capturar falhas de acessibilidade; e detalhes da implementação da abordagem.

4 ABORDAGEM

Com base no conteúdo exposto e trabalhos descritos no capítulo anterior, o presente trabalho propõe uma abordagem de teste de mutação para avaliar a acessibilidade de aplicações Android, chamada `AccessibilityMDroid`. Este capítulo descreve a abordagem e está organizado da seguinte maneira: a Seção 4.1, descreve como foi gerado o modelo de defeitos de acessibilidade, que serviu como base para a definição dos operadores de mutação descritos na Seção 4.2. Na Seção 4.3 discorre-se sobre como a análise de mutantes é realizada; na Seção 4.4 estão os detalhes mais práticos da ferramenta `AccessibilityMDroid`.

4.1 MODELO DE DEFEITOS

Um guia de acessibilidade, de maneira geral, sumariza as principais recomendações para tornar o conteúdo presente da tela da aplicação móvel mais acessível. A partir dos trabalhos analisados na Seção 3.2, adotou-se como referência o guia WCAG. Este guia foi escolhido por possuir critérios de sucesso escritos como declarações testáveis, e ainda, uma nova versão do guia mantém conformidade com sua versão anterior. O guia WCAG 2.1 foi aplicado por Acosta-Vargas et al. (2020). Durante este experimento, os autores relacionam possíveis critérios de sucesso que atendem aos quatro princípios do guia WCAG. Ao conhecer os critérios de sucesso, é possível negá-los e iniciar a construção de um modelo de defeitos. A Tabela 4.1 exemplifica a ideia de negação do critério de sucesso para a construção do modelo de defeitos.

Tabela 4.1: Relação entre princípios e critérios de sucesso do WCAG, sua negação e o *code element* pertinente.

Princípio	**Critério de sucesso**	**Negação do critério**	***Code elements***	
			Atributos XML	Métodos Java
Perceptível	Texto Redimensionável	Texto sem tamanho definido	`:textSize`	`setTextSize`
	Identificação do propósito de entrada	Tipo de entrada não definido	`:inputType`	`setInputType`
	Texto Redimensionável	Texto sem tamanho definido	`:autoSizeTextType`	`setAutoSizeTextTypeWithDefaults`
Operável	Teclado; Ordem de Foco	Ordem de foco não especificada	`:nextFocusDownId`	`setNextFocusDowndId`
	Área de toque mínima	Tamanho mínimo não definido	`:minWidth e :minHeight`	`setMinWidth e setMinHeight`
Entendível	Rótulo ou Instruções	Ausência de rótulo	`:labelFor`	`setLabelFor`
	Rótulo ou Instruções	Ausência de rótulo	`:hint`	`setHint`
Robusto	Mensagens de Status	Ausência de mensagem de status	`:importantForAccessibility`	`setImportantForAccessibility`
	Mensagens de Status	Ausência de mensagem de status	`:accessibilityLiveRegion`	`setAccessibilityLiveRegion`

Uma vez negado cada critério de sucesso, pode-se considerá-lo como um defeito. Tal defeito implica na construção dos operadores de mutação de acessibilidade em aplicações móveis. A negação do critério "Rótulos ou instruções" faz com que se tenha um defeito de ausência de rótulo. Dentro do desenvolvimento móvel para plataforma Android o comportamento dos elementos de interface são definidos por diferentes atributos. Para tais atributos, deu-se o nome de *code elements*. Através da API do Android nota-se que existem diversos *code elements* que caracterizam a utilização de rótulo em um elemento da GUI. Para a negação do critério de sucesso em questão podem-se remover os *code elements*: labelFor e hint. Dessa forma, mais de um operador de mutação pode ser derivado a partir da negação do critério "Rótulos ou instruções" e das demais negações. O mapeamento completo feito a partir dos *code elements* da API Android é representado pela Tabela 4.2. Um total de 55 *code elements* foram elencados. Eles estão organizados pelos princípios WCAG (1ª coluna) e critérios de sucesso (2ª coluna). Um *code element* pode aparecer como um atributo XML (3ª coluna), um método Java (4ª coluna) ou ambos. Por exemplo, o *code element* contentDescription pode ser definido em um atributo XML ou por um método Java. Alguns métodos Java estão relacionados a dois ou mais elementos de

código (por exemplo, setLineSpacing e setAutoSizeTextTypeUniformWithConfiguration); neste caso, eles são diferenciados por seus parâmetros.

Alguns elementos de código podem ser diretamente vinculados a um critério de sucesso. Por exemplo, o *code element* inputType é definido como: "O tipo de dado sendo colocado em um campo de texto, usado para ajudar um método de entrada a decidir como permitir que o usuário insira o texto". Um dos critérios de sucesso do WCAG é chamado "Identificar Propósito de Entrada ", que tem como objetivo verificar se o propósito de cada campo de entrada foi determinado. Por outro lado, alguns elementos do código não estão diretamente ligados a um critério de sucesso. Adotar um determinado *code element* não implica que o aplicação atenda ao critério de sucesso WCAG associado. Os critérios de sucesso podem definir padrões comportamentais que só podem ser atendidos pela combinação de mais de um *code element*. A suposição feita é que o mapeamento resultante apenas sugere que o uso de um *code element* específico pode impactar em um dos critérios de sucesso.

Além do mapeamento entre os *code elements* e os critérios de sucesso, também foi realizado um estudo para investigar: (i) a prevalência de *code elements* de acessibilidade e sua relação com potenciais problemas de acessibilidade; (ii) os problemas de acessibilidade mais comuns nas aplicações Android; e (iii) a correlação entre problemas de acessibilidade e *code elements*, indicando se a presença do último é uma indicação de que a acessibilidade foi considerada no desenvolvimento da aplicação. Ao fim do estudo, foi possível concluir que *code elements* que influenciam a acessibilidade não são amplamente utilizados; apenas um quinto dos *code elements* ocorre em mais de 10% das aplicações. Falhas de acessibilidade são comuns em aplicações Android, encontrou-se um total de 12.108. Todas as aplicações tiveram ao menos 3 falhas de acessibilidade. Ao analisar as correlações, evidenciou-se que as aplicações que adotam diferentes tipos de *code elements* tendem a ter uma densidade menor de falhas de acessibilidade. Mais detalhes do estudo podem ser encontrados em (Silva et al., 2020) e também no Apêndice B.

Tabela 4.2: Mapeamento de *code element* para princípios WCAG e seus critérios de sucesso.

Princípio	**Critério de Sucesso**	***Code Elements*** Atributos XML	Métodos Java
Perceptível	Conteúdo não textual	`:contentDescription`	`setContentDescription`
	Legendas	—	`addCaptioningChangeListener`
	Orientação de tela	`:screenOrientation`	`setScreenOrientation`
	Identificar a finalidade da entrada	`:inputType`	`setInputType`
	Identificar a finalidade da entrada	`:inputMethod`	`--`
	Uso de Cor; Contraste	`:background`	`setBackgroundColor`
	Uso de Cor; Contraste	`:textColor`	`setTextColor`
	Redimensionar texto	`:autoSizeTextType`	`setAutoSizeTextTypeWithDefaults`
	Redimensionar texto	`:textSize`	`setTextSize`
	Redimensionar texto	`:autoSizeMinTextSize`	`setAutoSizeTextTypeUniformWithConfiguration`
	Redimensionar texto	`:autoSizePresetSizes`	`setAutoSizeTextTypeUniformWithPresetSizes`
	Espaçamento de Texto	`:lineSpacingMultiplier`	`setLineSpacing`
	Espaçamento de Texto	`:lineSpacingExtra`	`setLineSpacing`
	Espaçamento de Texto	`:letterSpacing`	`setLetterSpacing`
Operável	Animation from Interactions	`:animateLayoutChanges`	`setLayoutTransition`
	Teclado; Ordem de Foco	`:nextFocusForward`	`setNextFocusForwardId`
	Teclado; Ordem de Foco	`:nextFocusDown`	`setNextFocusDownId`
	Teclado; Ordem de Foco	`:nextFocusRight`	`setNextFocusRightId`
	Título da página	`:accessibilityPaneTitle`	`setAccessibilityPaneTitle`
	Ordem de Foco	`:accessibilityTraversalBefore`	`setAccessibilityTraversalBefore`
	Ordem de Foco	`:accessibilityTraversalAfter`	`setAccessibilityTraversalAfter`
	Cabeçalho e Rótulos	`:accessibilityHeading`	`setAccessibilityHeading`
	Foco Visível	`:cursorVisible`	`setCursorVisible`
	Tamanho Alvo	`:minWidth`	`setMinWidth`
	Tamanho Alvo	`:minHeight`	`setMinHeight`
	Ordem de Foco	`:screenReaderFocusable`	`setScreenReaderFocusable`
Entendível	Rótulo ou Instruções	`:hint`	`setHint`
	Rótulo ou Instruções	`:labelFor`	`setLabelFor`
Robusto	Nome, Papel, Valor	`--`	`onInitializeAccessibilityNodeInfo`
	Nome, Papel, Valor	`--`	`replaceAccessibilityAction`
	Mensagens de Status	`:accessibilityLiveRegion`	`setAccessibilityLiveRegion`
	Mensagens de Status	`:importantForAccessibility`	`setImportantForAccessibility`
	Mensagens de Status	`:hapticFeedbackEnabled`	`setHapticFeedbackEnabled`

4.2 DEFINIÇÃO DOS OPERADORES DE MUTAÇÃO

Operador de mutação de acessibilidade se torna interessante, pois se o mesmo não consegue ser aplicado, já é um indício de que o desenvolvedor não se preocupou em implementar atributos de acessibilidade nos elementos de GUI. Agora, imagine que o desenvolvedor tomou o cuidado em definir os atributos de acessibilidade na aplicação móvel. Mesmo que os atributos estejam definidos é importante garantir que o conjunto de teste execute ações que interajam com o elemento de GUI. Sendo assim, se o *log* de acessibilidade da aplicação original for igual ao da aplicação mutante, resultando-se em um mutante vivo, provavelmente o conjunto de teste precisa ser revisado e melhorado.

O principal objetivo em definir os operadores de mutação de acessibilidade é certificar de que o conjunto de teste criado pelo testador explore todos, ou pelo menos a maioria, dos elementos de GUI da aplicação. Para cumprir tal objetivo, foram definidos 9 operadores de mutação que vão de desencontro aos critérios de sucesso dos princípios de acessibilidade. Todos os operadores possuem a mesma característica de remoção. Optou-se por operadores de remoção, pois estudos anteriores evidenciam que tais operadores produzem menos mutantes, porém eficazes (Delamaro et al., 2014). Para definir este conjunto levaram-se em conta 9 *code elements* da Tabela 4.2. A Tabela 4.3 por sua vez, elenca os operadores de mutação definidos.

Tabela 4.3: Relação entre princípio do guia WCAG e os OM propostos.

Princípio	**Operador de Mutação**	**Descrição**
Perceptível	Missing textSize - MTS	Remove o *code element* textSize
	Missing inputType - MIT	Remove o *code element* inputType
	Missing autoSizeTextType - MASTT	Remove o *code element* autoSizeTextType
Operável	Missing nextFocusDownId - MNFD	Remove o *code element* nextFocusDown
	Missing minTouchSizeArea - MMTSA	Remove os *code elements* minHeight e minWidth
Entendível	Missing labelFor - MLF	Remove o *code element* labelFor
	Missing hint - MH	Remove o *code element* hint
Robusto	Missing importantForAccessibility - MIA	Remove o *code element* importantForAccessibility
	Missing accessibilityLiveRegion - MALR	Remove o *code element* accessibilityLiveRegion

A definição de cada operador depende do entendimento básico de alguns elementos ao desenvolver para a plataforma Android. Os principais são: (i) *widget w*, elemento visível e presente na GUI da aplicação móvel; (ii) *activity A*, constituída de um ou mais *w*, representa a interface da aplicação. Os 9 operadores propostos são descritos a seguir.

4.2.1 Missing textSize - MTS

O *code element* textSize define de forma estática o tamanho da fonte de um *widget*. Unidades complexas de tamanho também são utilizadas: sp, dp, pt, px, mm, in. Pode ser definido via arquivo XML (Código 4.1) ou pela linguagem Java (Código 4.2).

Código 4.1: Uso do textSize - XML.

```
<TextView
    android:id="@+id/text_view2"
    android:text="Text.."
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textColor="#FF0000"
    android:textSize="25dp"
/>
```

Código 4.2: Uso do textSize - Java.

```
public void perform_action(View v){
    TextView tv1 = (TextView) findViewById(R.id.text_view2 );
    tv1.setTextSize(25);
}
```

A mutação consiste na remoção da linha que utiliza o *code element* textSize. Versões mutantes dos Códigos 4.1 e 4.2 são apresentadas, respectivamente, pelos Códigos 4.3 e 4.4.

Código 4.3: Mutante *m* do OM MTS - XML.

```
<TextView
    android:id="@+id/text_view2"
    android:text="Text..)"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textColor="#FF0000"
    - android:textSize="25dp"
/>
```

Código 4.4: Mutante *m* do OM MTS - Java.

```
public void perform_action(View v){
    TextView tv1 = (TextView) findViewById(R.id.text_view1 );
    - tv1.setTextSize(50);
}
```

4.2.2 Missing inputType - MIT

O *code element* inputType permite especificar vários comportamentos para o método de entrada. Se o campo de entrada EditText for específico para número de telefone, o atributo recebe o valor "phone". Para texto simples "textAutoCorrect". O operador MIT remove o *code element* inputType de campos de entrada da aplicação. Os Códigos 4.5 e 4.6 demonstram as duas maneiras de uso inputType.

Código 4.5: Uso do inputType - XML.

```
<EditText
    android:id="@+id/phone"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:hint="@string/phone_hint"
    android:inputType="phone"
/>
```

Código 4.6: Uso do inputType - Java.

```
switch (motionEvent.getAction()) {
    case MotionEvent.ACTION_DOWN:
        editpass.setInputType(InputType.TYPE_CLASS_TEXT);
        break;
}
```

Versões mutantes dos Códigos 4.5 e 4.6 são apresentadas, respectivamente, pelos Códigos 4.7 e 4.8.

Código 4.7: Mutante *m* do OM MIT - XML.

```
<EditText
    android:id="@+id/phone"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:hint="@string/phone_hint"
    - android:inputType="phone"
/>
```

Código 4.8: Mutante *m* do OM MIT - Java.

```
switch (motionEvent.getAction()) {
    case MotionEvent.ACTION_DOWN:
        - editpass.setInputType(InputType.TYPE_CLASS_TEXT);
        break;
}
```

4.2.3 Missing autoSizeTextType - MASTT

O *code element* autoSizeTextType permite o dimensionamento automático do componente TextView. O dimensionamento ocorre horizontalmente e verticalmente para ocupar o espaço disponível limitado pelo próprio TextView. Este redimensionamento permite que o conteúdo exposto pelo TextView continue legível sob o *zoom* do *smartphone*. O atributo pode receber o valor "none", que desativa o comportamento de redimensão, e o valor "uniform" que o ativa. A utilização nos contextos XML e Java pode ser vista nos Códigos 4.9 e 4.10.

Código 4.9: Uso do autoSizeTextType - XML.

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="200dp"
    android:autoSizeTextType="uniform"
/>
```

Código 4.10: Uso do autoSizeTextType - Java.

```
public void onClick(View view) {
    TextView textview = (TextView) view;
    int AUTO_SIZE_TYPE
        = TextViewCompat.AUTO_SIZE_TEXT_TYPE_UNIFORM;
    TextViewCompat
        .setAutoSizeTextTypeWithDefaults(textview,
            AUTO_SIZE_TYPE);
}
```

Versões mutantes dos Códigos 4.9 e 4.10 são apresentadas, respectivamente, pelos Códigos 4.11 e 4.12.

Código 4.11: Mutante *m* do OM MASTT - XML.

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="200dp"
    - android:autoSizeTextType="uniform"
/>
```

Código 4.12: Mutante *m* do OM MASTT - Java.

```
public void onClick(View view) {
    TextView textview = (TextView) view;
    int AUTO_SIZE_TYPE
        = TextViewCompat.AUTO_SIZE_TEXT_TYPE_UNIFORM;
    - TextViewCompat
        .setAutoSizeTextTypeWithDefaults(textview,
            AUTO_SIZE_TYPE);
}
```

4.2.4 Missing nextFocusDown - MNFD

Usuários também podem navegar pela aplicação utilizando os botões de flechas do teclado. O *code element* nextFocusDown auxilia nesta interação definindo qual será o foco do próximo *widget*. Ele é definido via XML (Código 4.13) ou dentro do contexto Java (Código 4.14).

Código 4.13: Uso do nextFocusDown - XML.

```
<Button
    android:id="@+id/button1"
    android:nextFocusDown="@+id/button2"
    ... />
<Button
    android:id="@id/button2"
    ... />
```

Código 4.14: Uso do nextFocusDown - Java.

```
public void onFocusChange(View v, boolean hasFocus) {
    findViewById(R.id.email)
        .setNextFocusDownId(R.id.password);
}
```

Versões mutantes dos Códigos 4.13 e 4.14 são apresentadas, respectivamente, pelos Códigos 4.15 e 4.16.

Código 4.15: Mutante *m* do OM MNFD - XML.

```
<Button
    android:id="@+id/button1"
    - android:nextFocusDown="@+id/button2"
    ... />
<Button
    android:id="@id/button2"
    ... />
```

Código 4.16: Mutante *m* do OM MNFD - Java.

```
public void onFocusChange(View v, boolean hasFocus) {
    - findViewById(R.id.email)
        .setNextFocusDownId(R.id.password);
}
```

4.2.5 Missing minTouchSizeArea - MMTSA

Os *code elements* minWidth e minHeight são responsáveis por definir o tamanho mínimo de um elemento *widget* da plataforma Android. Podem ser utilizados dentro do contexto XML (Código 4.17) e Java (Código 4.18).

Código 4.17: Uso do minWidth e minHeight - XML.

```
<TextView
    android:minWidth="48dp"
    android:minHeight="48dp"
    android:id="id1"
/>
```

Código 4.18: Uso do minWidth e minHeight - Java.

```
public View createTabContent(String tag) {
    View v = new View(mContext);
    v.setMinimumWidth(48);
    v.setMinimumHeight(48);
    return v;
}
```

Versões mutantes dos Códigos 4.17 e 4.18 são apresentadas, respectivamente, pelos Códigos 4.19 e 4.20.

Código 4.19: Mutante *m* do OM MMTSA - XML.

```
<TextView
    - android:minWidth="48dp"
    - android:minHeight="48dp"
    android:id="id1"
/>
```

Código 4.20: Mutante *m* do OM MMTSA - Java.

```
public View createTabContent(String tag) {
    View v = new View(mContext);
    - v.setMinimumWidth(48);
    - v.setMinimumHeight(48);
    return v;
}
```

4.2.6 Missing labelFor - MLF

O *code element* labelFor é um rótulo que acompanha um *widget*. Pode ser definido via arquivo XML ou pela linguagem Java. De modo geral, provê rótulos de descrição e exploração para algum elemento da tela. Este atributo é apenas usado para aplicações Android com API 17 ou superior. Os Códigos 4.21 e 4.22 demonstram a utilização do labelFor.

Código 4.21: Uso do labelFor - XML.

```
<TextView
    android:labelFor="EditText1"
    android:text="Texto" />

<EditText android:id="EditText1"/>
```

Código 4.22: Uso do labelFor - Java.

```
TextView textView =
    (TextView) view.findViewById(
        R.id.aac_edit_text_fixed_heading_1 );
textView.setLabelFor(editText);
```

Versões mutantes dos Códigos 4.21 e 4.22 são apresentadas, respectivamente, pelos Códigos 4.23 e 4.24.

Código 4.23: Mutante *m* do OM MLF - XML.

```
<TextView
    - android:labelFor="EditText1"
    android:text="Texto" />

<EditText android:id="EditText1"/>
```

Código 4.24: Mutante *m* do OM MLF - Java.

```
TextView textView =
    (TextView) view.findViewById(
        R.id.aac_edit_text_fixed_heading_1 );
- textView.setLabelFor(editText);
```

4.2.7 Missing hint - MH

O *code element* hint é um rótulo temporário atribuído somente aos campos editáveis. Eles são necessários para que o TalkBack, ou qualquer outro leitor de tela, reporte corretamente qual informação a aplicação móvel necessita.

Código 4.25: Uso do hint - XML.

```
<EditText
    android:gravity="left"
    android:id="EditTextEmail"
    android:hint="Email"
/>
```

Código 4.26: Uso do hint - Java.

```
myEditText.setOnFocusListener(new OnFocusListener(){
    public void onFocus(){
        myEditText.setHint("Email");
    }
}
```

Versões mutantes dos Códigos 4.25 e 4.26 são apresentadas, respectivamente, pelos Códigos 4.27 e 4.28.

Código 4.27: Mutante *m* do OM MH - XML.

```
<EditText
    android:gravity="left"
    android:id="EditTextEmail"
    - android:hint="Email"
/>
```

Código 4.28: Mutante *m* do OM MH - Java.

```
myEditText.setOnFocusListener(new OnFocusListener(){
    public void onFocus(){
        - myEditText.setHint("Email");
    }
}
```

4.2.8 Missing importantForAccessibility - MIA

Descreve se o *widget* é ou não importante para acessibilidade. Se for importante, o *widget* dispara eventos de acessibilidade e é relatado aos serviços de acessibilidade que consultam a tela. Este *code element* foi adicionado na API 16 do Android e pode ser definido por XML (Código 4.29) ou por Java (Código 4.30).

Código 4.29: Uso do importantForAccessibility - XML.

```
<TextView
    android:id="@+id/myTextviewId1"
    android:importantForAccessibility="1"
/>
```

Código 4.30: Uso do importantForAccessibility - Java.

```
myTextView1 = (TextView) findViewById(R.id.myTextviewId1);
myTextView1.setImportantForAccessibility(1);
```

Versões mutantes dos Códigos 4.29 e 4.30 são apresentadas, respectivamente, pelos Códigos 4.31 e 4.32.

Código 4.31: Mutante *m* do OM MIA - XML.

```
<TextView
    android:id="@+id/myTextviewId1"
    - android:importantForAccessibility="1"
/>
```

Código 4.32: Mutante *m* do OM MIA - Java.

```
myTextView1 = (TextView) findViewById(R.id.myTextviewId1);
- myTextView1.setImportantForAccessibility(1);
```

4.2.9 Missing accessibilityLiveRegion - MALR

O *code element* accessibilityLiveRegion indica ao serviço de acessibilidade se o usuário será notificado no momento em que existir uma mudança na View. Foi adicionado a partir da versão da API 19. O valor “none” indica que mudanças na View não devem ser anunciadas. Ao inserir “polite”, as mudanças na View serão notificadas. Ao introduzir “assertive” o serviço de acessibilidade deve interromper a leitura atual para informar mudanças na View. Os Códigos 4.33 e 4.34 representam o uso do atributo nos contextos XML e Java.

Código 4.33: Uso do accessibilityLiveRegion - XML.

```
<TextView
    android:id="TextViewId1"
    android:focusable="true"
    android:accessibilityLiveRegion = "polite"
/>
```

Código 4.34: Uso do accessibilityLiveRegion - Java.

```
public ToolbarProgressBar(Context context, AttributeSet attrs){
  super(context, attrs);
  setAlpha(0.0f);
  ViewCompat.setAccessibilityLiveRegion(this,
    ViewCompat.
      ACCESSIBILITY_LIVE_REGION_POLITE);
}
```

Versões mutantes dos Códigos 4.33 e 4.34 são apresentadas, respectivamente, pelos Códigos 4.35 e 4.36.

Código 4.35: Mutante *m* do OM MALR - XML.

```
<TextView
    android:id="TextViewId1"
    android:focusable="true"
    - android:accessibilityLiveRegion = "polite"
/>
```

Código 4.36: Mutante *m* do OM MALR - Java.

```
public ToolbarProgressBar(Context context, AttributeSet attrs){
  super(context, attrs);
  setAlpha(0.0f);
  - ViewCompat.setAccessibilityLiveRegion(this,
    ViewCompat.
      ACCESSIBILITY_LIVE_REGION_POLITE);
}
```

4.3 O PROCESSO DE TESTE

A abordagem `AccessibilityMDroid` foi implementada em uma ferramenta de mesmo nome. Esta ferramenta além de gerar mutantes usando os operadores de mutação de acessibilidade, inclui uma estratégia com a qual seja possível realizar a análise dos mutantes. O processo de teste para a aplicação desta estratégia está representado na Figura 4.1.

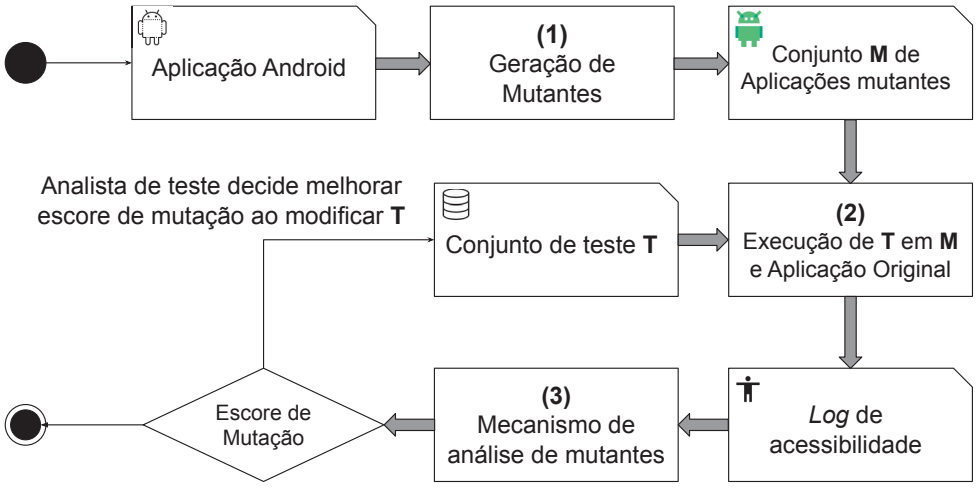


Figura 4.1: Fluxo de execução da abordagem.

O processo inclui três etapas. A primeira é a geração de mutantes usando os operadores de mutação de acessibilidade definidos. Esta etapa produz um conjunto de aplicações mutantes M. Na segunda etapa, a aplicação original e os mutantes em M são executados com um conjunto de teste T, que pode ser construído com a estratégia de preferência do testador. No entanto, para a análise de mutantes, o processo requer que T seja implementado e executado usando uma ferramenta de acessibilidade, como as relatadas na Seção 3.1. A terceira etapa, a análise de mutantes, permite calcular o escore de mutação comparando os relatórios de acessibilidade produzidos por um verificador de acessibilidade para as aplicações originais e mutantes. Se

os *logs* de acessibilidade forem diferentes, ou seja, diferentes falhas de acessibilidade forem encontradas, o mutante pode ser considerado morto. Portanto, se o *log* de acessibilidade da aplicação original for o mesmo do aplicação mutante, resultando em um mutante vivo, o conjunto de teste provavelmente precisará ser revisado e aprimorado. Se o escore não for satisfatório, o testador pode adicionar novos casos de teste ou modificar os existentes em T para que mais mutantes sejam mortos.

4.3.1 Exemplo de execução

Para ilustrar a abordagem `AccessibilityMDroid` utilizou-se `Sample`, uma aplicação de amostra criada a partir de um *template* disponibilizado pelo ambiente Android Studio. `Sample` é uma aplicação simples (Figura 4.2) com apenas uma tela de formulário de login, composta por um botão de submissão e pelos campos de entrada de texto: Nickname, Email e Senha.

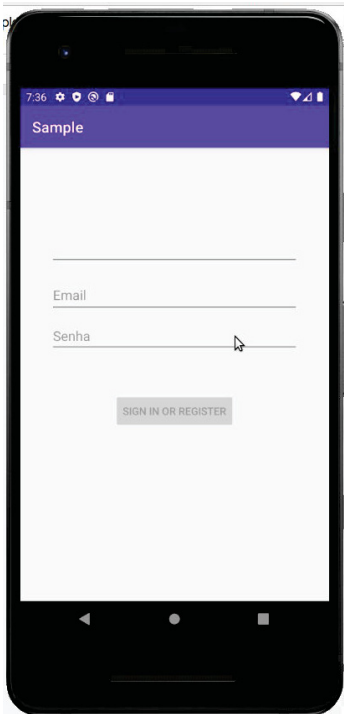


Figura 4.2: Aplicação `Sample`.

Um trecho de código desta aplicação é apresentado no Código 4.37. O operador MH (Missing hint) remove do elemento GUI (Nickname) o *code element* hint (Linha 22 do Código 4.37).

Código 4.37: Mutante gerada pelo OM MH.

```
14 <EditText
15    android:id="@+id/nickname"
16    android:layout_width="0dp"
17    android:layout_height="wrap_content"
18    android:layout_marginStart="24dp"
19    android:layout_marginTop="96dp"
20    android:layout_marginEnd="24dp"
21    android:inputType="text"
22    - android:hint="Nickname"
23    android:selectAllOnFocus="true"
24    app:layout_constraintEnd_toEndOf="parent"
25    app:layout_constraintStart_toStartOf="parent"
26    app:layout_constraintTop_toTopOf="parent" />
```

Suponha que para aplicação `Sample`, um conjunto de teste *T*, conforme representado no Código 4.38, esteja disponível. Quando *T* é executado com o Espresso no mutante *m* (Etapa 2), um *log* é gerado. Este *log* é comparado ao *log* gerado pela execução de *T* na aplicação original (Etapa 3). A partir da diferença entre os dois registros de acessibilidade, é possível determinar a morte do mutante.

Código 4.38: Conjunto de teste implementado com Espresso

```
@Test
public void loginTest() {
   var appCompatEditText = onView(allOf(
     withId(R.id.username),
     childAtPosition(allOf(withId(R.id.container),
     childAtPosition(withId(android.R.id.content), 0)), 1),
     isDisplayed()));

   appCompatEditText.perform(replaceText("email"), closeSoftKeyboard());

   var appCompatEditText2 = onView(allOf(
     withId(R.id.password),
     childAtPosition(allOf(withId(R.id.container),
     childAtPosition(withId(android.R.id.content), 0)), 2),
     isDisplayed()));

   appCompatEditText2.perform(replaceText("123456"), closeSoftKeyboard());

   var appCompatEditText3 = onView(allOf(
     withId(R.id.password), withText("123456"),
     childAtPosition(allOf(withId(R.id.container),
     childAtPosition(withId(android.R.id.content), 0)), 2),
     isDisplayed()));

   appCompatEditText3.perform(pressImeActionButton());
}
```

Nesse caso, *T* não foi suficiente para mostrar a diferença entre a aplicação original e a mutante. Como ambos produzem o mesmo *log* (Código 4.40), o mutante ainda está vivo. O testador agora tenta melhorar *T* e percebe que os testes existentes não interagem com um dos campos de entrada da aplicação. Considere o conjunto modificado *T* como *xT*. Após as alterações (ilustrado no Código 4.39), a Etapa 2 é executada novamente e o *log* para *m* é agora o Código 4.41; difere-se do original na primeira linha.

Código 4.39: Conjunto de teste *xT* alterado pelo testador.

```
@Test
public void loginTest() {
+  onView(withId(R.id.nickname)).perform(typeText("nick"),
+  closeSoftKeyboard());
   var appCompatEditText = ...
}
```

Código 4.40: *Log* de acessibilidade gerado a partir da execução de *T* na Aplicação original e em *m*.

```
AppCompatEditText{id=2131230902,res-name=nickname}: View falls below the minimum recommended
    size for touch targets. Minimum touch target size is 48x48dp. Actual size is 331.4x45.0dp
    (screen density is 2.6).
AppCompatEditText{id=2131230902,res-name=nickname}: View falls below the minimum recommended
    size for touch targets. Minimum touch target size is 48x48dp. Actual size is 331.4x45.0dp
    (screen density is 2.6).
AppCompatEditText{id=2131230917,res-name=password}: View falls below the minimum recommended
    size for touch targets. Minimum touch target size is 48x48dp. Actual size is 331.4x45.0dp
    (screen density is 2.6).
AppCompatEditText{id=2131230917,res-name=password}: View falls below the minimum recommended
    size for touch targets. Minimum touch target size is 48x48dp. Actual size is 331.4x45.0dp
    (screen density is 2.6).
AppCompatEditText{id=2131230917,res-name=password}: View falls below the minimum recommended
    size for touch targets. Minimum touch target size is 48x48dp. Actual size is 331.4x45.0dp
    (screen density is 2.6).
```

Código 4.41: *Log* de acessibilidade gerado a partir da execução do conjunto alterado xT em m.

```
+ AppCompatEditText{id=2131230902,res-name=nickname}:  View is missing
   speakable text needed for a screen reader
AppCompatEditText{id=2131230902,res-name=nickname}: View falls below the minimum recommended
    size for touch targets. Minimum touch target size is 48x48dp. Actual size is 331.4x45.0dp
    (screen density is 2.6).
AppCompatEditText{id=2131230902,res-name=nickname}: View falls below the minimum recommended
    size for touch targets. Minimum touch target size is 48x48dp. Actual size is 331.4x45.0dp
    (screen density is 2.6).
AppCompatEditText{id=2131230917,res-name=password}: View falls below the minimum recommended
    size for touch targets. Minimum touch target size is 48x48dp. Actual size is 331.4x45.0dp
    (screen density is 2.6).
AppCompatEditText{id=2131230917,res-name=password}: View falls below the minimum recommended
    size for touch targets. Minimum touch target size is 48x48dp. Actual size is 331.4x45.0dp
    (screen density is 2.6).
AppCompatEditText{id=2131230917,res-name=password}: View falls below the minimum recommended
    size for touch targets. Minimum touch target size is 48x48dp. Actual size is 331.4x45.0dp
    (screen density is 2.6).
```

Ao empregar este procedimento para matar mutantes de acessibilidade, T atinge um escore de mutação mais alto, cobre mais elementos de GUI e potencialmente revela outras falhas de acessibilidade.

4.4 ASPECTOS DE IMPLEMENTAÇÃO

Dada a rápida evolução da plataforma Android, Moran et al. (2018) prepararam a ferramenta MDroid+ para que sua lista de operadores seja modificada ou estendida. Esta extensão, segundos os autores, é feita com a implementação de dois componentes: (i) Localizador de operador e (ii) Modificador de operador. Caso o operador esteja em um arquivo Java, a interface Locator é utilizada; para arquivos XML, a interface TextBasedDetector é adotada. A interface MutationOperator é utilizada para ambos os tipos de arquivo.

O Código 4.42 representa a implementação do localizador do operador de mutação MissingSetHint. Esta classe é responsável por retornar o índice de linhas de ocorrência do trecho de código que se deseja alterar. No caso atual, busca-se a ocorrência do método “setHint” dentro de arquivos da extensão Java.

Código 4.42: Localizador do operador MissingSetHint.

```
1  public class MissingSetHintLocator implements Locator {
2     @Override
3     public List<MutationLocation> findExactLocations(List<MutationLocation> locations) {
4        List<MutationLocation> exactMutationLocations = new ArrayList<MutationLocation>();
5        for(MutationLocation loc : locations){
6           try{
7              loc.setStartColumn(loc.getStartColumn()+1);
8              List<String> lines = FileHelper.readLines(loc.getFilePath());
9              String targetLine = lines.get(loc.getLine());
10             int assignIndex = targetLine.indexOf("setHint");
11             if(assignIndex<0){
12                   continue;
13             }
14             exactMutationLocations.add(loc);
15          }catch(Exception e){
16             e.printStackTrace();
17          }
18       }
19    return exactMutationLocations;
20    }
21 }
```

Depois de implementar o Localizador, precisa-se definir o Modificador. O Modificador determina o comportamento depois que a linha de código é encontrada (Código 4.43). Como o

operador MissingSetHint representa a exclusão do método setHint dos arquivos Java da aplicação móvel, o Modificador implementa a remoção.

Código 4.43: Modificador do operador MissingSetHint.

```
public class MissingSetHint implements MutationOperator{
      @Override
      public boolean performMutation(MutationLocation location) {
            List<String> newLines = new ArrayList<String>();
            List<String> lines = FileHelper.readLines(location.getFilePath());
            for(int i=0; i < lines.size(); i++){
                  String currLine = lines.get(i);
                  if(i != location.getStartLine()){
                        newLines.add(currLine);
                  }
            }
            FileHelper.writeLines(location.getFilePath(), newLines);
            return true;
      }
}
```

Ao fim da implementação da etapa de geração de mutantes da abordagem deste trabalho, integrou-se ao projeto da MDroid+ 36 arquivos Java, que totalizam uma soma de 1.566 linhas de código. A implementação encontra-se disponível em: `https://osf.io/vfs2d/?view_only=6c3af7cdbb7f4132a9367e196735c68f`.

4.5 CONSIDERAÇÕES FINAIS

Neste capítulo, apresentaram-se os elementos que compõem a abordagem da dissertação. Primeiro, definiu-se o conjunto de operadores de mutação de acessibilidade. Em seguida, o processo de teste é apresentado e exemplificado. Por fim, são apresentados aspectos ligados a implementação da extensão da ferramenta MDroid+.

No próximo capítulo, descreve-se o estudo experimental conduzido para avaliar os operadores propostos e a implementação para realizar a análise dos mutantes gerados.

5 AVALIAÇÃO

O capítulo atual apresenta os experimentos conduzidos para avaliar a abordagem `AccessibilityMDroid`. A avaliação foi conduzida com o conjunto definido de operadores de mutação de acessibilidade e também com o processo de análise de mutante. A avaliação está organizada da seguinte forma: a Seção 5.1 elenca as questões de pesquisa; a Seção 5.2 descreve a configuração realizada para o estudo experimental; na Seção 5.3 estão os resultados encontrados; na Seção 5.4 estão as ameaças à validade do estudo experimental.

5.1 QUESTÕES DE PESQUISA

O principal objetivo dos operadores propostos é auxiliar na geração de dados de teste e avaliação de falhas de acessibilidade. Para avaliar adequadamente esses aspectos definiram-se três questões de pesquisa:

Q1. **Como é a aplicabilidade dos operadores de mutação de acessibilidade?** Esta questão visa a investigar se os operadores e processo propostos são aplicáveis na prática. Para responder a esta pergunta, avaliou-se o custo da aplicação da abordagem analisando o número de mutantes gerados, bem como o número de casos de teste necessários.

Q2. **Quão adequados são os conjuntos de testes existentes em relação ao teste de mutação de acessibilidade?** Esta questão avalia o uso dos operadores propostos como um critério de avaliação. Eles são usados para avaliação de qualidade dos conjuntos de teste que acompanham as aplicações selecionadas com relação à acessibilidade. Para isso, analisou-se a capacidade dos conjuntos de teste existentes de matar os mutantes gerados.

Q3. **Os operadores de mutação contribuem para revelar novas falhas de acessibilidade?** Esta questão avalia o uso dos operadores propostos como um critério de seleção de teste. Observou-se a eficácia dos conjuntos de teste adequados aos operadores propostos em relação ao número de violações de acessibilidade.

5.2 CONFIGURAÇÃO DO ESTUDO

Para viabilizar a condução do experimento e responder às questões de pesquisa elencadas, foi implementada parte dos operadores definidos. O subconjunto de OM é constituído por: MTS, MIT, MH, MIA, MLF e MNFD (definidos na Seção 4.2). Estes operadores estão relacionados com os *code elements* mais utilizados na prática, segundo o estudo conduzido por Silva et al. (2020). A amostra do estudo experimental é composta por aplicações Android de código-aberto do repositório F-Droid[1]. Os repositórios das aplicações selecionadas possuem atualizações recentes (2019/2020), contendo conjuntos de teste implementados com Espresso. O conjunto de teste que acompanha o projeto é referenciado como T. Removeram-se as aplicações que não compilaram e cujos testes não eram compatíveis com o recurso de verificação de acessibilidade. Todos os artefatos usados na área de avaliação estão disponíveis em: `Repositório OSF`.

Para cada aplicação, utilizou-se o conjunto de operadores de mutação definido para gerar os mutantes. Para cada mutante gerado, executou-se T. Ao fim da execução de T coletaram-se

[1] `https://www.f-droid.org`

os *logs* de acessibilidade de cada mutante para compará-los com o *log* da aplicação original. Desta forma, obteve-se o conjunto de mutantes mortos por T. Depois disso, inspecionou-se manualmente os mutantes vivos percebendo que, muitas vezes, alguns dos casos de teste de T apesar de executarem o código mutado, não produziram nenhuma diferença no *log* devido a alguma limitação da Espresso. Nesse caso, marcou-se o mutante como “coberto”. Outros mutantes foram marcados como "inalcançáveis", uma vez que suas mutações estão relacionadas a *widgets* que não são acessíveis na aplicação (por exemplo, código morto).

Então, contou-se o número de mutantes gerados, mortos, cobertos e inalcançáveis por T. Em seguida, estendeu-se T para que todos os mutantes fossem mortos ou pelo menos cobertos. A este conjunto de teste estendido deu-se o nome de xT. A inclusão de um caso de teste foi conduzida da seguinte maneira: (i) escolher um mutante vivo (não coberto ou morto por T); (ii) registrar um teste que exercite a mutação usando o *Espresso Test Recorder* no Android Studio e, se necessário, refatorar o código de teste para torná-lo reproduzível[2]; e (iii) analisar se o mutante é morto pelo novo teste, caso contrário marcá-lo como coberto. As informações sobre os mutantes foram coletadas novamente para xT.

Como indicadores de custo, coletou-se o número de testes do conjunto de teste T, e seu tamanho, dado pelo número de linhas do código de teste LC. Quanto à eficácia, contou-se por conjunto de teste o número de falhas de acessibilidade relatadas pela verificação de acessibilidade da ferramenta Espresso.

A Tabela 5.1 mostra informações sobre as sete aplicações selecionadas. `Authorizer` é a aplicação com o maior valor de LC (28.286), enquanto `AnyMemo` tem 30 *activities* (A.). `AlarmClock` é a aplicação com o menor número de LC: 1.349, e `Equate` tem apenas 2 *activities*. A tabela também mostra o número de casos de teste (CT) e LC para o conjunto original T e para xT. Observe que `AlarmClock` tem 41 testes e 1.068 linhas de código de teste (LC (T)). `Kolabnotes` tem apenas um teste, mas `AnyMemo` tem o menor LC (T) (76). Com relação a xT, `AlarmClock` e `Authorizer` requiriram mais testes (ambos 43) e mais LC (xT) (1.341 e 1.700, respectivamente). `PleesTracker` tem o menor número de casos de teste (5) e LC (xT) (345). No entanto, `Authorizer` exigiu mais casos de teste adicionais, 32, enquanto `Piwigo` apenas um.

Tabela 5.1: Aplicações selecionadas.

Aplicação*	**LC**	**#A.**	**#CT(T)**	**LC(T)**	**#CT(xT)**	**CT(xT)**
`AlarmClock`	1.349	5	41	1.068	43	1.341
`AnyMemo`	19.751	30	3	76	13	932
`Authorizer`	28.286	7	11	652	43	1.700
`Equate`	5.826	2	6	511	9	709
`Kolabnotes`	11.025	9	1	494	6	884
`Piwigo`	4.744	7	8	408	9	579
`PleesTracker`	1.868	5	2	89	5	345

* O nome da aplicação é um link clicável para o projeto GitHub.

5.3 ANÁLISE DOS RESULTADOS

A Tabela 5.2 resume os principais resultados da avaliação e é usada nesta seção para responder às questões de pesquisa. Esta tabela mostra o número de mutantes que foram gerados (colunas G), mortos por algum teste (colunas M), cobertos (colunas C), e inalcançáveis (colunas I). Observe que os resultados são mostrados para 4 de 6 operadores descritos na Tabela 4.3; os

[2]O código gerado pelo *Espresso Test Recorder* pode ser muito específico e falhar nas reexecuções.

operadores MLF e MNFD não geraram nenhum mutante para as aplicações selecionadas. Para cada aplicação, duas linhas são apresentadas, uma para os resultados obtidos por *T* e outra para *xT*. As últimas quatro colunas listam o total para todos os operadores, enquanto as últimas linhas trazem o total para todos as aplicações.

Por exemplo, para `AnyMemo` o operador MTS gerou 64 mutantes, 11 inalcançáveis. O conjunto de teste *T* não foi capaz de matar nenhum mutante, mas cobriu 14. O conjunto *xT* cobriu 52, ou seja, 38 mutantes adicionais podem ser cobertos. Observe que os mutantes mortos não são contados como cobertos. Considerando todos os operadores, apenas os mutantes do operador MH foi morto por *xT* e 70 mutantes foram cobertos de 84 mutantes gerados. Para esta aplicação, 4 mutantes foram gerados a partir de uma alteração de tela que é alcançada apenas quando integrado com aplicações de terceiros. Para exercitar esses mutantes exigir-se-ia o uso de outras ferramentas além do Espresso e portanto, não foi possível cobri-los. Mas eles não podem ser classificados como inalcançáveis. Por causa disso, a soma de mutantes mortos, cobertos e inalcançáveis não é igual ao número de mutantes gerados para esta aplicação, como acontece para as demais aplicações.

Tabela 5.2: Resumo dos resultados por operador.

Aplicação Android		**Operadores de Mutação**																**Total**			
		MTS				MIT				MH				MIA							
		G	M	C	I	G	M	C	I	G	M	C	I	G	M	C	I	G	M	C	I
`AlarmClock`	*T*	12	-	9	-	1	-	-	-	1	-	-	-	-	-	-	-	14	-	9	-
	xT		-	12			-	1			1	-			-	-			1	13	
`AnyMemo`	*T*	64	-	14	11	22	-	-	-	-	-	-	-	-	-	-	-	86	-	14	11
	xT		1	52			-	18			-	-			-	-			1	70	
`Authorizer`	*T*	18	-	1	-	27	-	3	-	18	-	3	-	9	-	2	-	73	-	9	-
	xT		-	18			-	27			6	12			-	9			6	67	
`Equate`	*T*	3	-	-	-	2	-	-	-	2	1	-	1	-	-	-	-	7	1	-	1
	xT		3	-			-	2			1	-			-	-			1	5	
`Kolabnotes`	*T*	23	-	8	-	13	-	3	-	12	-	-	-	-	-	-	-	48	-	11	-
	xT		-	23			-	13			8	4			-	-			8	40	
`Piwigo`	*T*	1	-	-	-	3	-	3	-	1	1	-	-	-	-	-	-	5	1	3	-
	xT		-	1			-	3			1	-			-	-			1	4	
`PleesTracker`	*T*	24	-	8	-	-	-	-	-	-	-	-	-	-	-	-	-	24	-	8	-
	xT		-	24			-	-			-	-			-	-			-	24	
Total	*T*	145	-	40	11	64	-	9	-	34	2	3	1	9	-	2	-	257	2	54	12
	xT		1	133			-	68			17	24			-	9			18	223	

Número de mutantes **G**erados, **M**ortos, **C**obertos, **I**nacessível pelo conjunto de teste original *T* e o conjunto estendido *xT*.

5.3.1 Aplicabilidade da abordagem

Para responder à **Q1**, avaliou-se o número de mutantes gerados por cada operador. Observa-se na Tabela 5.2 que o operador MTS gerou a maior quantidade de mutantes (145 no total), seguido por MIT (64), MH (34) e MIA (9). MTS gerou mutantes para todas as aplicações, MIT para 6 e MH para 5. O operador MIA gerou mutantes apenas para a aplicação `Authorizer`.

No total, foram gerados 253 mutantes, sendo `AnyMemo` com mais mutantes (86) e `Piwigo` com 5. Isso significa que as aplicações móveis selecionadas contêm mais *code elements* associados ao princípio Perceptível (operadores MTS e MIT), o que pode indicar que os desenvolvedores estão preocupados com as descrições de conteúdo para elementos não textuais mais do que o princípio Robusto (operador MIA que gerou mutantes para apenas uma aplicação) ou Operável (operador MNFD que não gerou nenhum mutante).

Os operadores MIT e MIA geraram mutantes que não podiam ser mortos; apenas um mutante de MTS foi morto, e 17 dos 34 mutantes gerados por MH foram mortos. O processo usando o Espresso foi capaz de distinguir mutantes, na grande maioria gerados pela remoção do *code element* hint. Analisando os mutantes vivos, identificaram-se 223 como cobertos e 12 como inalcançáveis. Mutantes inalcançáveis foram gerados principalmente para `AnyMemo`.

Para uma análise mais aprofundada, a Tabela 5.3 contém o número de mutantes gerados de cada aplicação divido pela quantidade de linhas de código (LC). As duas últimas colunas apresentam informações sobre o esforço necessário para adicionar novos casos de teste para que um conjunto de testes adequado para mutantes de acessibilidade seja obtido.

Tabela 5.3: Esforços para construir xT.

Aplicação	**#Mutantes / Aplicação (KLC)**					**A-CT**	**A-LC**
	MTS	MIT	MH	MIA	Total		
`AlarmClock`	8,9	0,7	0,7	0,0	10,37	2	273
`AnyMemo`	3,2	1,1	0,0	0,0	4,35	10	856
`Authorizer`	0,6	0,9	0,6	0,3	2,58	32	1.048
`Equate`	0,5	0,3	0,3	0,0	1,20	3	198
`Kolabnotes`	2,0	1,1	1,0	0,0	4,35	5	390
`Piwigo`	0,2	0,6	0,2	0,0	1,05	1	171
`PleesTracker`	12,8	0,0	0,0	0,0	12,8	3	256
Média	4,0	0,67	0,4	0,043	5,42	8	456

A-CT representa o número de Casos de Teste adicionados a T para obter xT.
A-LC representa o número de Linhas de Código adicionado a T para obter xT.

Observe que um maior número de mutantes é gerado para as maiores aplicações em termos de LC e número de *activities*: `AnyMemo`, `Authorizer` e `Kolabnotes`. Dado que os operadores propostos apenas removem *code elements*, o número de mutantes tende a ser igual ao número de elementos existentes associados aos critérios de sucesso do guia WCAG.

Devido a essa característica, os operadores não geraram mutantes equivalentes. Isso é uma vantagem, porque a identificação de tais mutantes costuma ser custosa. Além disso, não encontrou-se mutantes natimortos nem mutantes triviais. Os primeiros são mutantes que não compilam e os segundos são mutantes que falham na inicialização (Ammann e Offutt, 2016, Capítulo 9).

Também mediu-se o esforço de adicionar novos casos de teste, considerando os valores da Tabela 5.1. Como mostra a Tabela 5.3, `Authorizer` exigiu mais esforço, exigiu 32 testes adicionais (com 1.048 A-LC), seguido por `AnyMemo`: que exigiu 10 testes adicionais (com 856 A-LC); e `Kolabnotes`: 5 testes (390 A-LC). Essas aplicações são os maiores em termos de tamanho.

Resposta à Q1: O número de mutantes está relacionado ao tamanho da aplicação, principalmente ao número de elementos da GUI e *code elements* associados aos critérios de sucesso de acessibilidade. Operadores MTS e MIT, relacionados ao princípio Perceptível produzem mais mutantes, enquanto nenhum mutante é gerado pelo operador MNFD, relacionado ao princípio Operável. E nenhum mutante gerado a partir do operador MLF, do princípio Entendível. Além disso, não foi identificado nenhum mutante natimorto, trivial ou equivalente.

Implicações: Os operadores são de estilo de exclusão e dependem do uso de *code elements* relacionados à acessibilidade. O número de mutantes gerados cresce proporcionalmente ao número de elementos do código de acessibilidade usados na aplicação. Os operadores MTS e

MIT geraram mais mutantes, o que pode indicar que os *code elements* relacionados ao princípio Perceptível são os mais utilizados nas aplicações selecionadas. O conjunto de operadores definido neste trabalho representa uma primeira proposta, e pretende-se melhorar o conjunto com outros tipos de operadores, que por exemplo adicione ou modifique *code elements*. Além disso outros *code elements* e critérios de sucesso podem ser considerados.

Não observou-se nenhum natimorto ou mutante trivial. Isso é importante porque implicam em custo. Esses tipos de mutantes são muito comuns nos testes de mutação da plataforma Android (Linares-Vásquez et al., 2017).

Notou-se a capacidade limitada do Espresso de detectar falhas de acessibilidade e, como consequência, um número reduzido de mutantes foram mortos. Por causa disso, outras ferramentas de teste de acessibilidade devem ser usadas em versões futuras de `AccessibilityMDroid`. Também pretende-se implementar mecanismos para determinar automaticamente os mutantes cobertos. A análise de mutantes é uma desvantagem da maioria das abordagens de teste de mutação para aplicações Android. A grande maioria não oferece uma forma automática de realizar essa tarefa, nem mesmo fornece uma forma de classificar o mutante como morto.

5.3.2 Adequação dos conjuntos de teste existentes

A **Q2** avalia a adequação dos conjuntos de teste em relação aos operadores propostos. A resposta pode fornecer indícios sobre a qualidade dos casos de teste em relação a falhas de acessibilidade e se os desenvolvedores estão preocupados com o teste de tal propriedade não funcional. Para responder a essa pergunta, a Tabela 5.4 traz a porcentagem de mutantes mortos e cobertos por T, por aplicação.

Em média, os conjuntos originais foram capazes de matar apenas 4,9% dos mutantes. A porcentagem de mortos chega a 20% para `Piwigo`, a aplicação com o menor número de mutantes. Mas essa porcentagem é igual a zero para cinco aplicações. A porcentagem de mutantes cobertos é melhor, 32,39% em média. As melhores porcentagens foram obtidas por `AlarmClock` (64,3%) e `Piwigo` (75%). As outras cinco aplicações atingiram um percentual inferior a 40%.

Tabela 5.4: Resultados de adequação de conjuntos de teste originais.

Aplicação	**Morto**	**Coberto**
`AlarmClock`	0,0%	64,3%
`AnyMemo`	0,0%	18,9%
`Authorizer`	0,0%	12,3%
`Equate`	14,3%	0,0%
`Kolabnotes`	0,0%	22,91%
`Piwigo`	20%	75%
`PleesTracker`	0,0%	33,33%
Média	4,9%	32,39%

Resposta à Q2: Os conjuntos de testes existentes das aplicações estudadas mataram ou cobriram apenas uma fração dos mutantes relacionados à acessibilidade.

Implicações: Em geral, existem oportunidades para melhorar a qualidade dos testes de GUI em aplicações móveis. Embora a cobertura de código e o teste de mutação tenham melhor suporte no nível de teste de unidade, é necessário mais suporte de ferramenta no nível de

GUI. Como os mutantes de acessibilidade exigem melhor cobertura de teste em nível de GUI, os resultados apresentados aqui ajudaram a expor essas fraquezas.

5.3.3 Falhas de Acessibilidade

Respondendo à Q2, observa-se que os testes existentes obtiveram uma pequena cobertura de mutantes de acessibilidade, e novos testes são necessários para obter conjuntos de teste adequados. No entanto, é importante saber se tais testes e esforço adicionais contribuem para melhorar a qualidade do teste em termos de falhas de acessibilidade reveladas. A **Q3** visa a responder a essa pergunta.

A Tabela 5.5 mostra o número de falhas de acessibilidade apontadas no *log* gerado pela Espresso quando os conjuntos de teste original (T) e estendido (xT) são executados na aplicação original; a última coluna também mostra a porcentagem de melhoria. Para T, `AlarmClock` tem mais falhas de acessibilidade (126), enquanto `PleesTracker` tem apenas 2 falhas. Em média, tem-se 45,28 falhas de acessibilidade por aplicação. Analisando à adequação do conjunto de teste xT com relação aos mutantes gerados, `Piwigo` tem mais falhas (447) e `PleesTracker` apresentou a melhor porcentagem de melhoria. A menor porcentagem de melhoria foi obtida para `AlarmClock`. Em média, xT revelou 186,4 falhas de acessibilidade. As melhorias variaram de 3,2 a 3.650%.

Tabela 5.5: Falhas de acessibilidade detectadas por T e xT.

Aplicação	**#falhas(T)**	**#falhas(xT)**	**Melhoria**
`AlarmClock`	126	130	3,2%
`AnyMemo`	24	355	1.479%
`Authorizer`	65	201	209,2%
`Equate`	19	27	42,1%
`Kolabnotes`	43	70	62,8%
`Piwigo`	38	447	1.076,3%
`PleesTracker`	2	75	3.650%
Média	45,28	186,4	931,8%

Resposta à Q3: Os conjuntos de teste gerados para cobrir os mutantes contribuem para melhorias significativas no número de falhas de acessibilidade detectadas. Em média, os conjuntos de teste estendidos melhoraram em torno de 932% o número de falhas de acessibilidade reveladas nos conjuntos de teste originais.

Implicações: Os resultados evidenciaram que o uso dos operadores de mutação contribuiu para aumentar o número de falhas de acessibilidade reveladas. Prevê-se que a qualidade do conjunto de testes também seja melhorada, além do ponto de vista da acessibilidade.

5.4 AMEAÇAS À VALIDADE

Existem algumas ameaças à validade no estudo conduzido. As ameaças foram categorizadas a partir da terminologia apresentada por Wohlin et al. (2012).

Validade de Construção. Refere-se ao grau em que a manipulação das métricas em um estudo realmente representa a construção no mundo real (Melo, 2018). Neste contexto, pode ser citado o uso incorreto de testes estatísticos. Sendo assim, como o estudo empírico tinha uma amostra de apenas 7 aplicações, não foi aplicada nenhuma técnica estatística.

Validade Externa. Não é fácil garantir a representatividade das aplicações móveis Android. Além disso, a amostra adotada tem apenas aplicações nativas Android com conjunto de teste implementado com Espresso. Para atenuar essa especificidade, selecionaram-se aplicações do repositório F-Droid, um conjunto diversificado de aplicações de código-aberto com atualizações recentes. F-Droid também foi usado em outros estudos (Mao et al., 2016; Zeng et al., 2016; Gu et al., 2019).

Validade da Conclusão. A estratégia de análise de mutantes está ligada ao *log* construído pela ferramenta Espresso. No entanto, a abordagem do atual estudo também é compatível com outras ferramentas que monitoram a aplicação em execução e produzem *logs* de acessibilidade como MATE (Eler et al., 2018) e a11y (Toff, 2019); planeja-se integrá-los no futuro.

Validade Interna. A determinação dos mutantes cobertos foi executada manualmente e está sujeita a erros. Para minimizar esta ameaça, esta análise foi conduzida cuidadosamente e verificada novamente. Além disso, pode haver erros de implementação em qualquer uma das ferramentas ou rotinas usadas no estudo.

5.5 CONSIDERAÇÕES FINAIS

Neste capítulo, apresentou-se a avaliação da abordagem proposta para o teste de mutação de acessibilidade de aplicações Android. Primeiro, implementaram-se seis dos nove operadores de mutação de acessibilidade definidos no Capítulo 4. Em seguida, para cada aplicação, geraram-se os mutantes. Com base no conjunto de teste original, verificou-se quais mutantes foram mortos ou pelo menos cobertos. Seguindo a abordagem, estendeu-se o conjunto de testes original para cobrir mais mutantes. Os resultados empíricos mostram que os conjuntos de testes originais cobrem apenas uma parte dos mutantes relacionados à acessibilidade. Além disso, os conjuntos estendidos de teste contribuíram para melhorias significativas no número de falhas de acessibilidade detectadas. O próximo capítulo conclui este trabalho de dissertação, levantando as principais contribuições e possíveis trabalhos futuros.

6 CONCLUSÃO

Aplicações móveis fazem parte das atividades cotidianas e desempenham um papel importante para pessoas com algum tipo de deficiência. No entanto, tornar as aplicações mais acessíveis ainda é um desafio para o mercado e a comunidade acadêmica. Desenvolvedores podem utilizar ferramentas de teste de acessibilidade para ajudar nesta tarefa, mas estas apresentam algumas limitações. Elas produzem relatórios sobre falhas de acessibilidade, que geralmente não cobrem toda a aplicação móvel, pois são dependentes do conjunto de teste disponível.

Um conjunto de teste pode ser avaliado e aprimorado com o teste de mutação. Todavia, não foi encontrada na literatura uma abordagem de teste de mutação que trabalhe com aspectos de acessibilidade em aplicações Android (Deng et al., 2015; Linares-Vásquez et al., 2017; Moran et al., 2018; Escobar-Velásquez e Linares-Vásquez, 2019; Liu et al., 2020). Dentro deste contexto, este trabalho define um conjunto de 9 operadores de mutação de acessibilidade Android e introduz `AccessibilityMDroid`, uma abordagem de teste de mutação que inclui geração e análise de mutantes.

A abordagem foi avaliada por meio de uma amostra composta por 7 aplicações Android de código-aberto. Toda aplicação possui em seu projeto um conjunto de teste T. O estudo empírico mostrou que: (i) `AccessibilityMDroid` é aplicável e gerou um total de 257 mutantes; (ii) que os conjuntos T existentes não cobrem a maioria dos mutantes gerados a partir dos operadores de mutação de acessibilidade propostos; e (iii) uma vez que T foi estendido seguindo o fluxo da abordagem, o número de mutantes cobertos aumentou incluindo também o número de falhas de acessibilidade reveladas. Em média, os conjuntos de teste estendidos (xT) melhoraram em torno de 932% o número de falhas de acessibilidades reveladas.

A principal contribuição deste trabalho é a definição do conjunto de 9 operadores de mutação que exploram aspectos de acessibilidade, um requisito não funcional da aplicação. Os operadores de mutação foram definidos a partir de um modelo de defeitos, construído através do mapeamento feito entre os *code elements* e o guia de acessibilidade WCAG. Além de apresentar e implementar o conjunto de operadores de mutação, desenvolveu-se um fluxo para geração e análise dos mutantes. A análise consiste na comparação do *log* de acessibilidade produzido por T, um conjunto de teste presente no projeto e implementado com a Espresso. Em resumo este trabalho contribui para o teste de acessibilidade de aplicações Android e também propõe um processo para:

- Auxiliar os testadores no processo de melhoria contínua de seus conjuntos de teste;
- Avaliar o conjunto de teste do projeto sob uma perspectiva de acessibilidade;
- Avaliar as ferramentas de teste de acessibilidade;
- Auxiliar os desenvolvedores a alcançarem uma aplicação Android mais acessível.

6.1 CONTRIBUIÇÃO ACADÊMICA

Ao longo do desenvolvimento deste trabalho alguns aspectos da pesquisa foram submetidos à avaliação da comunidade acadêmica. Como resultado estão as seguintes publicações:

1. SILVA, H. N.; FARAH, P. R.; MENDONÇA, W. D. F.; VERGILIO, S. R. Assessing Android Test Data Generation Tools via Mutation Testing. Em: the IV Brazilian

Symposium, 2019, Salvador. Proceedings of the IV Brazilian Symposium on Systematic and Automated Software Testing - SAST 2019.

2. SILVA, H. N.; ENDO, A. T.; ELER, M. M.; VERGILIO, S. R.; DURELLI, V. On the Relation between Code Elements and Accessibility Issues in Android Apps. Em: V Brazilian Symposium on Systematic and Automated Software Testing - SAST 2020.

Trabalhos submetidos e em processo de avaliação:

1. SILVA, H. N.; LIMA, J. P.; VERGILIO, S. R; ENDO, A. T. A mapping study on mutation testing for mobile applications. Submetido para Software Testing Verification and Realibilty (STVR).

2. SILVA, H. N.; VERGILIO, S. R; ENDO, A. T. Mutation Operators for Accessibility Testing of Android Apps. Submetido para Mutation 2021.

Tendo como objetivo a replicabilidade do estudo e o fomento por discussões na academia, o ambiente experimental está disponível no Repositório OSF[1]. Nele encontram-se o conjunto de aplicações utilizado e as rotinas (*shell script*) para execução completa do fluxo da abordagem: execução de T na aplicação original, chamada da geração de mutantes, execução de T em cada mutante, relatório final de escore por operador de mutação e mutante gerado.

6.2 LIMITAÇÕES E TRABALHOS FUTUROS

A abordagem `AccessibilityMDroid` ainda não abrange de forma significativa a variedade de aplicações móveis Android. As aplicações utilizadas no estudo empírico, além de serem nativas e de código-aberto, necessariamente continham em seus projetos um conjunto de teste implementado com Espresso. A estratégia utilizada na abordagem para verificar a morte do mutante depende do conjunto de recomendações de acessibilidade checadas pela Espresso. Ao longo da execução dos conjuntos de testes, foi possível notar que alguns Operadores de Mutação geraram violações de acessibilidade em que a Espresso não foi capaz de detectar.

Como trabalho futuro, planeja-se abrir *issues* dos problemas de acessibilidade encontrados no experimento de validação. Também propor como melhoria para a ferramenta Espresso a ampliação das checagens de acessibilidade, uma vez que seu *log* não foi capaz de distinguir a diferença de inúmeros mutantes gerados pelos operadores de mutação. `AccessibilityMDroid` ainda pode ser aprimorada e estendida, podendo:

- Trabalhar com aplicações em extensão APK, e assim incluir na avaliação aplicações comerciais (código-fechado);
- Experimentar, além da Espresso, diferentes oráculos, por exemplo, o oráculo implementado pela MATE (Eler et al., 2018);
- Ampliar o conjunto de operadores de mutação de acessibilidade, agora focados em incluir e alterar *code elements*.

[1]Disponível em: `https://osf.io/vfs2d/?view_only=6c3af7cdbb7f4132a9367e196735c68f`

REFERÊNCIAS

Ableson, F., Sen, R., King, C. e Ortiz, C. E. (2012). *Android in action*. Manning Publications Co., 3rd edition.

Acosta-Vargas, P., Salvador-Ullauri, L., Jadán-Guerrero, J., Guevara, C., Sanchez-Gordon, S., Calle-Jimenez, T., Lara-Alvarez, P., Medina, A. e Nunes, I. L. (2020). Accessibility assessment in mobile applications for android. Em Nunes, I. L., editor, *Advances in Human Factors and Systems Interaction*, páginas 279–288, Cham. Springer International Publishing.

Acosta-Vargas, P., Salvador-Ullauri, L., Pérez-Medina, J. L., Zalakeviciute, R. e Hernandez, W. (2019). Heuristic Method of Evaluating Accessibility of Mobile in Selected Applications for Air Quality Monitoring. Em *International Conference on Applied Human Factors and Ergonomics*, páginas 485–495. Springer.

Agrawal, H., Demillo, R., Hathaway, B., Hsu, W., Hsu, W., Krauser, E., J. Martin, R., Mathur, A. e Spafford, E. (1999). Design Of Mutant Operators For The C programming language.

Alshayban, A., Ahmed, I. e Malek, S. (2020). Accessibility Issues in Android Apps: State of Affairs, Sentiments, and Ways Forward. Em *International Conference on Software Engineering (ICSE)*, volume 12. ACM.

Ammann, P. e Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press.

Andrews, J. H., Briand, L. C. e Labiche, Y. (2005). Is mutation an appropriate tool for testing experiments? Em *Proceedings of the 27th international conference on Software engineering*, páginas 402–411. ACM.

Ballantyne, M., Jha, A., Jacobsen, A., Hawker, J. S. e El-Glaly, Y. N. (2018). Study of Accessibility Guidelines of Mobile Applications. Em *Proceedings of the 17th International Conference on Mobile and Ubiquitous Multimedia*, páginas 305–315. ACM.

Cisco (2018). Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2018–2023 White Paper - Cisco. `https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-738429.html`. (Acesso: 04/05/2020).

Coelho, R., Almeida, L., Gousios, G. e van Deursen, A. (2015). Unveiling exception handling bug hazards in Android based on Github and Google code issues. Em *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, páginas 134–145. IEEE.

Damaceno, R. J. P., Braga, J. C. e Mena-Chalco, J. P. (2018). Mobile device accessibility for the visually impaired: problems mapping and recommendations. *Universal Access in the Information Society*, 17(2):421–435.

Deitel, P., Deitel, H., Deitel, A. e Morgano, M. (2015). *Android para Programadores: uma abordagem baseada em aplicativos*. Editora Bookman, 3th edition.

Delamaro, M., Do, S., Souza, S., Maldonado, J., Pinto, S., Fabbri, F., Auri, M., Vincenzi, A., Barbosa, E. e Jino, M. (2017). *Introdução ao teste de software*. Elsevier Brasil, 1st edition.

Delamaro, M. E., Offutt, J. e Ammann, P. (2014). Designing deletion mutation operators. Em *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, páginas 11–20.

DeMillo, R. A., Lipton, R. J. e Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. volume 11, páginas 34–41.

Deng, L., Mirzaei, N., Ammann, P. e Offutt, J. (2015). Towards mutation analysis of Android apps. Em *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, páginas 1–10.

Deng, L. e Offutt, J. (2018). Experimental Evaluation of Redundancy in Android Mutation Testing. *International Journal of Software Engineering and Knowledge Engineering*, 28(11n12):1597–1618.

Eler, M. M., Rojas, J. M., Ge, Y. e Fraser, G. (2018). Automated Accessibility Testing of Mobile Apps. Em *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, páginas 116–126.

Endo, A. T. (2013). *Model based testing of service oriented applications*. Tese de doutorado, Universidade de São Paulo.

Escobar-Velásquez, Camilo, O.-R. M. e Linares-Vásquez, M. (2019). MutAPK: Source-Codeless Mutant Generation for Android Apps. Em *2019 IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Fischer, J. F. (2015). Uma análise da eficácia de assertivas executáveis como observadora de falhas em software. Dissertação de Mestrado, Pontifícia Universidade Católica do Rio de Janeiro.

Gamma, E. e Beck, K. (2019). The new major version of the programmer-friendly testing framework for Java. `https://junit.org`. (Acesso: 12-10-2020).

Google (2019a). Arquitetura da plataforma. `https://developer.android.com/guide/platform?hl=pt-PT`. (Acesso: 09/11/2020).

Google (2019b). Espresso. `https://developer.android.com/training/testing/espresso`. (Acesso: 12-10-2020).

Google (2019c). Melhorar seu código com verificações de lint. `https://developer.android.com/studio/write/lint?hl=pt-BR`. (Acesso: 12-10-2020).

Google (2019d). Scanner de acessibilidade. `https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor&hl=pt_BR`. (Acesso: 12-10-2020).

Grønli, T.-M., Hansen, J., Ghinea, G. e Younas, M. (2014). Mobile application platform heterogeneity: Android vs Windows Phone vs iOS vs Firefox OS. Em *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, páginas 635–641. IEEE.

Gu, T., Sun, C., Ma, X., Cao, C., Xu, C., Yao, Y., Zhang, Q., Lu, J. e Su, Z. (2019). Practical GUI Testing of Android Applications via Model Abstraction and Refinement. Em *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, página 269–280. IEEE Press.

Hartley, S. D. (2011). World Report on Disability (WHO). Relatório técnico, WHO and World Bank.

Iqbal, M. (2020). App Download and Usage Statistics (2020). `https://www.businessofapps.com/data/app-statistics/#2.1`. (Acesso: 12-11-2020).

ISO/IEC 25010 (2011). ISO/IEC 25010:2011, systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models.

Jabbarvand, R. e Malek, S. (2017). μDroid: an energy-aware mutation testing framework for Android. Em *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, páginas 208–219. ACM.

Jia, Y. e Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.*, 37(5):649–678.

Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R. e Fraser, G. (2014). Are Mutants a Valid Substitute for Real Faults in Software Testing? Em *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, página 654–665, New York, NY, USA. Association for Computing Machinery.

Kim, S., Clark, J. e McDermid, J. (1999). The rigorous generation of Java mutation operators using HAZOP. *Informe técnico, The University of York*.

Kirkpatrick, A., Connor, J. O., Campbell, A. e Cooper, M. (2018). Web Content Accessibility Guidelines (WCAG) 2.1. `https://www.w3.org/TR/WCAG21/`. (Acesso: 12-10-2020).

Lecheta, R. R. (2015). *Google Android - Aprenda a criar aplicações para dispositivos móveis com o Android SDK*. Novatec Editora, 5th edition.

Li, N. e Offutt, J. (2016). Test oracle strategies for model-based testing. *IEEE Transactions on Software Engineering*, 43(4):372–395.

Linares-Vásquez, M., Bavota, G., Tufano, M., Moran, K., Di Penta, M., Vendome, C., Bernal-Cárdenas, C. e Poshyvanyk, D. (2017). Enabling Mutation Testing for Android Apps. Em *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, páginas 233–244, New York, NY, USA. ACM.

Lisper, B., Lindstrom, B., Potena, P., Saadatmand, M. e Bohlin, M. (2017). Targeted mutation: Efficient mutation analysis for testing non-functional properties. Em *Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation Workshops, (ICSTW)*, páginas 65–68.

Liu, J., Xiao, C. X., Xu, L., Dou, L., Podgurski, C. A., Xiao, X. e Podgurski, A. (2020). DroidMutator: An Effective Mutation Analysis Tool for Android Applications. Em *Proceedings of the 42nd International Conference on Software Engineering Companion*, ICSE-Companion. ACM.

Luna, E. e El Ariss, O. (2018). Edroid: A Mutation Tool for Android Apps. Em *2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT)*, páginas 99–108. IEEE.

Ma, Y.-S., Offutt, J. e Kwon, Y. R. (2005). MuJava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133.

Machado, D. R., Machado, R. P. e Conforto, D. (2014). Dispositivos móveis e usuários cegos: recomendações de acessibilidade em discussão. *Nuevas Ideas en Informática Educativa TISE*.

Maldonado, J. C. M. e Jino, M. (1991). *Critérios potenciais usos: Uma contribuição ao teste estrutural de software*. Tese de doutorado, Unicamp.

Mao, K., Harman, M. e Jia, Y. (2016). Sapienz: Multi-objective automated testing for android applications. Em *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, página 94–105, New York, NY, USA. Association for Computing Machinery.

Mednieks, Z., Laird Dornin, G. e Nakamura, M. (2012). *Programando o Android*. São Paulo: Novatec, 3rd edition.

Melo, S. M. (2018). *Um framework para avaliação sistemática de técnicas de teste no contexto de programação concorrente*. Tese de doutorado, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, SP. 207 pgs.

Miller, E. (1977). The philosophy of testing in program testing techniques. páginas 1–3. IEEE Computer Society Press.

Moher, D., Liberati, A., Tetzlaff, J. e Altman, D. G. (2009). Preferred Reporting Items for Systematic Reviews and Meta-Analyses: The PRISMA Statement. *BMJ*, 339.

Moran, K., Tufano, M., Bernal-Cárdenas, C., Linares-Vásquez, M., Bavota, G., Vendome, C., Di Penta, M. e Poshyvanyk, D. (2018). Mdroid+: A mutation testing framework for Android. Em *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, páginas 33–36. ACM.

Myers, G. J., Sandler, C. e Badgett, T. (2011). *The art of software testing*. John Wiley & Sons.

Papadakis, M., Shin, D., Yoo, S. e Bae, D.-H. (2018). Are Mutation Scores Correlated with Real Fault Detection? A Large Scale Empirical Study on the Relationship between Mutants and Real Faults. Em *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, página 537–548, New York, NY, USA. Association for Computing Machinery.

Pressman, R. e Maxim, B. (2016). *Engenharia de Software*. McGraw Hill Brasil, 8th edition.

Reda, R. (2019). GitHub - RobotiumTech/robotium: Android UI Testing. `https://github.com/RobotiumTech/robotium`. (Acesso: 12-10-2020).

Salihu, I.-A., Ibrahim, R., Ahmed, B. S., Zamli, K. Z. e Usman, A. (2019). AMOGA: A Static-Dynamic Model Generation Strategy for Mobile Apps Testing. *IEEE Access*, 7:17158–17173.

Silva, C., Eler, M. M. e Fraser, G. (2018). A survey on the tool support for the automatic evaluation of mobile accessibility. Em *Proceedings of the 8th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-Exclusion*, DSAI 2018, página 286–293. ACM.

Silva, H. N., Endo, A. T., Eler, M. M., Vergilio, S. R. e Durelli, V. H. S. (2020). On the relation between code elements and accessibility issues in android apps. Em *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*, SAST 20, página 40–49, New York, NY, USA. Association for Computing Machinery.

Souza Neto, J. B. (2019). *Uma abordagem para Teste de Mutacão de programas de processamento de Big Data*. Tese de doutorado, Universidade Federal do Rio Grande do Norte.

Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y. e Su, Z. (2017). Guided, stochastic model-based gui testing of android apps. Em *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, página 245–256, New York, NY, USA. Association for Computing Machinery.

Tian, J. (2005). *Software quality engineering: testing, quality assurance, and quantifiable improvement*. John Wiley & Sons.

Toff, D. (2019). A11y ally. `https://github.com/quittle/a11y-ally`. (Acesso: 12-09-2020).

Vendome, C., Solano, D., Liñán, S. e Linares-Vásquez, M. (2019). Can Everyone use my app? An Empirical Study on Accessibility in Android Apps. Em *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, páginas 41–52.

Wasserman, A. I. (2010). Software engineering issues for mobile application development. Em *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, páginas 397–400.

Wei, Y. (2015). MuDroid: Mutation testing for Android apps. *Univ. College London, London, UK, Tech. Rep*.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B. e Wesslén, A. (2012). *Experimentation in Software Engineering*, páginas 123–151.

Zeng, X., Li, D., Zheng, W., Xia, F., Deng, Y., Lam, W., Yang, W. e Xie, T. (2016). Automated test input generation for android: Are we really there yet in an industrial case? Em *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, página 987–992.

Zhang, P. e Elbaum, S. (2012). Amplifying tests to validate exception handling code. Em *Proceedings of the 34th International Conference on Software Engineering*, páginas 595–605.

APÊNDICE A – A MAPPING STUDY ON MUTATION TESTING FOR MOBILE APPLICATIONS

Este apêndice inclui o mapeamento da literatura sobre teste de mutação em Aplicações Android.

A mapping study on mutation testing for mobile applications

Henrique Neves da Silva
Federal University of Paraná
Curitiba, PR, Brazil
henriqueneves@ufpr.br

Jackson Antonio do Prado Lima
Federal University of Paraná
Curitiba, PR, Brazil
japlima@ufpr.br

Andre Takeshi Endo
Universidade Tecnológica Federal do Paraná
Cornélio Procópio, PR, Brazil
andreendo@utfpr.edu.br

Silvia Regina Vergilio
DInf, Federal University of Paraná
Curitiba, PR, Brazil
silvia@inf.ufpr.br

ABSTRACT

The use of mutation testing for mobile applications (apps for short) is still a challenge. Mobile apps are usually event-driven and encompass graphical user interfaces and a complex execution environment. Then, they require mutant operators to describe specific apps faults, and the automation of the mutation process phases like execution and analysis of the mutants is not an easy task.

To encourage research addressing such challenges, this paper presents results from a mapping study on mutation testing for mobile apps.

Following a systematic plan, we found 16 primary studies that were analyzed according to three aspects: *(i)* basic information of the field, *(ii)* study characteristics such as focus, proposed operators and automated support for the mutation testing phases; and *(iii)* evaluation aspects.

The great majority of studies (98%) have been published in the last three years. The most addressed language is Java, and Android is the only operating system considered. Mutant operators of GUI and Configuration types are prevalent, in a total of 138 operators found. Most studies implement a supporting tool, but few tools support mutant execution and analysis. The evaluation conducted by the studies includes apps mainly from the finance and utilities domain. Nevertheless, there is a lack of benchmark and more rigorous experiments.

Future research should address other specific types of faults, languages, and operating systems. They should offer support for mutant execution and analysis, as well as to reduce the mutation testing cost and limitations in the mobile context.

KEYWORDS

Fault-based Testing; Android; Mutation operators.

1 INTRODUCTION

A mobile application (*mobile app*, for short) is a program that runs on a mobile device such as a smartphone or tablet. These devices are present in large part of everyday activities, not only for communicating, messaging, or leisure tasks in general, but also as a work and/or study tool. As a consequence, the number of mobile apps has drastically increased over the last years. However, developing these apps with quality is still a challenge. Many of them reach the market containing significant bugs that often lead to failures and unsatisfied users [PS4]. This is maybe due to the characteristics of these apps. They are event-driven, involve rich graphical user interfaces (GUIs), and interact in a complex environment that includes users, devices, sensors, and other apps [1]. Besides, mobile apps are usually developed under time-to-market pressure without (or little) employing software engineering and testing practices [2].

We can then conclude that testing techniques and tools to deal with mobile apps' particularities are fundamental. Many techniques have been proposed, and mobile app testing is an active research topic [2–4]. Existing literature has been surveyed by some secondary studies [5–7]. Most of the techniques and tools are focused on modeling and testing of the GUI interaction, such as GUITAR [8], Stoat [2, 9], iMPAcT [10, 11], AMOGA [PS14], and APE [4]. While targeting common user interaction scenarios has shown to be a promising approach, other subtle bugs related to specific features, uncovered code, and less executed events can be missed.

To deal with this limitation, some researchers have advocated that the use of mutation testing for mobile apps can allow developers to test and find specific bugs [PS1]. This is motivated by the fact that mutation testing has been found to be effective in different domains, contributing to improving the test efficacy in terms of revealed faults [12, 13]. For a program under test P, mutation testing generates a set of mutants M by applying small changes in P. Each change is performed by a mutation operator, which is a transformation rule describing common faults that can be present in P. Then test cases are evaluated to kill the mutants, that is, to check whether output in a mutant m is different from the output of P. In the end, the mutants are analyzed, and the mutation score is calculated, given by the ratio between the number of killed mutants and the total of non-equivalent mutants. A mutant m is equivalent to P if there is no test capable of distinguishing them, that is, m and P produce the same output for all inputs. The mutation score can be used as a measure to evaluate a test set T, as well as an indicator to know if the program has been tested enough.

To apply mutation testing for mobile apps, some challenges need to be addressed. First, it is necessary to characterize the bugs such apps exhibit and shape this knowledge in the form of operators for a mutant generation. This is a challenging task because, as mentioned, the environments in which the apps operate are heterogeneous and complex. Another challenge regards the phases of mutant execution and analysis, which are also impacted in the context of mobile app development. The research on the Mutation Testing of Apps poses some difficulties. Such a subject has raised interest, and mutation approaches have been proposed in the literature. However, we have

not found works reporting the characteristics of such approaches, fault-models that serve as a basis to propose the operators, how they differ from traditional mutation testing, and how they have been implemented and evaluated.

To provide such a general overview, it is important to identify challenges faced, gaps, and trends to guide the research on this subject. Motivated by these facts and the importance and popularity of mutation testing, this paper presents the results of a mapping study on mutation testing of mobile apps. To this end, we adopted the process of Petersen et al. [14], resulting in a protocol with research questions, inclusion and exclusion criteria, construction of the search string, and selection of known search databases. We found 16 primary studies that are analyzed according to three aspects: *(i) basic information of the field*, such as evolution along the years, main research groups and publication fora; *(ii) characteristics of the studies*: study focus and type, kind of operators, mutation analysis support and tools, addressed particularities such as platforms and operating systems; and *(iii) evaluation aspects*: such as target systems and baselines used for comparison. In addition to this, some research trends and gaps were identified that allow researchers to guide future investigations.

The results show an increasing number of papers on this research subject. The great majority of studies (98% of them) have been published in the last three years. The most addressed language is Java, and Android is the only operating system considered. Mutant operators of GUI and Configuration types are prevalent, in a total of 138 operators found. Most studies implement a supporting tool, but few tools support mutant execution and analysis (score calculation). The evaluation conducted by the studies includes apps mainly from the finance and tools domain. Nevertheless, other domains like games, multimedia, and communication are also common. We observe a lack of benchmark and more rigorous experiments and some difficulties to allow replicability.

In this way, the contribution of this work is twofold: *(i)* to present the main characteristics of mutation testing studies according to a classification schema generated interactively during the analysis of the studies found; and *(ii)* to help researchers in the identification of research opportunities and encourage new works on this subject to address existing challenges.

The remainder of this paper is organized as follows. Section 2 provides an overview of related work. Section 3 describes how the map was conducted. Section 4 presents and analyses the main findings. Section 5 points out trends and identifies research opportunities. Section 6 discusses the threats to validity of our results, and Section 7 concludes.

2 RELATED WORK

In the literature, we find some mappings, surveys, and systematic reviews on the two fields related to this work: *(i)* mutation testing, and *(ii)* mobile app testing; they are described as follows. There are several publications reviewing mutation testing. Jia and Harman [12] provide a comprehensive survey of trends and results on the field. Their work covers topics such as theories, optimization techniques, equivalent mutant detection, empirical studies, and mutation tools. Their analysis reveals an increasingly practical trend in this subject. Similarly, Papadakis et al. [13] survey the recent advances in mutation testing. This study concludes that the main open problem is the detection of the equivalent and redundant mutants.

Other systematic studies focus on specific topics of mutation testing. For instance, Madeyski et al. [15] describe a systematic review encompassing 22 studies to obtain a complete list of existing solutions for detection of equivalent mutants. The systematic review of Silva et al. [16] involves 263 papers that explore search-based techniques in the context of Mutation Testing. These techniques have been applied to optimize some mutation testing tasks, such as test data generation, mutant generation, and selection of effective and non-redundant mutation operators. The mapping of Souza et al. [17] reports studies on test data generation for mutation testing. The authors observed that the main challenge is to efficiently generate test cases based on mutants; this is still a research gap. They identified 19 studies, applying ten different test data generation techniques, and only a few of them provide a comprehensive evaluation, including industrial-scale programs.

Prado Lima and Vergilio [18] present results from mapping on Higher Order Mutation Testing (HOMT). A Higher-Order Mutant (HOM) is generated by two or more changes in the program being tested. They summarize characteristics of HOMT approaches, HOM generation strategies, evaluation aspects, trends, and research opportunities. Pizzoleto et al. [19] performed a systematic review on cost reduction for mutation testing. They selected 175 peer-reviewed studies, from which 153 present either original or updated contributions. The review points out six main goals for cost reduction and 21 techniques.

Regarding the test of mobile apps, we also find some secondary studies. The work of Mendez-Porras et al. [20] reviews 83 studies on automated testing of mobile apps, testing techniques, and empirical assessments. They conclude that the number of proposals for automated testing of mobile apps has increased in recent years. The main approaches identified were model-based testing (30%), capture/replay (15.5%), model-learning testing (10%), systematic testing (7.5%), fuzz testing (7.5%), random testing (5%), and scripted-based testing (2.5%).

The study of Zein et al. [5] also focuses on mobile app testing techniques and challenges. The 79 selected studies were mapped to a classification schema. Several research gaps are identified, and specific key testing issues for practitioners are discussed: there is a need for eliciting testing requirements early in the development process; a need to conduct research in real-world development environments; specific testing techniques targeting application lifecycle conformance and mobile services testing; and comparative studies for security and usability testing.

The review of Kong et al. [7] highlights the main trends, pinpoint the main methodologies, and enumerate the challenges faced by the Android testing approaches, as well as the directions where the community effort is still needed. From 103 reviewed papers, the authors proposed a taxonomy that explores several dimensions, including the objectives (i.e., what functional or nonfunctional concerns are addressed by the approaches) that were pursued and the techniques (i.e., what type of testing methods model-based, concolic, etc.) that were adopted.

The mapping of Tramontana et al. [6] includes 131 papers based on automated functional testing of mobile apps. The papers were

classified on the basis of the supported testing activities, the characteristics of the techniques and tools, and the evaluation methodologies adopted to validate them. The results show a substantial prevalence of Android-based approaches, a lack of contributions from industry, and the absence of specific venues and journals focused on mobile testing automation.

By analyzing the works mentioned previously, all surveys and reviews on mobile apps testing have been published recently, showing that this is a current research subject. Even among the secondary studies on mutation testing, we find recent works, justifying the importance and interest in this technique. Nevertheless, we observe that none of the mentioned works targets the intersection of both fields. Papadakis et al. [13] mention some tools for Android apps, but the particular characteristics of such tools are not discussed. Only a general analysis as other traditional mutation testing tools is provided. Works on testing of mobile apps do not focus on mutation testing.

Our work has a different focus, which is mutation testing for mobile apps. Our mapping offers characteristics of mutation operators and tools, which are specific for this kind of application. Moreover, this work presents challenges, research gaps, and trends regarding mobile apps, which can help the researchers in future investigations. We cannot find such results in the works most related to ours, which map the general areas, or that deal with mutation testing in other contexts.

3 MAPPING PROCESS

In this section, we describe the main steps of our mapping, conducted following the guidelines of Petersen et al. [14]. First of all, and to justify our work, we conducted a search for related work with the following string: *(mutation OR mutant OR fault-based) AND (test OR testing OR analysis) AND (app OR android OR mobile OR "mobile application") AND (map OR mapping OR review OR survey)*. We did not find any work with the same goal of ours (see Section 2 for the related work). Given this fact, which justifies and shows the need and relevance of our mapping, we performed the steps described next.

3.1 Definition of Research Questions

To overview existing research on the Mutation Testing of Apps, we used three groups of research questions. The first one provides basic information about the field. The second group characterizes studies and their proposals. Finally, the third group refers to the aspects of the evaluation/validation conducted in the found studies. The research questions from each group are presented below.

RQ1: Basic information of the field

RQ1.1: *In which fora is the research on Mutation Testing of Apps published?* This question allows the identification of the target public and the main fora, where research on the Mutation Testing of Apps can be found and published.

RQ1.2: *How have the number and frequency of publications evolved over the years?* This question aims to analyze the interest in the Mutation Testing of Apps over the years and to assess how relevant and active this topic is.

RQ1.3: *What are the main research groups?* This question aims to identify the main authors and research groups, as well as corresponding institutions and countries.

RQ2: Characteristics of studies

RQ2.1: *What is the main focus of the found studies?* This question aims to identify the main goals of the primary study regarding mutation testing. For instance, to identify studies that propose or evaluate/validate some solution.

RQ2.2: *What types of mutation operators are proposed?* This question aims to classify the most common types of mutation operators defined for mobile apps. We also address the platform, programming languages, and artifacts targeted by such operators. This question intends to identify possible research gaps and to point the need for future work.

RQ2.3: *What are the supporting tools?* This question aims to identify the existing tools that support the operators proposed and phases of mutation testing.

RQ3: Evaluation Aspects

RQ3.1: *How have the proposals been evaluated?* This question analyzes different aspects of the evaluation/validation conducted in the studies. We analyze used systems, baseline approach used for comparison, and threats to validity.

3.2 Definition of the Search String

We formulated our search string, considering the mapping goals, and posed research questions. The resulting search terms were composed of synonymous for the main terms "Mutation Testing" and "Mobile Apps", taking into account different spelling. Then, we constructed the search string using logical operators, "OR" and "AND". After tests, we adopted the following search string that returned the most significant number of relevant articles[1].

> (mutation **OR** mutant **OR** fault-based) **AND** (test **OR** testing **OR** analysis) **AND** (apps **OR** "mobile application" **OR** android)

To verify the accuracy of the keywords chosen to compose the search string, we used a control group. Such a group comprises a set of previously known studies, presented in Table 1 and the number of citations extracted from Google Scholar in February 2020.

3.3 Selection criteria

Table 2 shows the adopted inclusion and exclusion criteria. In summary, we screened peer-reviewed studies written in English, available online, and verified whether each one fits our mapping goals.

3.4 Conducting the review

The search started and finished in February 2020. The review was conducted in a set of steps, presented in Figure 1 following the PRISMA (Preferred Reporting Items for Systematic reviews and

[1]We included the word android since it is recognized by many authors as the main mobile operating system [PS4].

Table 1: Control group.

Year	Authors	Title	Citations
2015	Deng et al. [PS1]	Towards mutation analysis of Android apps	35
2017	Jabbarvand and Malek [PS2]	μDroid: an energy-aware mutation testing framework for Android	28
2017	Linares-Vásquez et al. [PS3]	Enabling Mutation Testing for Android Apps	36
2017	Deng et al. [PS4]	Mutation operators for testing Android apps	54
2018	Linares-Vásquez et al. [PS7]	MDroid+: A mutation testing framework for Android	12

Table 2: Inclusion and exclusion criteria.

Inclusion Criteria	
I1	The paper is related to Software Engineering area;
I2	The paper is related to Mutation Testing for Mobile Apps.
Exclusion Criteria	
E1	Out of scope, not related to Mutation Testing for Mobile Apps;
E2	Not available online;
E3	Not in English;
E4	Abstracts, posters, reviews, conference reviews, chapters, thesis, keynotes, and doctoral symposiums;
E5	Non-peer reviewed publication.

Meta-Analyses) statement [21]. We performed a systematic search for grey literature in databases like AWS Whitepapers & Guides, Google AI, Google Cloud, Google Scholar, Microsoft Academic, and Microsoft Research; yet, we did not find any study. Thus, we maintained this study as a systematic map rather than a multivocal literature review [22, 23].

First, we selected an initial set of studies using the search string in the digital libraries, considering the title, abstract, and keywords. The engines for performing the search are online repositories chosen due to their popularity and because they provide many leading software engineering publications. However, some of them required adaptations, such as searching in title, abstract, and keywords individually. We did not define an initial publication date for the studies, and then all the returned papers were included. Table 3 shows the data sources used and the period covered, with the data separated by Digital Libraries. In total, we performed the mapping in 8 data sources. As observed, a total of 2343 studies were found, including the period from 1884 to 2020.

In the second step, we removed repeated studies, remaining 2094 ones. Then we applied the selection criteria and obtained 185 studies. The selection was conducted by the first and second authors, who read the title, abstract, and index terms of the papers (keywords). In case of doubts, reading in the following order was performed: introduction, conclusion, and the entire paper. If such a doubt persists, meetings, and discussions with the other authors were performed to solve it. Other two authors also revised the final set of selected studies. Next, we performed a snowballing reading (forward and backward snowballing procedures, following Wohlin's guidelines [24]), using the control group as a starting set. In this step, we found two papers. In the end, 16 papers remained.

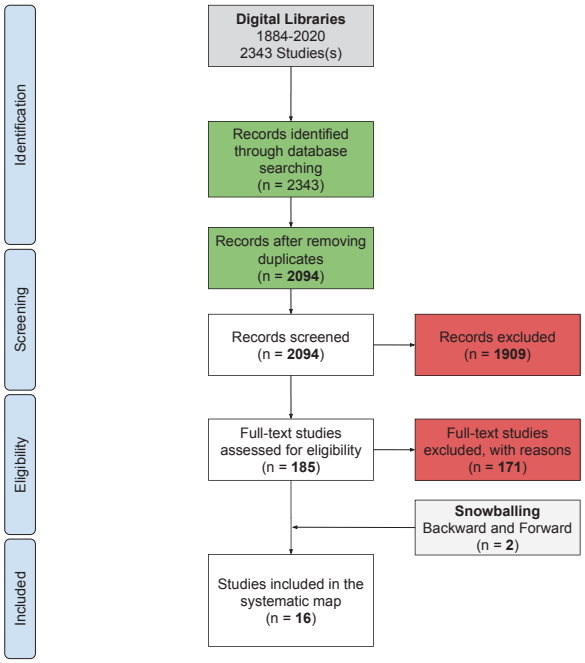


Figure 1: PRISMA flow diagram. For each data source, the number of studies found is presented, as well as the range of publication data from these studies.

The references for the selected papers are presented in the *Primary Studies (PS)* Section.

Table 3: Number of papers returned by each source.

Source	Studies	Period Covered
Digital Libraries		
ACM	929	1985-2020
EBSCOhost	388	1988-2020
Ei Compendex	194	1884-2020
IEEEXplore	24	2013-2019
ScienceDirect	371	1996-2020
Scopus	106	1996-2019
Springer Link	159	1993-2020
Wiley Online Library	172	1988-2020
Snowballing		
Forward and Backward procedures	2	2020-2020
Total	2345	1884-2020

3.5 Classification scheme, data extraction and dissemination

We collected the following information to address the research questions: title, authors, institution, publication venue, publication year, and depending on the study type (proposal of solution, validation research, or experimental research) other data were collected such as all the characteristics of the proposal and conducted the evaluation. The raw data were analyzed and are available at Open Science Framework (OSF) [25].

To avoid the researchers' bias threat during the extraction, which can negatively affect the results, we use a classification scheme derived interactively, which encompasses different dimensions to classify the studies found and answer each research question. Table 4 presents the schema with a brief description of each category.

First, we look at the main focus that can be primary, that is, the study has mutation testing as the main focus and the secondary, case where the mutation testing is used only for evaluation purposes. After this, we also identified the type of research conducted in the study using the classification proposed by Wieringa et al. [26][2]. These categories show if the study proposes a novel solution to the mutation testing problem, and/or conducts some validation/experiment. Then we took the studies proposing mutation operators and analyzed the classification of operators proposed by the authors. Then we grouped similar classifications, and as a result, we established eight categories, used to answer RQ2.2. We also identified when a tool was developed to support some of the mutation testing phases (RQ2.3). To this end, we adopted the common phases of mutation testing described in literature [12].

To answer RQ3, we took the studies that conducted an evaluation or validation and collected information about the used mobile apps, threats, and baseline approaches. Regarding the apps used in such studies, we extracted the domain they belong considering the information provided by the authors. If this information was not available, we used the Google Play's or F-Droid's classification. Based on the original classification collected, we grouped them into 11 categories.

[2]Table 4 describes only the three types found in our study.

4 OUTCOMES

This section presents the results and answers to the research questions.

4.1 Basic information of the field

Answers to RQ1 are related to some basic information of the field and are presented in the next sub-sections.

4.1.1 RQ1.1 – Publication fora. We observed that 12 papers (75%) were published in events (conferences, symposiums, and workshops), and 4 papers in journals. The preferred journals are Information and Software Technology (IST), International Journal of Software Engineering and Knowledge Engineering (IJSEKE), International Journal on Advanced Science, Engineering and Information Technology (IJASEIT), and IEEE Access. Among the events with more than one paper, they are: European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), International Conference on Software Engineering (ICSE), and International Conference on Software Testing, Verification and Validation Workshops (ICSTW). Figure 2 shows the distribution of main venues: journals and events with more than two papers. The preferred fora is the ICSTW with three papers. Notice that the venues are all related to the general area of Software Engineering and, specifically, with software testing.

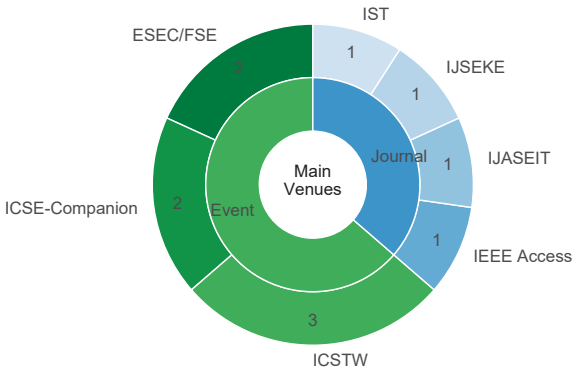


Figure 2: Main publication fora.

4.1.2 RQ1.2 – Publications over the years. The number of publications over the years is depicted in Figure 3 together with a trend line. This figure also distinguishes the venue (Journal or Event). The first paper appeared in 2015, and since then, the interest in the Mutation Testing of Apps has been increasing, as evinced by the trend line. The most significant number of publications were published in 2017, with 5 out of 16 studies (≈ 31%), without considering the search ended in February 2020, and that some studies of this year may not be included. Fifteen studies (98%) were published in the period of 2017-2020.

4.1.3 RQ1.3 – Research groups. We found 45 authors from 25 different institutions. The main research groups are from George Mason University (5 papers), Universidad de Los Andes (3 papers), and with 2 papers: Università Della Svizzera Italiana, College of William and

Table 4: Classification schema.

Dimension/Category	Description
Focus of the Study – RQ2.1	
Primary	Main contribution is related to mutation testing of mobile apps.
Secondary	Mutation testing is used only to support the evaluation.
Type of the Study – RQ2.1	
Proposal of solution	Proposes a solution and argues for its relevance. The contributions must be novel or at least a significant improvement of an existing technique.
Evaluation research	Investigates a problem or an implementation of technique(s) in practice.
Validation research	Investigates a solution proposal that has not yet been implemented in practice, proposed by the author(s) or someone else.
Type of operator – RQ2.2	
Configuration	Operators that modify attributes or parameters that configure the operation of a mobile app.
Connectivity	Operators that interfere with the app's communication, like Bluetooth, WiFi, and HTTP requests.
GUI	Operators that mutate GUI elements and their event handlers, life cycle, and navigation.
Intent	Operators that introduce changes related to the Android-specific component Intent, used mostly for communication among apps.
Location	Operators that inject geolocation related faults (GPS).
Persistence	Operators that mutate persistence mechanisms for local storage like files, and databases (SQLite).
Sensors	Operators that modify events and instructions related to sensors (e.g., gyroscope, step counter, proximity).
Traditional	Operators that introduce modifications related to general bugs found in programming languages (Java).
Automated phases of the mutation testing process – RQ2.3	
Mutant generation	Support for the generation of mutants.
Mutant execution	Implementation of mechanisms to support the mutants execution.
Mutant analysis	Support for processing results related to mutation testing, e.g., mechanisms to determine stillborn, live, killed, or equivalent mutants, as well as the score calculation.
App domain – RQ3.1	
Communication and Social	Apps that foster different kinds of communication between users, in different contexts: messaging, social media, blogs, news, and magazines.
Entertainment	Apps that cover a plenty of common human activities targeting entertainment.
Finance	Apps that involve any kind of financial operations, like banking, stocks, and so on.
Games	Apps that are games, including all existing genres like Action, Arcade, Puzzle, Strategy, etc.
Maps and Navigation	Apps that manipulate geolocation information.
Health	Apps that intend to support a healthier life for users, like medical monitoring and fitness.
Multimedia	Apps that aim to view, collect and edit medias in general like photos, songs, and videos.
Productivity	Apps that support users productivity in different activities.
Sample	Apps provided by vendors to showcase technical concepts and capabilities.
Science and Education	Apps related to scientific knowledge or leisure.
Utilities	Apps that assist in daily tasks of users.

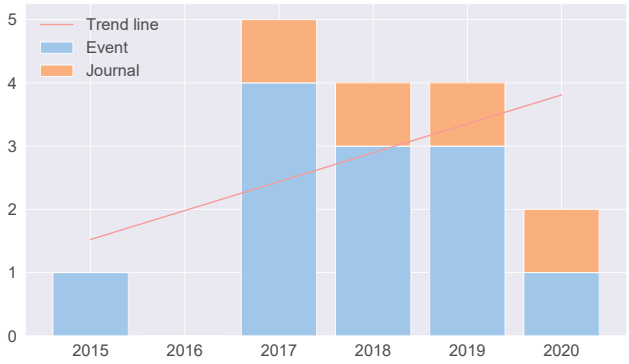


Figure 3: Publications over the years.

Mary, University of Sannio, Universiti Tun Hussein Onn Malaysia, and Towson University. Table 5 shows the authors with more than two studies published and the number of corresponding studies (NS). Whilst, we identified researchers from 13 different countries. Table 6 presents the countries found and the corresponding number of studies (NS).

4.2 Characteristics of the studies

In this section, we address the second group of questions concerning the characteristics of the studies

4.2.1 RQ2.1 – Study Focus and Type. Table 7 lists the classification "Study Focus" and "Study Type" of the primary studies. Most studies (14 out of 16, ≈ 88%) has as the primary focus to address some challenges regarding mutation testing of apps. Only two studies (≈ 12%) have mutation testing as a secondary focus. The focus of these studies is test data generation [PS14, PS15]. Both use a tool named muDroid$_{Wei}$[3] to assess the mutation score obtained by the test suite generated by the tools they implement. Most studies (12 out of 16, ≈ 75%) are classified in the category of proposal solution. The same number of studies (12) conduct some kind of validation/evaluation.

Besides the classification per paper in Table 7, Figure 4 shows the interaction between study types. Bars represent the number of studies; each category is associated with one or more bars, and the bars are associated with one or more categories. When the bar is associated with more than one category, this means that there is an intersection between them, represented by a line with bullets. The majority of the studies (12 out of 16, ≈ 75%) can be qualified as "Proposal of solution". And among those, 8 (67%) conduct a validation. This shows that the authors had the central attention to validate/evaluate their proposal. Only 4 studies are classified as only proposal study or only evaluation research. Only two studies [PS2,

[3]See RQ2.3 for further information about the tools.

Table 5: List of the most prominent authors.

Author	NS
Jeff Offutt	5
Lin Deng	5
Mario Linares-Vásquez	3
Asmau Usman	2
Carlos Bernal-Cárdenas	2
Christopher Vendome	2
Denys Poshyvanyk	2
Gabriele Bavota	2
Kevin Moran	2
Massimiliano Di Penta	2
Michele Tufano	2
Nariman Mirzaei	2
Paul Ammann	2

Table 6: List of countries.

Country	NS
United States of America	10
Colombia	3
Brazil	2
Italy	2
Malaysia	2
Switzerland	2
Chile	1
China	1
France	1
Nigeria	1
Portugal	1
Spain	1
Sweden	1

Table 7: Study focus and type.

Ref.	Study Focus		Study Type		
	Primary	Secondary	Proposal	Evaluation	Validation
[PS1]	✓		✓		
[PS2]	✓		✓	✓	✓
[PS3]	✓		✓	✓	✓
[PS4]	✓		✓		✓
[PS5]	✓			✓	
[PS6]	✓		✓		
[PS7]	✓		✓		
[PS8]	✓			✓	
[PS9]	✓			✓	
[PS10]	✓		✓		✓
[PS11]	✓			✓	
[PS12]	✓		✓		
[PS13]	✓		✓		✓
[PS14]		✓	✓		✓
[PS15]		✓	✓		✓
[PS16]	✓		✓		✓

PS3] belong to the three categories. These studies formally carry out the construction of a fault model, propose mutation operators from the model, and validate the operators in an experiment with a considerable number of subjects.

4.2.2 RQ2.2 – Mutation operators. The mutation operators proposed by the studies are focused on native Android apps. When the mutation operator targets a programming language, only Java is taken into account. Table 8 shows the artifacts subjected to mutation operators. Most studies propose operators that mutate source code in Java (75%) and XML (69%). Java was the first official language to program native Android apps, while XML files are used to specify GUIs and the app's configuration. When the study proposes a source-codeless approach, the SMALI[4] intermediate format is generated from the app's executable (.APK). Three studies (19%) work directly with the app's executable file.

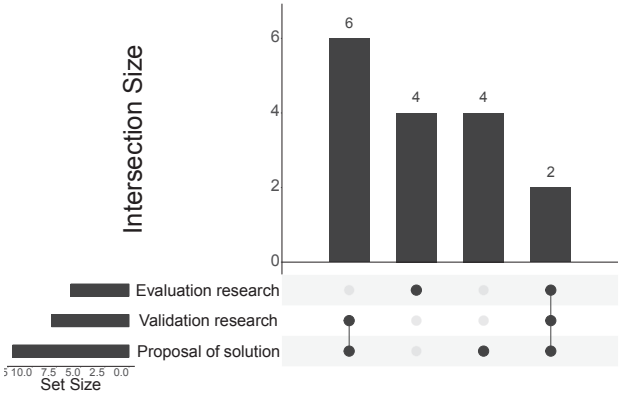


Figure 4: Interaction between study types.

[4] A code created by decompiling the DEX (Dalvik Executable) files, which are the executable files included in Android apps (APK files) [PS6, PS14].

Table 8: Mutated artifacts.

Artifact	Perc.	Primary Studies
Java	75%	[PS1–PS9, PS11, PS13, PS16]
XML	69%	[PS1–PS5, PS7–PS12]
SMALI	19%	[PS12, PS14, PS15]

Table 9 classifies the mutation operators proposed in the studies. We identified a total of 138 mutation operators; most of them (≈64%) proposed in [PS2, PS3]. The work of Jabbarvand et al. [PS2] focuses on energy-aware faults[5], so the operators are mainly in the Configuration, Connectivity, and Location types. On the other hand, Linares-Vasquez et al. [PS3] propose a more diverse distribution of operator types. Eight studies do not introduce any new operator. While 76% of the operators were proposed in studies from 2015 to 2017, new mutation operators were still introduced in 2018 ([PS10] – 14 operators), 2019 ([PS13] – 3 operators), and 2020 ([PS16] – 16 operators).

Table 9: Types and number of mutation operators proposed in the studies.

Year	Ref.	Configuration	Connectivity	GUI	Intent	Location	Persistence	Sensors	Traditional	Total
2015	[PS1]	1	-	5	2	-	-	-	-	8
2017	[PS2]	23	13	4	-	8	-	2	-	50
2017	[PS3]	3	6	14	3	1	7	-	4	38
2017	[PS4]	-	-	2	-	-	-	-	1	3
2017	[PS5]	2	1	2	-	1	-	-	-	6
2017	[PS6]	-	-	-	-	-	-	-	-	0
2018	[PS7]	-	-	-	-	-	-	-	-	0
2018	[PS8]	-	-	-	-	-	-	-	-	0
2018	[PS9]	-	-	-	-	-	-	-	-	0
2018	[PS10]	3	-	11	-	-	-	-	-	14
2019	[PS11]	-	-	-	-	-	-	-	-	0
2019	[PS12]	-	-	-	-	-	-	-	-	0
2019	[PS13]	-	-	3	-	-	-	-	-	3
2019	[PS14]	-	-	-	-	-	-	-	-	0
2020	[PS15]	-	-	-	-	-	-	-	-	0
2020	[PS16]	-	1	6	4	1	-	-	4	16
Total		32	21	47	9	11	7	2	9	138

Operators of GUI and Configuration types (47 and 32, respectively) are more prevalent. One can argue that apps are GUI-centered and have a lot of specific API to be set up (Configuration). Following, there are 21 operators related to connectivity (mainly, to access the Internet) and 11 operators for location (GPS); both are widely used features of a mobile device. Type Intent targets an Android-specific mechanism to inter-app communication (9 operators). The Traditional type deals with operators proposed but that are not Android-specific (9 operators), and 7 operators are related to persistence operations. The Sensor type is the least explored, including only 2 operators [PS2].

4.2.3 RQ2.3 – Mutation testing tools. Table 10 lists the eight mutation testing tools for mobile apps, shown in alphabetical order. Their websites are accessible by clickable links in column "Available". Notice that there are two tools called muDroid; we refer to them by using the first author's name, as muDroid$_{\text{Deng}}$ [PS1] and muDroid$_{\text{Wei}}$ [27][6]. Five tools (62.5%) are available for download and usage, while four of them (50%) are open source.

Table 10: Mutation testing tools.

Tool Name	Available?	Open Source?	Mutant Generation	Mutant Execution	Mutant Analysis	Adopted in
BacterioWeb	No	No				[PS6]
DroidMutator	Yes	Yes	✓	✓		[PS16]
Edroid	No	No	✓			[PS10]
MDroid+	Yes	Yes	✓			[PS3, PS7, PS11, PS16]
μDroid	Yes	No	✓	✓	✓	[PS2]
muDroid$_{\text{Deng}}$	No	No	✓	✓	✓	[PS1–PS5, PS7–PS9]
muDroid$_{\text{Wei}}$	Yes	Yes	✓	✓	✓	[PS3, PS14, PS15]
MutAPK	Yes	Yes	✓			[PS12]

The phases of mutation testing supported by the tools are also shown in Table 10. While BacterioWeb is intended to support the mutation testing process, its initial study only shows the architecture of the tool [PS6]. Then none of the three phases is automated. All the remaining tools support the mutant generation phase. The mutant execution and analysis phases are not always implemented: four tools handle mutant execution, and three tools of mutant analysis. Only μDroid, muDroid$_{\text{Deng}}$, and muDroid$_{\text{Wei}}$ automate the three phases.

The last column shows in which studies the tool was adopted. muDroid$_{\text{Deng}}$ was the most used tool (eight studies), being three studies conducted by research groups that did not implement the tool (i.e., a third-party group). The next one is MDroid+, adopted in four studies (two studies conducted by third parties). muDroid$_{\text{Wei}}$ is adopted in three studies, all from third-party groups. The other tools showed up in only one study.

4.3 Evaluation Aspects

In this section, we analyze the evaluation aspects in order to provide answers for RQ3. Validation/evaluation is essential to provide evidence about the usefulness and applicability of the proposals. Only the studies that conducted some kind of validation or evaluation are considered in the analysis described below. Regarding the validation of a proposed approach, experiments are made using a different number of mobile apps. We identified 137 unique apps

[5]Energy-aware faults can imply on high battery consumption of the mobile device.

[6]muDroid$_{\text{Wei}}$ is described in a non-peer reviewed publication and as such, it is not included in this systematic map.

Table 11: App Domain.

App Domain	Perc.	Primary Studies
Utilities	75%	[PS2–PS5, PS8–PS11, PS13–PS16]
Finance	69%	[PS3–PS5, PS8–PS11, PS13–PS16]
Communication and Social	50%	[PS2–PS4, PS10, PS13–PS16]
Games	50%	[PS3–PS5, PS8, PS9, PS13, PS14, PS16]
Multimedia	50%	[PS2–PS5, PS10, PS13, PS15, PS16]
Science and Education	50%	[PS3–PS5, PS8, PS9, PS13, PS14, PS16]
Productivity	44%	[PS3, PS5, PS11, PS13–PS16]
Entertainment	31%	[PS3, PS5, PS11, PS13, PS16]
Health	25%	[PS3, PS13–PS15]
Maps and Navigation	25%	[PS2, PS3, PS13, PS16]
Sample	6%	[PS3]

used in the studies. Most of them are not so used, ≈ 59% of them (81 out of 137) were used once, and ≈ 31% twice (42 out of 137). Only ≈ 10% of the apps were used more than twice.

Among the most used apps, Tippy Tipper was used 8 times. The others are MunchLife (7 times), Tipster (5 times), AlarmKlock, Jamendo and JustSit (4 times), and the following apps were employed 2 times: CountdownTimer, Translate, PasswordMakerPro, NetCounter, Multi SMS, A2DP Volume, AnyMemo_135, and Aard-Dictionary.

Most of these apps are obtained from the benchmarks: Androtest [PS3, PS11], Dynodroid [PS4, PS8, PS9], EvoDroid [PS4, PS8, PS9], as well as websites like F-Droid [PS2, PS16], GitHub [PS5, PS15], and Google Play [PS10]. In addition, we classified each app according to its domain. Table 11 shows the categories of our classification schema, the percentage of studies that use apps from such domains, and corresponding references. Observe that most of the studies use apps from Utilities and Finance domains. This is in line with the fact that Utilities apps are the most popular category of android apps in worldwide[7]. Besides that, the test setup of such apps is more straightforward, once pre-conditions like authentication are not required.

A rigorous evaluation should consider the other state-of-the-art/practice techniques selected for a comparison with the proposed approach. We identified 7 techniques/approaches/tools commonly used as baseline; they are shown in Table 12. Most of them are general-purpose mutation testing tools for Java programs (namely, Major, Pit, Javalanche, and MuJava). Furthermore, the most commonly used baseline approaches are Major and muDroid$_{\text{Deng}}$.

Table 13 brings other tools adopted during the experiments, divided by groups. The first group contains 10 test generation tools. Monkey is the most used tool to support the studies, used by 25% of the primary studies. The second group involves test frameworks for automating the execution of GUI tests. Robotium is mentioned in 7 studies, while Espresso and uiautomator are cited in just two studies [PS2, PS11]. Other tools cited (last group) are Apktool used to reverse engineer APK files, and App Sensorium used to measure energy consumption.

We also analyzed the main threats mentioned by the authors in the studies. Only 9 studies [PS2–PS5, PS8, PS9, PS11, PS13, PS14] mention some threat. Considering these studies, we observe that *representativeness of mobile apps* appears in 100% of the studies. This fact points out some concerns about the scalability and generalization of the proposed approaches. This threat is followed by *manual mutant check* [PS4, PS5, PS8, PS9] and *flaws in the implementation* [PS4, PS8, PS9, PS13] (44% each).

In 22% of the studies, we identified threats concerning *test data generation tool has random behavior* [PS13, PS14] and *construction of only one test set for each mutation operator* [PS8, PS9]. Other threats are related to the strategy for calculating the mutation score, settings of the test data generation tool, representativeness of faults when hand-seeded, representation of actual app bugs by mutation operators, presumption that the set of operators is complete, power measurement sensitivity, execution environment, and comparison criteria used in the mutation tools.

Although many studies conducted experiments and analysis of results, few of them (19%) apply some kind of statistical test. The Wilcoxon test is used in one study [PS3], followed by Spearman's rank correlation [PS2], ANOVA [PS11], and Grubbs[8] [PS2]. Through this analysis, we observed a lack of statistical tests applied to evaluate the results.

Table 12: Baseline techniques.

Baseline	Perc.	Primary Studies
Major	12%	[PS2, PS3]
muDroid$_{\text{Deng}}$	12%	[PS2, PS3]
Pit	6%	[PS3]
Javalanche	6%	[PS3]
MuJava	6%	[PS3]
MDroid+	6%	[PS16]
muDroid$_{\text{Wei}}$	6%	[PS3]

Table 13: Tools used to support the studies.

Tool Name	Perc.	Primary Studies
Test Generation		
Monkey	25%	[PS2, PS5, PS10, PS11]
A^{3}E	6%	[PS5]
AMOGA	6%	[PS14]
APE	6%	[PS11]
Dynodroid	6%	[PS5]
EvoDroid	6%	[PS4]
iMPAcT	6%	[PS13]
PUMA	6%	[PS5]
Stoat	6%	[PS11]
TEGDroid	6%	[PS15]
GUI Test Execution		
Robotium	37%	[PS2, PS4, PS5, PS9, PS14, PS15]
Espresso	6%	[PS2]
uiautomator	6%	[PS11]
Others		
Apktool	6%	[PS15]
App Sensorium	6%	[PS2]

[7] Market reach of the most popular Android app categories as of September 2019; in these stats, our Utilities category is referred to as "tools".

[8] Grubbs test is used to detect outliers.

5 RESEARCH OPPORTUNITIES

During the analysis conducted to answer our RQs, we identified some trends and research gaps. Based on them, we discuss in this section, the main research opportunities.

5.1 Evolution of the field

We observe that the great majority of the studies have been published in the last three years. We can conclude that mutation testing of android apps is a trendy research topic. We identified several research groups with researchers from 13 different countries. While there exist some initiatives from the Software Engineering community like the IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft), we observe a few number of specific fora (or events) on mobile testing.

> **Opportunity 1.** To promote specific events, fora, and special issues on mobile testing.

5.2 Study focus and types

The focus of almost all studies is to contribute to the mutation testing of mobile apps. Few studies have as focus on the use of mutation testing for test set assessment [PS11, PS14, PS15]. A reason for this is the lack of supporting tools that offer integrated support for the mutation testing process, mainly automatic mutant execution and analysis. Analyzing the found studies that use mutation testing for test case assessment, two of them use the tool muDroid$_{\text{Wei}}$, applying only six traditional mutation operators from the Java language. Only one study is dedicated to execute and analyze mutants generated with operators that explore specific Android faults.

> **Opportunity 2.** To apply mutation testing of mobile apps for a test suite quality assessment. To investigate such an application as a means to evaluate and advance the state of the art in automated test generation for mobile apps.

5.3 Operators proposed

We observe that many studies introduce new operators, most of these studies are recent. However, all studies target the dominant and open-source operating system Android, although there are other mobile operating systems (e.g., iOS). Concerning the programming languages, Java is the only one addressed. No study was found considering Kotlin, which is also an official language for Android. While cross-platform development solutions like React-Native, Flutter, Xamarin have gained some market attraction, no research in this direction was found as well.

> **Opportunity 3.** To target other mobile platforms, operating systems and programming languages, besides Android and Java.

Mutation operators for mobile apps are mostly applied at the source code level, and intermediate formats like SMALI should be further more explored. From 16 studies, only one study applied mutation operators to the SMALI format [PS12]. Another direction is to investigate how to apply mutations at bytecode or dex file levels. A clear advantage for performance is to avoid the app's building process. Moreover, such a solution is likely to work with apps implemented in Kotlin and even in some cross-platform solutions.

> **Opportunity 4.** Better explore the application of mutation operators in different artifacts like bytecode and dex files.

The most common category of operators is the ones describing GUI, connectivity, and configuration faults. But depending of the app domain, other particularities that lead to failures should be explored.

> **Opportunity 5.** Potential to explore other kinds of operators. For instance, domain-specific operators or others related to context-awareness and accessibility.

5.4 Tool support

We observe that existing mutation testing tools mainly support the mutant generation phase. Few tools support the mutant execution and analysis phases: four tools handle mutant execution and only three mutant analysis. Among these last ones: μDroid that analyses the energy consumption to distinguish the mutants; and muDroid$_{\text{Deng}}$, and muDroid$_{\text{Wei}}$ that analyze traditional mutants for Java. We can conclude that the analysis of alive/dead mutants is still challenging because it may depend on the kind of used operator and the app domain. Different kinds of operators may imply distinct ways to perform such an analysis.

> **Opportunity 6.** To improve tool support for mutant execution and analysis.

Still, regarding the mutation analysis support, other tasks deserve attention. For instance, to reduce the number of stillborn, stubborn, and equivalent mutants. Stillborn mutants do not compile due to an incorrect syntactic change. According to Deng et al. [PS1], a mutation system must be prepared to recognize stillborn mutants and remove them from consideration. The DroidMutator tool [PS16] includes the "Type Checking" module to reduce the number of stillborn mutants. The types of variables that can be analyzed include Java primitive data types and object types, Android component types, and developer-defined object types. By checking the type of a variable, the tool can more accurately locate the mutated code, reduce the number of generated stillborn mutants, and improve the effectiveness of the mutation.

The so-called stubborn mutants, those ones that are difficult to kill, have a greater impact in the mobile context. The way adopted

to decide whether a mutant is dead may increase the number of stubborn mutants. The determination of equivalent mutants is an undecidable task and a mutation testing limitation. In our mapping, we have not found works addressing stubborn and equivalent mutants.

Opportunity 7. To offer mechanisms to deal with stillborn, equivalent, and stubborn mutants.

Such mechanisms are essential to reduce test efforts in the application of mutation testing in practice. To this end, mutant-driven test generation also seems to be a topic to be explored.

Opportunity 8. To automatically generate test cases to kill mutants and improve mutation score.

Another limitation is the mutant execution cost. If test suites are at the system level, compiling, packing, deploying, and running each mutant as an individual app is costly. Different directions can be explored like mutant parallelization, schemata, and so on. It is likely to have some overlapping operators, so there is a need for in-depth studies to remove similar or redundant operators. We noticed that only the studies conducted by Deng and Offutt [PS8, PS9] propose to reduce the cost of mutation testing by excluding redundant mutation operators or improving the implementation of their design. According to the authors, the high computational cost of Android mutation testing implies limits in its use in the industry.

Opportunity 9. To investigate strategies to reduce mutant execution costs and optimize Android mutation systems.

5.5 Evaluation Aspects

Choudhary et al. [1] presented a comparative study of the main existing test input generation techniques and corresponding tools for Android. They evaluated these tools according to four criteria: code coverage, fault detection capabilities, ease of use, and compatibility with multiple Android framework versions. The experimental setup is constructed with Vagrant and Virtualbox. Inside each machine instance, we can reproduce an experiment with the same apps and Android emulators. This kind of setup was not found in our mapping. Besides, few works apply statistical tests. A more rigorous evaluation is necessary. We identified in the studies a lack of available benchmarks of apps used in their evaluation process. To offer replication packages for experiments may be hard in the mobile context due to configuration options and other difficulties inherent to the apps.

Opportunity 10. To offer benchmarks and conduct more rigorous studies to evaluate mutation testing for mobile apps. To address existing challenges to replicate experiments.

6 THREATS TO VALIDITY

In this section, we identify possible threats to the validity of our review results, according to the taxonomy of Wohlin et al. [28].

Construct validity refers to the relation between theory and observation. The main threats in this category are related to the research questions, the digital databases, and the search string. Regarding the research questions, they may not address all aspects of the Mutation Testing of Apps. This threat was mitigated through discussions, and we believe that the questions reflect the goals of our work. Furthermore, we elaborated on questions to cover different aspects of a mutation testing proposal, tools used, and how the approaches are evaluated. Different questions could be elaborated by other researchers and lead to different results.

The chosen databases are well-known and index the main software engineering and testing publications. Concerning the search string, we carefully selected the terms that best fit our goals. We also refined our string several times using a control group, and we reduced the risk that relevant literature is omitted using snowballing. Hence, we surmise the found studies represent the Mutation Testing of Apps area.

Internal validity evaluates the relationship between the treatment and the output. In this study, the treatment is the set of papers included, and the outcome is the analysis reported. So, the data extraction is a potential threat. Subjective decisions might have occurred during primary studies selection and data extraction. Some relevant studies may not be selected as primary studies, and perhaps we may have extracted or misinterpreted some information. To reduce this threat, we followed a rigorous plan using the PRISMA statement, elaborated a well-defined inclusion and exclusion criteria, and applied a snowballing procedure. Furthermore, we had several meetings to clarify any doubt arisen during the process.

Conclusion validity is related to issues that affect the ability to draw the correct conclusion. An identified threat is the classification schema, considering how we grouped the papers and established relations between them. Besides, we noticed another threat regarding the granularity of the information presented in the primary studies. If some information is not described in these studies, it may affect our conclusions. To mitigate them, we first documented all relevant information from the primary studies, guided by the dimensions associated with the research questions, and defined the categories interactively. However, other researchers may obtain another scheme and ways to group and analyze the papers.

Reliability validity is concerned with issues that impact the ability to draw that the operations of a study can be repeated with the same results. We believe that our study is replicable, following the steps described, or using the raw data analyzed and disseminated by OSF [25].

7 CONCLUDING REMARKS

This paper presents results from a mapping study on mutation testing of apps, investigating in the primary studies: focus and type of research conducted, mutation operators proposed and supporting tools, and how the evaluation has been conducted. Furthermore, we analyze the main publication fora and how the field has been evolving over the years. We provide research opportunities to drive future research on this subject.

The map is based on 16 papers. We observed this is a trendy topic, with 98% of the papers published in the last three years, and a crescent interest in the field. Most studies propose new mutation operators and are focused on validation/evaluation of the proposal. Furthermore, many of them introduce a tool to support different mutation testing phases. However, we observe a lack of benchmark and more rigorous experiments, and some difficulties to allow replicability.

There are still many challenges to be addressed by future work to allow mutation testing of mobile apps to be adopted in practice. Several pragmatic questions remain open like the scalability of mutant execution and particularities of mutant analysis, such as determination of equivalent, stillborn, stubborn, and redundant mutants. Besides native development in Android and Java, other relevant mobile platforms and programming languages are not covered in the literature. Another question is to investigate how mutation testing can help to advance the state-of-the-art in automated test generation for mobile apps, a hot topic in the last years related to the automation of the mutant analysis phase that encompasses many challenges in the mobile context. These are all research opportunities that can be further explored.

This work is partially supported by the Brazilian agencies CAPES and CNPq (Andre T. Endo grant nr. 420363/2018-1 and Silvia Regina Vergilio grant nr. 305968/2018-1).

PRIMARY STUDIES

[PS1] L. Deng, N. Mirzaei, P. Ammann, J. Offutt, Towards mutation analysis of Android apps, in: Proceedings of the Eighth International Conference on Software Testing, Verification and Validation Workshops, ICSTW, IEEE, 2015, pp. 1–10. doi:10.1109/ICSTW.2015.7107450.

[PS2] R. Jabbarvand, S. Malek, μDroid: an energy-aware mutation testing framework for Android, in: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE, ACM, 2017, pp. 208–219. doi:10.1145/3106237.3106244.

[PS3] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, D. Poshyvanyk, Enabling Mutation Testing for Android Apps, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE, ACM, New York, NY, USA, 2017, pp. 233–244. doi:10.1145/3106237.3106275.

[PS4] L. Deng, J. Offutt, P. Ammann, N. Mirzaei, Mutation operators for testing Android apps, Information and Software Technology 81 (2017) 154–168. doi:10.1016/j.infsof.2016.04.012.

[PS5] L. Deng, J. Offutt, D. Samudio, Is Mutation Analysis Effective at Testing Android Apps?, in: Proceedings of the International Conference on Software Quality, Reliability and Security, QRS, IEEE, 2017, pp. 86–93. doi:10.1109/QRS.2017.19.

[PS6] M. P. Usaola, G. Rojas, I. Rodríguez, S. Hernández, An Architecture for the Development of Mutation Operators, in: Proceedings of the International Conference on Software Testing, Verification and Validation Workshops, ICSTW, IEEE, 2017, pp. 143–148. doi:10.1109/ICSTW.2017.31.

[PS7] K. Moran, M. Tufano, C. Bernal-Cárdenas, M. Linares-Vásquez, G. Bavota, C. Vendome, M. Di Penta, D. Poshyvanyk, Mdroid+: A mutation testing framework for Android, in: Proceedings of the 40th International Conference on Software Engineering: Companion, ICSE-Companion, ACM, 2018, pp. 33–36. doi:10.1145/3183440.3183492.

[PS8] L. Deng, J. Offutt, Reducing the Cost of Android Mutation Testing, in: Proceedings of the 30th International Conference on Software Engineering and Knowledge Engineering, SEKE, KSI Research Inc. and Knowledge Systems Institute Graduate School, 2018, pp. 542–591. doi:10.18293/SEKE2018-184.

[PS9] L. Deng, J. Offutt, Experimental Evaluation of Redundancy in Android Mutation Testing, International Journal of Software Engineering and Knowledge Engineering 28 (11n12) (2018) 1597–1618. doi:10.1142/S0218194018400193.

[PS10] E. Luna, O. El Ariss, Edroid: A Mutation Tool for Android Apps, in: Proceedings of the 6th International Conference in Software Engineering Research and Innovation, CONISOFT, IEEE, 2018, pp. 99–108. doi:10.1109/CONISOFT.2018.8645883.

[PS11] H. N. da Silva, P. R. Farah, W. D. F. Mendonça, S. R. Vergilio, Assessing Android Test Data Generation Tools via Mutation Testing, in: Proceedings of the IV Brazilian Symposium on Systematic and Automated Software Testing, SAST, ACM, New York, NY, USA, 2019, p. 32–41. doi:10.1145/3356317.3356320.

[PS12] C. Escobar-Velásquez, M. Osorio-Riaño, M. Linares-Vásquez, MutAPK: Source-Codeless Mutant Generation for Android Apps, in: Proceedings of the International Conference on Automated Software Engineering, ASE, IEEE, 2019, pp. 1090–1093. doi:10.1109/ASE.2019.00109.

[PS13] A. C. R. Paiva, J. M. E. P. Gouveia, J.-D. Elizabeth, M. E. Delamaro, Testing When Mobile Apps Go to Background and Come Back to Foreground, in: Proceedings of the International Conference on Software Testing, Verification and Validation Workshops, ICSTW, IEEE, 2019, pp. 102–111. doi:10.1109/ICSTW.2019.00038.

[PS14] I.-A. Salihu, R. Ibrahim, B. S. Ahmed, K. Z. Zamli, A. Usman, AMOGA: A Static-Dynamic Model Generation Strategy for Mobile Apps Testing, IEEE Access 7 (2019) 17158–17173. doi:10.1109/ACCESS.2019.2895504.

[PS15] A. Usman, N. Ibrahim, S. Anka, TEGDroid: Test Case Generation Approach for Android Apps Considering Context and GUI Events, International Journal on Advanced Science, Engineering and Information Technology 10 (2020) 16. doi:10.18517/ijaseit.10.1.10194.

[PS16] J. Liu, C. X. Xiao, L. Xu, L. Dou, C. A. Podgurski, X. Xiao, A. Podgurski, DroidMutator: An Effective Mutation Analysis Tool for Android Applications, in: Proceedings of the 42nd International Conference on Software Engineering Companion, ICSE-Companion, ACM, 2020. doi:10.1145/3377812.3382134.

REFERENCES

[1] Choudhary SR, Gorla A, Orso A. Automated test input generation for android: Are we there yet? (e). *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Institute of Electrical and Electronics Engineers Inc., 2015; 429–440, doi:10.1109/ASE.2015.89.

[2] Su T, Meng G, Chen Y, Wu K, Yang W, Yao Y, Pu G, Liu Y, Su Z. Guided, stochastic model-based GUI testing of android apps. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017; 245–256, doi:10.1145/3106237.3106298.

[3] Mao K, Harman M, Jia Y. Sapienz: Multi-objective automated testing for android applications. *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, Association for Computing Machinery: New York, NY, USA, 2016; 94–105, doi:10.1145/2931037.2931054.

[4] Gu T, Sun C, Ma X, Cao C, Xu C, Yao Y, Zhang Q, Lu J, Su Z. Practical GUI Testing of Android Applications via Model Abstraction and Refinement. *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, IEEE Press, 2019; 269–280, doi:10.1109/ICSE.2019.00042.

[5] Zein S, Salleh N, Grundy J. A systematic mapping study of mobile application testing techniques. *Journal of Systems and Software* 2016; **117**:334–356, doi:10.1016/j.jss.2016.03.065.

[6] Tramontana P, Amalfitano D, Amatucci N, Fasolino AR. Automated functional testing of mobile applications: a systematic mapping study. 2019; 149–201, doi:10.1007/s11219-018-9418-6.

[7] Kong P, Li L, Gao J, Liu K, Bissyandé TF, Klein J. Automated Testing of Android Apps: A Systematic Literature Review. *IEEE Trans. Reliability* 2019; **68**(1):45–66, doi:10.1109/TR.2018.2865733.

[8] Nguyen BN, Robbins B, Banerjee I, Memon A. Guitar: An innovative tool for automated testing of gui-driven software. *Automated Software Engg.* Mar 2014; **21**(1):65–105, doi:10.1007/s10515-013-0128-9.

[9] Su T. FSMdroid: Guided GUI Testing of Android Apps. *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, Association for Computing Machinery: New York, NY, USA, 2016; 689–691, doi:10.1145/2889160.2891043.

[10] Morgado IC, Paiva ACR. The IMPAcT Tool for Android Testing. Association for Computing Machinery: New York, NY, USA, 2019, doi:10.1145/3300963.

[11] Ferreira J, Paiva ACR. Android Testing Crawler. *Quality of Information and Communications Technology*, Piattini M, Rupino da Cunha P, García Rodríguez de Guzmán I, Pérez-Castillo R (eds.), Springer International Publishing: Cham, 2019; 313–326, doi:10.1007/978-3-030-29238-6_23.

[12] Jia Y, Harman M. An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.* 2011; **37**(5):649–678, doi:doi.org/10.1109/TSE.2010.62.

[13] Papadakis M, Kintis M, Zhang J, Jia Y, Traon YL, Harman M. Chapter six - mutation testing advances: An analysis and survey. *Advances in Computers* 2019; **112**:275–378, doi:10.1016/bs.adcom.2018.03.015.

[14] Petersen K, Feldt R, Mujtaba S, Mattsson M. Systematic mapping studies in software engineering. *12th Int. Conf. Eval. Assess. Softw. Eng.* 2008; :68–77doi:10.5555/2227115.2227123. Cited By 81.

[15] Madeyski L, Orzeszyna W, Torkar R, Jozala M. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Trans. Software Eng.* 2014; **40**(1):23–42, doi:10.1109/TSE.2013.44.

[16] Silva RA, do Rocio Senger de Souza S, de Souza PSL. A systematic review on search based mutation testing. *Information & Software Technology* 2017; **81**:19–35, doi:10.1016/j.infsof.2016.01.017.

[17] Souza FCM, Papadakis M, Durelli VHS, Delamaro ME. Test data generation techniques for mutation testing: A systematic mapping. *CIbSE*, Curran Associates, 2014; 419–432, doi:10.13140/RG.2.1.3699.9209.

[18] Prado Lima JA, Vergilio SR. A systematic mapping study on higher order mutation testing. *Journal of Systems and Software* 2019; **154**:92–109, doi:10.1016/j.jss.2019.04.031.

[19] Pizzoleto AV, Ferrari FC, Offutt J, Fernandes L, Ribeiro M. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software* 2019; **157**, doi:10.1016/j.jss.2019.07.100.

[20] Méndez-Porras A, Quesada-López C, Jenkins M. Automated testing of mobile applications: A systematic map and review. *CIbSE*, Curran Associates, Inc., 2015; 195. URL Open repository.

[21] Moher D, Liberati A, Tetzlaff J, Altman DG. Preferred Reporting Items for Systematic Reviews and Meta-Analyses: The PRISMA Statement. *BMJ* 2009; **339**, doi:10.1136/bmj.b2535.

[22] Garousi V, Felderer M, Mäntylä MV. The need for multivocal literature reviews in software engineering: Complementing systematic literature reviews with grey literature. EASE '16, Association for Computing Machinery: New York, NY, USA, 2016, doi:10.1145/2915970.2916008.

[23] Garousi V, Felderer M, Mäntylä MV. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology* 2019; **106**:101 – 121, doi:https://doi.org/10.1016/j.infsof.2018.09.006.

[24] Wohlin C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE'14, ACM, 2014; 38:1–38:10, doi:10.1145/2601248.2601268.

[25] da Silva HN, Prado Lima JA, Endo A, Vergilio SR. A Review on Mutation Testing in Mobile Applications may 2020, doi:XX.XXXXX/OSF.IO/XXXXX. URL ReviewMTApps.

[26] Wieringa RJ, Maiden NAM, Mead NR, Rolland C. Requirements engineering paper classification and evaluation criteria: a proposal and a discussion. *Requirements Eng* 2006; **11**:102–107, doi:10.1007/s00766-005-0021-6.

[27] Wei Y. Mudroid: Mutation testing for android apps. https://github.com/Yuan-W/muDroid 2015.

[28] Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers: USA, 2000, doi:10.5555/2349018.

APÊNDICE B – ON THE RELATION BETWEEN CODE ELEMENTS AND ACCESSIBILITY ISSUES IN ANDROID APPS

Este apêndice inclui o estudo que mapeia os *code elements* da API Android com os critérios de sucesso do guia de acessibilidade WCAG.

On the Relation between Code Elements and Accessibility Issues in Android Apps

Henrique Neves da Silva
Federal University of Paraná
Curitiba, PR, Brazil
henriqueneves@ufpr.br

Andre Takeshi Endo
Universidade Tecnológica Federal do Paraná
Cornélio Procópio, PR, Brazil
andreendo@utfpr.edu.br

Marcelo Medeiros Eler
University of São Paulo
São Paulo, SP, Brazil
marceloeler@usp.br

Silvia Regina Vergilio
DInf, Federal University of Paraná
Curitiba, PR, Brazil
silvia@inf.ufpr.br

Vinicius H. S. Durelli
Federal University of São João del Rei
São João del Rei, MG, Brazil
durelli@ufsj.edu.br

ABSTRACT

Mobile apps have gone mainstream and become part of our daily lives. Currently, many efforts have been made to make apps more accessible to people with disabilities. However, little is still known on how to implement more accessible apps. In the Android API, there are (code) elements that may be employed to (in)directly improve the app's accessibility. This paper aims to investigate the prevalence of accessibility code elements and their relation to potential accessibility issues. First, we identified code elements of the native Android API that may be related to accessibility features, and mapped them to principles and success criteria of the Web Content Accessibility Guidelines (WCAG) 2.1. Using a sample of 111 open source mobile apps available in Google Play, we conducted a characterization study to examine the prevalence of accessibility code elements. We also analyzed how these code elements are related to issues detected by the static analyzer Android Lint and the accessibility testing tool MATE. Our results indicate that code elements are not widely used; the ones directly related to accessibility are present in only a few apps. Additionally, our results would seem to suggest that apps that adopt accessibility code elements, tend to have less accessibility issues. By analyzing our results from the standpoint of the WCAG principles, we conclude that there is room for improvement in terms of how both the Android API and automated testing tools deal with accessibility-related issues.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Mobile apps. Accessibility.

1 INTRODUCTION

A mobile application (app) is a software program that runs in portable devices. Currently, apps play an essential role in the daily life of billions of people, including people with disabilities. A significant part of the global population has some form of disability [19] and many countries have devised specific legislation [17] to guarantee that both private and public organizations provide their users with accessible services or products. In this context, accessibility, defined as "the quality of being easily reached, entered, or used by people that have a disability" [15], plays a pivotal role in the creation of accessible services and products. Similarly, since there is a need to ensure that the accessibility features are being properly employed and are usable by people with disabilities, as accessibility testing and evaluation have also become essential.

Developing accessible products is not trivial. Each type of disability requires different design strategies. Conducting user studies to understand the characteristics of a wide range of users with different types of disabilities tends to be costly and time consuming. However, developers can resort to standards and guidelines that synthesize requirements and best practices concerning the development of accessible products for a broad range of disabilities. Many accessibility standards and guidelines have been proposed over the years, such as W3C's Web Content Accessibility Guideline[1] (WCAG), which is one of the most popular accessibility standards. WCAG encompasses several guidelines, each one related to different success criteria, grouped into four accessibility principles. It covers recommendations for making digital products more accessible for individuals with blindness and low vision, deafness and hearing loss, limited movement, speech disabilities, photo-sensitivity, learning disabilities, and cognitive limitations.

Devising accessible mobile apps requires developers to adopt such standards in their development process. In particular, the Android platform offers several resources to the development of accessible apps and developers can set different properties that may directly or indirectly impact accessibility principles. Such properties can be statically set using design layout files or written in the source code. However, setting many of these properties is not mandatory since they have default values handled by the platform. We refer to these elements of the code used to set such properties as *accessibility code elements*.

Accessibility evaluation can be conducted in different ways: by real users; by specialists using heuristics and checklists; or by automated tools. Accessibility evaluation should not be solely conducted based on automated tools [23] because they cannot detect problems that require human judgement (e.g. whether error descriptions are meaningful), but they can increase productivity and might be the only resource in some cases (e.g. limited time and budget). Automated tools commonly use recommendations of the standards and

[1] https://www.w3.org/TR/WCAG21

guidelines as oracles to check whether the app meets the requirements. For instance, the WCAG success criteria defined for each guideline are written as testable statements to check for guideline violations. We refer to potential violations uncovered by automated tools as *accessibility issues*.

Automated tools can perform static or dynamic analysis [20]. Static analysis can quickly analyze all assets of an app [13], but it cannot find violations that can only be detected during runtime (e.g., low color contrast).

Dynamic analysis requires the execution of the app and uses different strategies to explore the app (manual, test scripts, or automatic input generation) to find violations during runtime. Dynamic analysis finds more violations than static analysis because a limited number of violations can be checked statically, but dynamic analysis tends to be time consuming and to cover only a subset of the app due to weak test scripts or limited input generation algorithms [10, 16, 20]. Strategies to improve accessibility evaluation leveraging the benefits of static analysis would be a good contribution to this field.

Motivated by this fact and to offer some insights to propose such strategies, this work investigates the relation between accessibility issues and the presence of accessibility code elements. To identify the accessibility issues we used two automated evaluation tools: Android Lint [13], which performs static analysis; and MATE [10], which automatically generates random inputs to explore the app and run several accessibility tests in each visited screen. To select accessibility code elements we mapped code elements of the Android API to a WCAG principle, guideline and/or success criterion.

Our study is guided by three research questions to investigate: 1) the prevalence of the accessibility code elements in real-word Android apps; 2) the most common accessibility issues in such apps; and 3) the correlation between accessibility issues and code elements, indicating whether the presence of the later is an indication that accessibility has been considered in the app development, thus leading to an app with fewer accessibility issues. To answer our research questions, we analyzed the source code of 111 apps. Our results show that while accessibility code elements are not widely adopted, accessibility issues are prevalent in the apps with an average of 0.04 issue per LoC. We also found several correlations between code elements and accessibility issues. The results of our investigation give evidence that code elements data may be used to improve tools for accessibility testing and evaluation of mobile apps. Hence, the contributions of this work are manifold: i) the mapping of code elements in the Android API to WCAG guidelines, which serves as a guide to developers and researchers; ii) construction of a corpus of open source apps, identifying code elements and accessibility issue; and iii) an empirical study that relates code elements and issues from a mobile accessibility perspective, and derived implications to be considered in the accessibility evaluation area.

The remainder of this paper is organized as follows. Section 2 presents the results of our mapping, relating accessibility code elements to the WCAG principles and success criteria. Section 3 shows the study setup, while the results are outlined and analyzed in Section 4. Section 5 discusses related work. Finally, Section 6 presents concluding remarks and discusses future work.

2 MAPPING ANDROID API TO WCAG

Developers may use resources provided by the Android API to implement accessible mobile apps. Some of these accessibility-related resources manifest as XML attributes in GUI layout files or methods of Java classes used to set some property or to implement certain behavior. We refer to them as *accessibility code elements*. Additonally, developers can resort to standards such as WCAG.

WCAG encompasses four main principles: **Perceivable**, defined as "the information must be presentable to users in ways they can perceive"; **Operable**, "User interface components and navigation must be operable."; **Understandable**, "Information and the operation of user interface must be understandable."; **Robust**, "Content must be robust enough that it can be interpreted by a wide variety of user agents, including assistive technologies". Each principle has several guidelines and each guideline has different success criteria.

In this section, we present the mapping between Android code elements and WCAG we used to select the accessibility code elements of our study. Prior to devising this mapping, we looked at previous studies [7, 8, 10, 11, 22] and at the official website of the current version (30) of the Android API[2] to analyze all properties that can be set to a user interface component. Based on the description of each property, we checked whether it could be associated with a success criterion of the WCAG.

The mapping was conducted by one of the authors of this paper and the results were reviewed by two other authors with expertise in Android Development and Mobile Accessibility. Table 1 shows the accessibility code elements we identified (a total of 55 distinct elements). The accessibility code elements are organized by WCAG principle (1st column) and success criteria (2nd column). A code element may appear as an XML attribute (3rd column), a Java method (4th column), or both. For instance, element `contentDescription` can be set in an XML attribute or by a Java method. Some Java methods are related to two or more code elements (e.g., `setLineSpacing` and `setAutoSizeTextTypeUniformWithConfiguration`); in this case, they are distinguished by their parameters.

Some code elements can be directly linked to a success criterion. For instance, the code element `:inputType` is defined as: *"The type of data being placed in a text field, used to help an input method to decide how to let the user enter text"*. One of the success criterion of the WCAG is called *"Identify Input Purpose"*, which is intended to check whether the purpose of each input field collecting information was programmatically determined. On the other hand, some code elements are directly linked to a success criterion. For example, `:textColor` is used to determine the color of a text element. The color of the element alone does not directly impact accessibility, but when combined with the background color it may be related to the success criterion "`Contrast`", or "`Use of color`", depending on the context. Considering both cases, we distinguish accessibility code elements between DIA (Directly Impacts Accessibility) or IIA (Indirectly Impacts Accessibility) – refer to the last column of Table 1. Notice that adopting a given code element in practice does not imply that the app meets the associated WCAG success criterion. Success criteria may define intricate behavioral patterns that can only be fulfilled by combining code elements. Our assumption is

[2]https://developer.android.com/reference

that the resulting mapping only suggests that the use of a particular code element may impact on one of the success criteria.

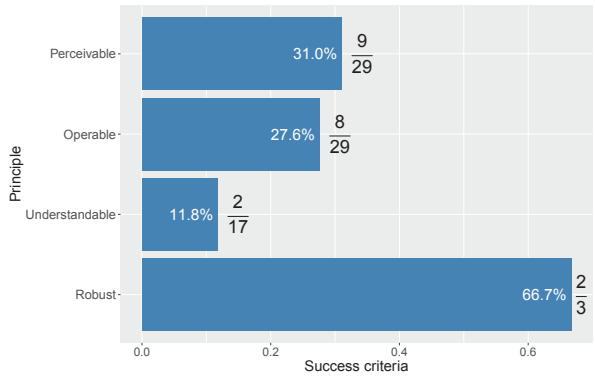


Figure 1: Success criteria covered by code elements.

We found code elements related to all WCAG Principles: 27 for Perceivable, 15 for Operable, 2 for Understandable, and 11 for Robust. Figure 1 summarizes, per principle, how many success criteria are covered by the code elements we identified. For Perceivable, 9 out of 29 criteria (31%) have some code element associated, Operable has 8 out of 29 (≈ 28%), Understandable has 2 out of 17 (≈ 12%), and Robust 2 out of 3 (≈ 67%).

3 STUDY SETUP

This section presents the setup of the empirical study we conducted to investigate whether the presence of accessibility code elements in the source code is an evidence that accessibility has been considering during development, hence resulting in more accessible apps.

3.1 Research Questions

Our study was guided by the following research questions (RQs):

- RQ1: To what extent code elements related to accessibility have been adopted in Android apps?
- RQ2: How common are accessibility issues in real-world Android apps?
- RQ3: Is there any correlation between code elements and accessibility issues?

RQ1 investigates the prevalence of the accessibility code elements in the apps. RQ2 aims to identify the most common accessibility issues (violations) provided by accessibility evaluation tools. RQ3 examines if the use of the accessibility code elements, which were mapped to WCAG, leads to less accessibility issues.

To answer our RQs, we need a sample of mobile apps; to collect information about the accessibility code elements; and to evaluate the accessibility of the mobile apps of our sample. We adopted R-studio and pandas to support the data analysis. Since our data departs from linearity and contains outliers, we used Spearman's rank correlation coefficient to compute the correlation between variables. Spearman's rank correlation coefficient is a robust, non-parametric correlation test for determining the extent to which two variables are linearly associated. In this context, results from this test range from -1 to 1, where 1 indicates a perfect positive linear relationship, -1 a perfect negative relationship, and 0 suggests no linear relationship. We interpret correlation values according to Cohen's guideline [5]: a small correlation has an absolute value of around 0.1, a medium correlation has a value of 0.3, and large correlations have values of 0.5 or greater.

In hopes of supporting future replication, datasets, results, and the implemented tool are available[3].

3.2 Mobile apps sample

We selected mobile apps from a repository of open source Android apps, named F-Droid[4]. We initially selected 527 projects, updated in 2019 or 2020 to make sure the developers had the chance to incorporate new features related to the Android framework. Next, we narrowed down our selection to apps that have a GitHub repository and are available on Google Play Store, meaning it is available for any Android user. As our implementation targets native apps written in Java, non-native projects were filtered out. Finally, we also removed projects that presented failures using the Android's default build tool Gradle, crashed in their initial screen or during a test session. The resulting sample comprises 111 apps.

Table 2 shows information about the apps in our sample: we summarize our sample in terms of lines of code (LoC), number of installs, score (ratings), amount of accessibility issues found by the tools Lint and MATE (see Section 3.3). The rating and installs metrics are based on end-user feedback collected from the Google Play store. The sample has apps with different sizes with the average around 10 KLoC. `atalk-android` is the biggest app with more than 250 KLoC, while `heb12-mobile` is the smallest app with 250 LoC. Some apps have a huge number of installs, such as `unicodepad` with more than 8 million installs, and a wide range of apps have been rated by users and the average score of all apps is 3.2. We adopted apps from 20 different categories collected from Google Play store. Most apps are tools (34.2%), followed by productivity (10.8%) and education (9%).

3.3 Tools

We extended Prof.Mapp [21] to collect data about the accessibility code elements. Prof.Mapp analyzes source code to identify and quantify the number of GUI elements, sensors, test frameworks, and system configurations of Android apps. Our extension quantifies properties settings or Java code that can directly or indirectly impact the accessibility aspects of the Android app, namely accessibility code elements identified in our mapping. Our implementation walks through the folders of an app project, and searches for code elements in two types of analysis (i) processing Java files and (ii) XML files. The result is the number of accessibility code elements used in each app separated by each type of asset (JAVA or XML), and what principles of the WCAG 2.1 guide (as mapped in Section 2) have been covered. For the Google Play data, we have implemented a script that crawls the store pages and it collects the number of installs, category, and ratings for each app in our sample.

We evaluated the app accessibility using two automated tools: Android Lint and MATE. Android Lint [13] is a static code analyzer integrated with Android Studio IDE. It can also be executed with the SDK tools and build package manager Gradle. The Android

[3]https://doi.org/10.17605/OSF.IO/JSHP3
[4]https://www.f-droid.org

Table 1: Mapping code elements to WCAG principles and success criteria.

Principle	Success criteria	Code Elements		DIA?*
		XML Attributes	Java Methods	
Perceivable	Non-text Content	:contentDescription	setContentDescription	Y
	Captions	—	addCaptioningChangeListener	Y
	Captions	—	getFontScale	Y
	Captions	—	getLocale	Y
	Captions	—	getUserStyle	N
	Captions	—	isEnabled	N
	Captions	—	removeCaptioningChangeListener	N
	Orientation	:screenOrientation	setScreenOrientation	Y
	Identifty Input Purpose	:inputType	setInputType	Y
	Identifty Input Purpose	:inputMethod	–	N
	Identifty Input Purpose	:phoneNumber	–	Y
	Identifty Input Purpose	:password	setTransformationMethod	Y
	Identifty Input Purpose	:numeric	–	Y
	Identifty Input Purpose	:digits	–	Y
	Identifty Input Purpose	:autoText	setAutoText	Y
	Identifty Input Purpose	:editable	setEditable	N
	Use of Color; Contrast	:background	setBackgroundColor	N
	Use of Color; Contrast	:textColor	setTextColor	N
	Resize Text	:autoSizeTextType	setAutoSizeTextTypeWithDefaults	Y
	Resize Text	:textSize	setTextSize	Y
	Resize Text	:textScaleX	setTextScaleX	Y
	Resize Text	:autoSizeMaxTextSize	setAutoSizeTextTypeUniformWithConfiguration	Y
	Resize Text	:autoSizeMinTextSize	setAutoSizeTextTypeUniformWithConfiguration	Y
	Resize Text	:autoSizePresetSizes	setAutoSizeTextTypeUniformWithPresetSizes	Y
	Text Spacing	:lineSpacingMultiplier	setLineSpacing	Y
	Text Spacing	:lineSpacingExtra	setLineSpacing	Y
	Text Spacing	:letterSpacing	setLetterSpacing	Y
Operable	Animation from Interactions	:animateLayoutChanges	setLayoutTransition	Y
	Keyboard; Focus Order	:nextFocusForward	setNextFocusForwardId	Y
	Keyboard; Focus Order	:nextFocusUp	setNextFocusUpId	Y
	Keyboard; Focus Order	:nextFocusDown	setNextFocusDownId	Y
	Keyboard; Focus Order	:nextFocusLeft	setNextFocusLeftId	Y
	Keyboard; Focus Order	:nextFocusRight	setNextFocusRightId	Y
	Page Titled	:accessibilityPaneTitle	setAccessibilityPaneTitle	Y
	Focus Order	:accessibilityTraversalBefore	setAccessibilityTraversalBefore	Y
	Focus Order	:accessibilityTraversalAfter	setAccessibilityTraversalAfter	Y
	Heading and Labels	:accessibilityHeading	setAccessibilityHeading	Y
	Focus Visible	:cursorVisible	setCursorVisible	Y
	Target Size	:minWidth	setMinWidth	Y
	Target Size	:minHeight	setMinHeight	Y
	Focus Order	:screenReaderFocusable	setScreenReaderFocusable	Y
	Focus Order	:focusable	setFocusable	N
Understandable	Label or Instructions	:hint	setHint	Y
	Label or Instructions	:labelFor	setLabelFor	Y
Robust	Name, Role, Value	–	onInitializeAccessibilityNodeInfo	Y
	Name, Role, Value	–	replaceAccessibilityAction	Y
	Status Messages	:accessibilityLiveRegion	setAccessibilityLiveRegion	Y
	Status Messages	:importantForAccessibility	setImportantForAccessibility	Y
	Status Messages	:hapticFeedbackEnabled	setHapticFeedbackEnabled	Y
	Status Messages	–	sendAccessibilityEvent	Y
	Status Messages	–	sendAccessibilityEventUnchecked	Y
	Status Messages	–	dispatchPopulateAccessibilityEvent	Y
	Status Messages	–	onPopulateAccessibilityEvent	Y
	Status Messages	–	onBindViewHolder	N
	Status Messages	–	onRequestSendAccessibilityEvent	Y

*DIA stands for Directly Impacting on the app's Accessibility; N implies an IIA element.

Table 2: Apps that comprise our sample.

App	LoC	#Installs	Score	#Lint	#MATE
10000sentences	5334	35037	4.32	5	353
activitydiary	6370	1766	3.89	2	137
aaegis	10613	32682	4.74	23	131
aelf-dailyreadings	10377 2	453101	3.98	2	172
androdns	1816	1025	0	3	69
android-gotify	5341	4177	4.29	2	11
android-gpsd-client	466	2	0	0	28
androidnetworktools	1380	4410	4	0	15
androidplanisphere	3443	4929	4.83	1	102
anlinux-adfree	3468	827	0	0	40
applock	6683	3381	2.82	22	12
archpackages	2317	6782168	4.42	0	56
atalk-android	250192	2748	4.36	148	231
authorizer	28286	18377	4.25	23	34
c-beam-droid	8255	369985	2.71	10	120
calculator-notification	423	20572	4.48	6	3
calliope-android-app	6333	940	3.43	18	522
contentblocker	5185	8	0	23	23
darkcroc	310	143	0	0	8
dash-wallet	2259	2756	4.14	20	2
dash-oms	310	511233	4.56	0	31
democracy-client	80548	183	5	0	14
diaguard	35424	2531	3.33	3	219
diceware-pass-gen	970	24651	4.56	0	25
dokuwikiandroid	4075	31	0	1	43
download-navi	17189	3323	4.18	2	896
dsaassistant	1585	161	4.2	4	109
exodus-android-app	2398	325269	4.39	4	8
fediphoto	1680	68	0	0	32
flashcards	1360	3154	1.57	0	40
funktrainer	2728	999283	3.66	0	35
getmitokens	293	18555	4.4	0	3
getoffyourphone	3216	1944285	4.07	1	23
goodtime	7524	37312	4.15	9	33
habpanelviewer	9100	8053	2.85	2	18
hacs	2754	35	0	3	48
hauk	5250	297	4.91	0	36
heb12-mobile	250	10104	3.46	0	438
helsinki-timetables	6085	941570	3.83	6	1109
hu-eduroam	591	34338	3.82	14	10
huewidgets	2484	497	4.25	0	5
kboard	1292	16	0	0	22
ldap-sync	2046	1273	3.25	5	26
lepsi-rozvrh	7670	54673	4.29	6	15
light-android-launcher	614	8005903	4.39	2	9
ltecleanerfoss	486	3687	3.27	0	30
lttrs-android	8726	70894	4.15	20	6
manglish	633	281	0	1	17
mbestyle	2030	431789	4.64	8	32
mediclog	754	17904	4.44	20	45
moneywallet	51744	3317	4.69	11	157
morse	1083	5097	3.78	5	19
motioneye-client	12751	1215	4.88	13	21
mundraub-android	10712	1700	4.31	20	398
mynotes	459	352399	4	3	17
notepad	3079	84866	4.38	8	25
onpc	17446	2429	4.86	7	62
openpods	832	66500	3.93	0	3
opensudoku	8003	88362	4.29	13	60
osmdashboard-main	2005	74	0	0	80
osmtracker-android	6947	7039	4.56	5	43

Table 2: Apps that comprise our sample (continued).

App	LoC	#Installs	Score	#Lint	#MATE
paintroid	11413	8266	3.71	0	116
pdfviewer	1279	154	0	1	60
phonograph	22340	21	0	21	40
phyphox-android	17149	66962	3.19	20	6
pilfershushjammer	2641	30078	4.16	0	31
pin	1290	346	0	4	9
piwigo-android	4744	253131	3.63	2	13
planes	4237	1533	4.77	3	24
presencepublisher	2885	203	4.5	0	23
priv-2048	3110	27984	4.39	68	320
priv-boardgame-clock	6604	2	0	26	68
priv-ct-exercises	5607	16229	4.49	41	119
priv-dame	2195	2070	0	10	156
priv-finance-manager	4985	2988	1.32	24	135
priv-footracker	3211	220	0	16	85
priv-itimer	3964	26567	4.39	38	99
priv-minesweeper	3664	9642	4.63	18	54
priv-wifimang	3711	29966	4.17	10	38
pslab-android	33235	854646	4.47	97	99
quickref	1365	9	0	3	36
redreader	31630	12262	0	21	320
rsandroidapp	7094	641	4.71	10	65
rxdroid	16512	307	0	23	164
safeprice	407	1890	3.67	0	13
sdbviewer	1454	475	3.67	0	29
secondscreen	8829	668	4.43	8	26
secure-login	8012	293	2.5	28	76
selfnet-wificonfig	290	2518	4.08	0	18
shelter	3424	903	4.47	2	6
simple-keyboard	14107	2390	3	2	2
simply-pace	717	2343	4.73	1	37
sine-android	1337	41574	4.58	10	358
siteswap_generator	5457	14195	4.48	18	117
sleepywifi	417	24	0	0	8
sumatoradictionary	6928	631	0	1	18
sync-fdroid	15940	3872	4.76	0	26
tedit	7178	1497	0	62	79
termbot	17463	363873	4.39	19	171
todoagenda	9412	1435	4.78	1	3
towercollector	13563	27710	4.46	30	31
trekarta	128511	693	4.33	59	76
unicodepad	4632	10800	4.47	6	671
video-transcoder	2918	6576	4.15	7	51
vol-android	1689	23470	4.49	3	26
weatherradar	1379	1906	4.31	0	56
wifisetup	330	62949	3.17	0	17
wonderdroid-x	2460	59998	4.16	1	50
world-scribe	7331	10530	4.57	24	181
yalpstore	18084	51444	3.92	14	150
your-local-weather	26511	5491	3.95	43	102
Min	250	2	0.0	0	2
Max	250,192	8,005,903	5.0	148	1109
Mean	10,397	207,802	3.2	11.70	97.06
Std Dev	27,449.8	994,707.9	1.8	20.33	166.47
Median	3,468	4,177	4.1	4	37.5
MAD	4,078.6	6,092.0	0.7	12.41	95.77

Lint's output is a report that lists potential code errors and issues related to several properties like correctness, performance, security, usability, and so on. Table 3 lists five different accessibility issues detected by Lint and the associated WCAG principle.

MATE [10] is a tool based on dynamic analysis, thus requires the execution of the app under test on a device or emulator. MATE automatically explores the behavior of an app via random inputs (user events) while accessibility checks are performed in the widgets reached during the tests. Due to MATE's random behavior, we ran it three times in all apps and the highest number of detected issues was taken into account. Table 3 shows the nine different issues detected by MATE. Considering both tools, 14 types of accessibility issues were considered in the evaluation of the apps of our sample.

3.4 Threats to Validity

There are some threats to the validity of our study.

Sample selection. As with many software engineering experiments, it is not easy to guarantee the representativeness of the apps used. There was still the aggravating factor of the set having only native apps. To mitigate this threat, we selected a diverse set of

Table 3: Accessibility issues detected by Lint and MATE.

Tool	Issue Name	Principle
Lint	ContentDescription	Perceivable
	GetContentDescriptionOverride	Perceivable
	KeyboardInaccessibleWidget	Operable
	LabelFor	Understandable
	ClickableViewAccessibility	Robust
MATE	ContrastMinimum	Perceivable
	DuplicateContentDescription	Perceivable
	IdentifyInputPurpose	Perceivable
	NonTextContent	Perceivable
	Orientation	Perceivable
	PageTitled	Operable
	Spacing	Operable
	TargetSize	Operable
	LabelOrInstructions	Understandable

open-source apps from F-Droid repository, available in Google Play, and with recent updates. F-Droid has been used as a repository in many other studies in mobile app testing [4, 6, 9, 10, 14, 18].

Mapping limitation. The map between code elements and the WCAG principles was done manually. Therefore some code elements of the Android API may have been left out. To mitigate this problem, we cover the code elements used in recent accessibility studies [7, 8, 10, 22] and official Google accessibility guides [11, 12].

Flaws in the implementation. There may be implementation errors in any of tools used in our study: Prof.Mapp, Lint, and MATE.

Accessibility evaluation. We used the accessibility issues found by Lint and MATE as proxies of the overall accessibility of the apps of our sample. However, automated tools are able to find accessibility issues related to a small subset of accessibility guidelines and recommendations [20]. Therefore, a more robust accessibility evaluation should also involve specialists and real users, which is not practical for large studies. Despite the general limitation of automated approaches, they can detect accessibility issues that could be easily avoided if developers were aware of basic accessibility concepts. In that case, we assume that the presence of simple issues may imply that more complex guideline violations also occur.

4 ANALYSIS OF RESULTS

For each research question, this section shows the analysis, findings, and implications of the results of our study.

4.1 RQ1 - Adoption of Code Elements

Figure 2 shows in how many apps a given accessibility code element presented in Table 1 appears, grouped by the corresponding WCAG principle[5]. DIA elements are represented by darker gray bars, while IIA by lighter ones. Notice that 17 out of the 55 accessibility code elements (see Table 1) selected for our investigation were not found in any app. Regarding principle *Perceivable*, ≈ 70% (18 out of 26) of identified code elements are found in the apps (Figure 2(a)). Elements related to this principle appear in the highest number of apps and are among the four most used: `background` in 99 apps (≈ 90%), `textColor` in 92 apps (≈ 83%), `textSize` in 91 apps (≈ 82%), and `inputType` in 79 apps (≈ 71%).

[5]The analysis focus on the presence in apps as their frequency has a significant strong correlation to the app' size.

Figure 2(b) shows results for *Operable*. While 8 out of 10 code elements that appear are DIA, they are adopted in 8 apps or less. The two elements of *Understandable* are shown in Figure 2(c); `Hint` occurs in 66 apps and `labelFor` in 12 apps. Figure 2(d) presents the results for *Robust*. Element `onBindViewHolder` is IIA and occurs in 46 apps. Other elements are all DIA but appear in less than 7 apps.

Figure 3 illustrates how many different code elements are used in each app of our sample. Only one app (`democracy-client`) employs 26 types of code elements (≈ 47%), while 2 apps do not use any code element. On average, the apps adopt 6.4 different types of code elements (median 6). If only code elements that directly impact on accessibility (namely, DIA) are taken into account, only 3.6 different types are used on average.

> *Answer to RQ1: Code elements that may influence accessibility are not widely used: only one fifth of code elements occur in more than 10% of the apps; around 31% of code elements do not appear at all. On average, apps use 6.4 different types of elements, and fewer (3.6) of the ones that directly impact accessibility.*

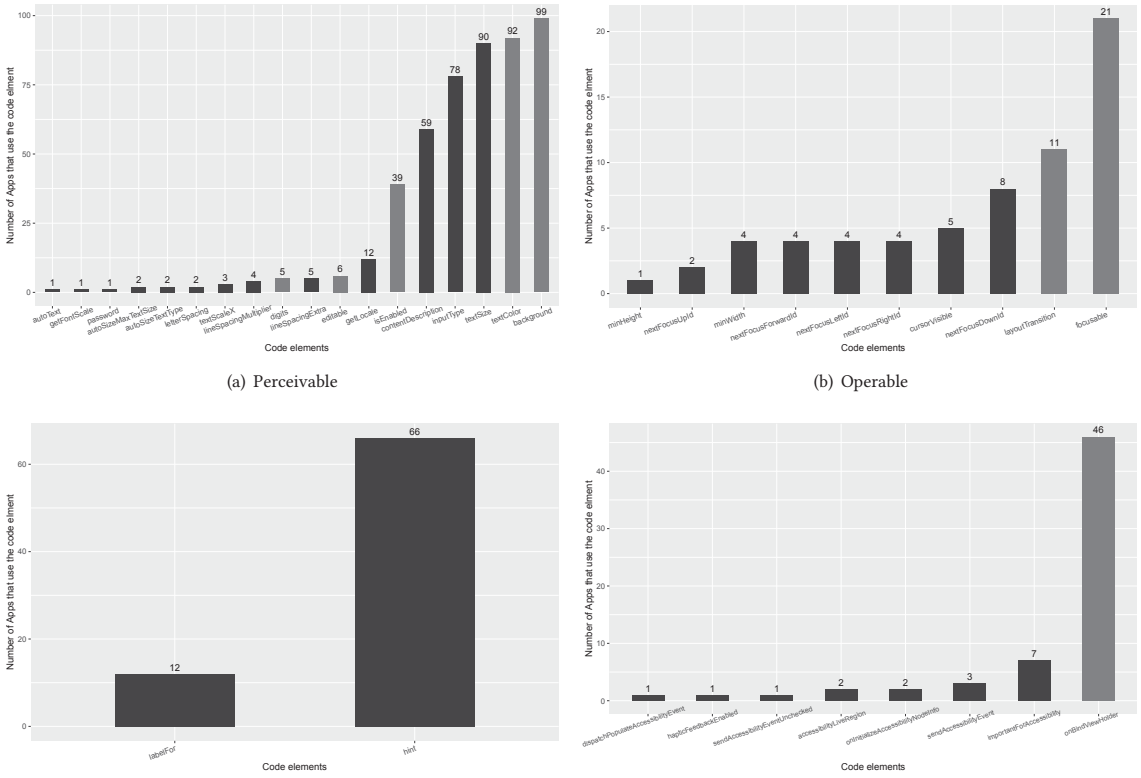
Implications: While most used elements are related to general elements of GUIs, the ones directly impacting accessibility are unusual or not used at all. This can be interpreted from two points of view. The first one is that developers are not deliberately targeting an app more accessible for people with disabilities. While this might be the case for specific domains like games, we believe that the lack of awareness or knowledge is the main reason [2, 3]. The results for this question along with the mapping in Section 2 could improve the current understanding of how to implement an accessible app. Moreover, the identification of code elements in open source apps may help to construct real examples for training materials in mobile accessibility and Android.

4.2 RQ2 - Accessibility Issues

In this study, we found a total of 12,108 accessibility issues. The median for the sample is 44 issues; each app has at least 3 issues, and up to 1,115 issues. The last two columns of Table 2 report the number of accessibility issues detected by Lint and MATE, respectively. For all apps, Lint detected 1,299 issues, 11.7 per app on average, while MATE detected 10,809 issues, and an average of 97.06 issues per app. MATE detected 8 times more issues than Android Lint because it checks violations related to a great number of success criteria.

Lint. In this sample, 29 apps (26.1%) do not present any accessibility issues detected by Lint; issue `GetContentDescriptionOverride` was not detected in any app. Figure 4 shows boxplots of the number of issues detected by Lint, divided by principle (see Table 3). The largest amount of issues is from the *Perceivable* principle, 789 out of 1299 (60.7%). Its related issue `ContentDescription` occurs in 65 apps. The next is *Understandable* with 243 issues (18.7%); `LabelFor` appears in 37 apps. Principle *Robust* has 147 issues (11.3%) and `ClickableViewAccessibility` occurs in 38 apps. Finally, *Operable* has 120 issues (9.23%) and its related issue `KeyboardInaccessibleWidget` occurs in 26 apps.

MATE. In this study, MATE uncovered accessibility issues (at least 2) in all 111 apps. Figure 5 shows boxplots of the number of



(a) Perceivable

(b) Operable

(c) Understandable

(d) Robust

Figure 2: The relation between code elements and #Apps that use it.

issues detected by MATE, divided by principle. While MATE can detect more types of issues (see Table 3), no issue is related to principle *Robust*. The number of issues found for principle *Perceivable* is also the highest with 6,546 out of 10,809 (60.6%). Its related issues `NonTextContent`, `IdentifyInputPurpose`, `ContrastMinimum`, `DuplicateContentDescription`, `Orientation` occur in 95, 34, 106, 3, and 108 apps, respectively. Principle *Operable* has 3,968 issues associated (36.7%); its issues `TargetSize`, `Spacing`, and `PageTitled` appear in 101, 49, and 88 apps, respectively. Finally, *Understandable* has 295 issues (2.7%), and its related issue `LabelOrInstructions` occurs in 67 apps.

> *Answer to RQ2: Accessibility issues are common in Android apps. All apps have at least 3 accessibility issues; on average, an app has 109.1 issues (a density of 0.04 issue per LoC). Most issues detected by Lint and MATE are related to principle Perceivable.*

Implications: While detected issues are not sufficient to assess the overall accessibility of an app given the limitation of automated tools, it provides a sensible indicator of lack of accessibility. The current scenario is worrisome; even simpler accessibility issues detected by Android Lint (integrated in Android SDK) are present in most apps, hindering the app usage by people with disabilities. We believe that a large portion of accessibility issues could be fixed automatically and motivates further research on program repair techniques. Software engineering bots integrated to development environments (e.g. GitHub) could also help to sensitize developers.

4.3 RQ3 - A Look at the Correlation Between Code Elements and Accessibility Issues

We set out to look at possible correlations between code elements and accessibility issues. To this end, we decided to combine some code-related metrics into multi-metric indices. We argue that this provides a more intuitive and robust assessment of the relation between a set of code elements and accessibility issues. The raw numbers were also normalized by LoC to cope with different sizes of apps. Therefore, occurrences of the 55 code elements were condensed in Compounded Metrics (prefix '`CM-`' in Figures 6 and 7). Prefix `CM-Sum` represents the sum of values for related elements; for instance, `CM-SumRobustCodeElements` is the sum of all occurrences of code elements related to principle *Robust*. Prefix `CM-Index` represents how many different related accessibility code elements occurred; for example, `CM-IndexDIACodeElements` tells us the number of different DIA elements found in a given app. Google Play metrics (Table 2) are also taken into account (prefix `GP-`).

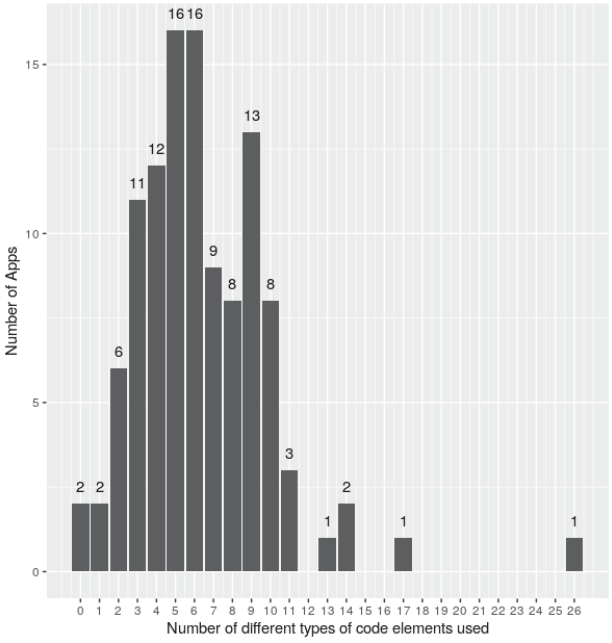


Figure 3: Usage of different types of code elements.

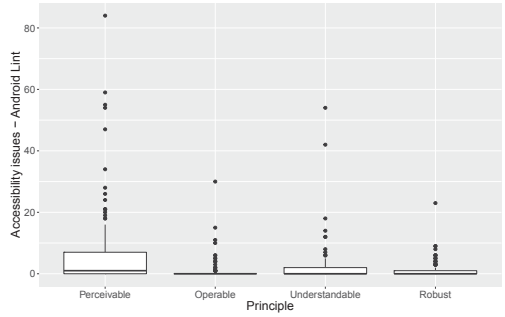


Figure 4: Accessibility issues x principles (Android Lint).

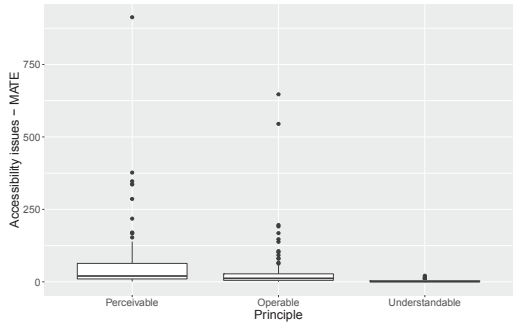


Figure 5: Accessibility issues x Principles (MATE).

Figure 6 shows the heatmap of the correlation between multi-metrics regarding code elements (x-axis) and accessibility issues from both Lint and MATE (y-axis). Interestingly, there is a medium, negative correlation between the number of installs (i.e., `GP-installs`) and `M-Orientation` (ρ = -0.338, p-value < 0.05). Even though users generally do not mention accessibility issues in their reviews [9], this might indicate that as apps become more popular, developers use the feedback from users to sort out orientation-related issues. A subtle indication that this might be the case is the small negative correlation between app ratings (i.e., `GP-score`) and orientation problems (ρ = -0.139, p-value = 0.146):[6] ratings tend to go down as orientation problems stay present in the app for longer.

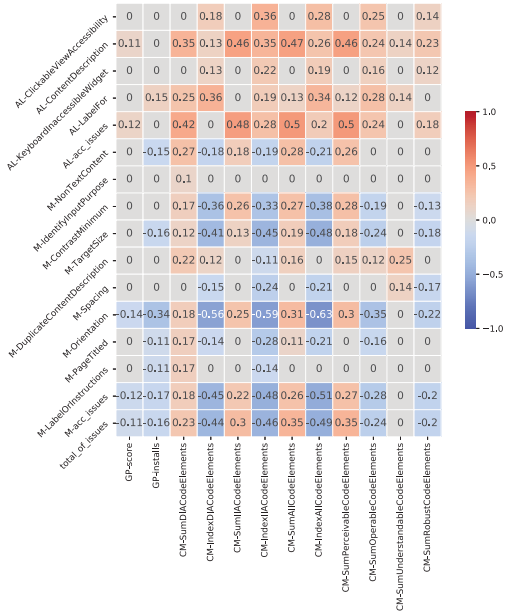


Figure 6: Heatmap of the correlation between code elements and accessibility issues from Lint (prefixed by `AL-`) and MATE (prefixed by `M-`).

It is worth highlighting that Figure 6 also shows a large positive correlation between all accessibility issues found by Android Lint (i.e., `AL-acc_issues`) and `CM-SumPerceivableCodeElements` (ρ = 0.502, p-value < 0.05). There is also a large positive correlation between `AL-acc_issues` and `CM-SumAllCodeElements` (ρ = 0.501, p-value < 0.05). These and other positive correlations may be explained by code elements being a predictor to the number of widgets in the app. So the greater the number of widgets, the higher the number of issues.

An interesting trend is related to `CM-Index` metrics: they have medium and large, negative correlation with different issue-related metrics. For instance, `CM-IndexAllCodeElements` and the total of issues has a correlation ρ = -0.49, p-value < 0.05. This indicates that apps that adopt more different types of code elements tend to have fewer accessibility issues per LoC. This might be an indication that developers are aware of the importance of setting certain properties and using specific resources for the overall app accessibility.

[6]Despite that this correlation is not statistically significant, we believe that it can be used to explain our results.

We also examined accessibility issues from the standpoint of how these issues can be mapped into violations of the WCAG guidelines. As shown in Figure 7, when interpreted in the light of violations of the WCAG guidelines, there is only two large negative correlation: `CM-IndexAllCodeElements` has a negative correlation with `W-operable_issues` (ρ = -0.55, p-value < 0.05), and `W-operable_issues` also has a negative correlation with `CM-IndexIIACodeElements` (ρ = -0.506, p-value < 0.05). Most positive medium correlations come from the way `W-perceivable_issues` interacts with `CM-SumPerceivableCodeElements` (ρ = 0.445, p-value < 0.05), `CM-SumAllCodeElements` (ρ = 0.441, p-value < 0.05), and `CM-SumIIACodeElements` (ρ = 0.405, p-value < 0.05). Principles *Understandable* and *Robust* have mostly small or no correlation. We believe that this is the case because principles *Perceivable* and *Operable* are more prevalent in both code elements and issues.

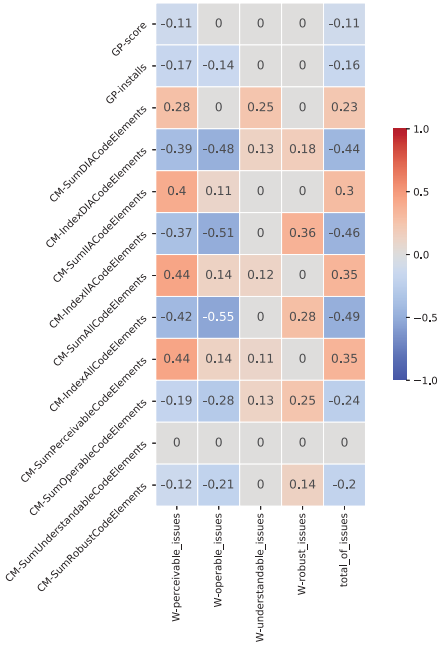


Figure 7: Heatmap of the correlation between code elements and accessibility issues grouped according to violations of the WCAG guidelines.

Answer to RQ3: Apps that adopt different types of code elements tend to have a smaller density of accessibility issues.

Implications: The trend of a diverse usage of code elements may imply fewer issues reinforces the belief that the developers' proficiency and awareness in mobile accessibility result in more accessible apps. The relations between code elements and accessibility issues gave some evidence that further advances with static analysis could be achieved. Furthermore, there is room for leveraging static analysis data to improve testing tools. In general, we observe that more effort is required for principles *Understandable* and *Robust*. As they involve a semantic understanding of accessibility aspects, this is a challenge for automated tools and open research opportunities.

All in all, we expect that the results herein presented help to pave the way for more effective static and dynamic analysis tools that leverage the knowledge of such code elements. Moreover, this study may support researchers and practitioners to make more educated decisions with respect to accessibility in Android apps.

5 RELATED WORK

There exists an effort to provide tools that automate the analysis of accessibility in mobile apps. Besides the tools we adopted in this study [13, 20] (Section 3.3), some of them are described as follows.

Oliveira and Filgueiras [8] present AccessibiLint, a tool for static code verification of accessibility rules for Android apps. Their tool was developed as an extension of Android Lint to improve accessibility in the early stage of the project's development. De Moura et al. [7] propose an API that can automatically run tests, analyzing whether the GUIs of an Android app conform to the rules proposed by TalkBack. The aim is to help developers ensure accessibility standards and thus make it feasible to develop apps. We plan to replicate this study to include other tools.

Other direction is to conduct empirical studies that aim to characterize accessibility in mobile apps. Acosta-Vargas et al. [1] combine a manual review with the guidelines of WCAG 2.1 and an automatic review with the app Accessibility Scanner Validator. They evaluated 10 popular Android apps. The results indicate that rated mobile apps violate the accessibility levels recommended by the World Wide Web Consortium.

Yan and Ramachandran [24] evaluate the status of accessibility in mobile apps. They investigate 479 Android apps and introduce two coverage metrics: inaccessible element rate (IAER) and accessibility issue rate (AIR). These metrics estimate the percentage of accessibility issues identified by the automated tool IBM Mobile Accessibility Checker (MAC). The IAER score showed that approximately 30% of the GUI elements had accessibility issues, and the AIR score indicated that 15% of the accessibility issues remained and need to be fixed to make the apps accessible.

Alshayban et al. [2] present results of a large-scale to understand the accessibility from three complementary perspectives: app, developers, and users. First, they analyze the prevalence of accessibility issues in over 1000 Android apps. Then they investigate the developer sentiments through a survey. In the end, they investigate user ratings and app popularity. Their analysis revealed that inaccessibility rates for apps developed by big companies are relatively similar to inaccessibility rates for other apps.

Vendome et al. [22] conducted an empirical study about accessibility in Android apps. First, they performed a pilot study with Android apps hosted in GitHub, looking for usage of accessibility APIs and content descriptors. They found that 36.96% of the apps do not have any labeled element and only 2.08% of the apps imported at least one Accessibility API. Then, its main contribution was the analysis of StackOverflow discussions that were collected and classified (using open-coding) into 58 categories related to accessibility aspects. This pilot study is the most similar to ours. The main differences are (i) we provided a more comprehensive identification of accessibility-related Android API as code elements, (ii) mapped them to WCAG, and (iii) related such elements to real accessibility issues detected by automated tools. To our best knowledge, this

is the first study that investigates source code elements that may impact on an app's accessibility, and how they are related to issues detected via automated tools. Our findings contribute to improve existing evaluation tools.

6 CONCLUDING REMARKS

The Android API has a rich set of features, some of which are aimed at helping developers to implement more accessible apps. However, our results suggest that those code elements are not widely employed in practice. This is particularly worse when only accessibility-specific elements are taken into account. Although accessibility evaluation should be performed with objective and subjective criteria in mind, accessibility issues found by automated tools suggest that properly setting accessibility elements tends to lead to more accessible apps and that statically-set properties could be further explored to support automatic accessibility evaluations.

Table 1 is one of the artifacts produced by this research and is available. Developers and researchers can use the exposed map of code elements and principles of the WCAG guide. The main objective is to facilitate the implementation of more accessibility apps. Also, anyone can suggest mofidications to the proposed mapping so that it continues to accurately reflect the information within the official Android API.

As future work, we plan to improve the representativeness of our sample by (i) increasing the number of apps; (ii) updating the Prof.Mapp tool parser so as it allows for the analysis of hybrid Android apps; (iii) take into account the analysis of SMALI intermediate code, so that it is possible to include in our samples apps whose source code is not available.

ACKNOWLEDGMENTS

This work is partially supported by the Brazilian agencies CAPES, CNPq and FAPESP (Andre T. Endo grant nr. 420363/2018-1; Silvia Regina Vergilio grant nr. 305968/2018-1 and Marcelo Medeiros Eler grant nr. 18/12287-6).

REFERENCES

[1] Patricia Acosta-Vargas, Luis Salvador-Ullauri, Janio Jadán-Guerrero, César Guevara, Sandra Sanchez-Gordon, Tania Calle-Jimenez, Patricio Lara-Alvarez, Ana Medina, and Isabel L. Nunes. 2020. Accessibility Assessment in Mobile Applications for Android. In *Advances in Human Factors and Systems Interaction*, Isabel L. Nunes (Ed.). Springer International Publishing, Cham, 279–288.

[2] Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek. 2020. Accessibility Issues in Android Apps: State of Affairs, Sentiments, and Ways Forward. In *International Conference on Software Engineering (ICSE)*, Vol. 12. ACM.

[3] Humberto Lidio Antonelli, Sandra Souza Rodrigues, Willian Massami Watanabe, and Renata Pontin de Mattos Fortes. 2018. A survey on accessibility awareness of Brazilian web developers. In *Proceedings of the 8th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion*. 71–79.

[4] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2016. Automated test input generation for android: Are we there yet?. In *Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*. 429–440.

[5] Jacob Cohen. 1988. *Statistical Power Analysis for the Behavioral Sciences* (2nd ed.). Routledge Academic. 567 pages.

[6] Henrique Neves da Silva, Paulo Roberto Farah, Willian Douglas Ferrari Mendonça, and Silvia Regina Vergilio. 2019. Assessing Android Test Data Generation Tools via Mutation Testing. In *Brazilian Symposium on Systematic and Automated Software Testing*. ACM, 32–41.

[7] Cícero Joasyo Mateus De Moura, Souza De Oliveira, Kenyo Abadio Crosara Faria, and Eduardo Noronha De Andrade Freitas. 2018. A New API for Android Accessibility Testing. In *Proceedings - 2017 International Conference on Computational Science and Computational Intelligence, CSCI 2017*. IEEE, 594–598.

[8] Lucia Vilela Leite de Oliveira, Arthur Floriano Barbosa Andrade Filgueiras. 2019. AccessibiLint: A Tool for Early Accessibility Verification for Android Native Applications. In *IHC '19: Proceedings of the 18th Brazilian Symposium on Human Factors in Computing Systems*.

[9] Marcelo Medeiros Eler, Leandro Orlandin, and Alberto Dumont Alves Oliveira. 2019. Do Android app users care about accessibility?: An analysis of user reviews on the Google play store. In *IHC 2019 - Proceedings of the 18th Brazilian Symposium on Human Factors in Computing Systems*. ACM.

[10] Marcelo Medeiros Eler, Jose Miguel Rojas, Yan Ge, and Gordon Fraser. 2018. Automated Accessibility Testing of Mobile Apps. In *Proceedings - 2018 IEEE 11th International Conference on Software Testing, Verification and Validation, ICST 2018*. IEEE, 116–126.

[11] Google. 2020. Basic Android Accessibility: making sure everyone can use what you create! https://codelabs.developers.google.com/codelabs/basic-android-accessibility/#0 Google Codelabs.

[12] Google. 2020. Build more accessible apps. https://developer.android.com/guide/topics/ui/accessibility Google guide.

[13] Google. 2020. Improve your code with lint checks. https://developer.android.com/studio/write/lint Android Lint.

[14] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. [n. d.]. *Practical GUI Testing of Android Applications via Model Abstraction and Refinement (APE)*. Technical Report. http://gutianxiao.com/ape

[15] Susanne Iwarsson and A Ståhl. 2003. Accessibility, usability and universal design - Positioning and definition of concepts describing person-environment relationships. *Disability and rehabilitation* 25 (02 2003), 57–66.

[16] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. 2015. Understanding the Test Automation Culture of App Developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–10.

[17] Jonathan Lazar. 2019. *Web Accessibility Policy and Law*. Springer, London, 247–261.

[18] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-Objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, 94–105.

[19] World Health Organization and World Bank. 2011. World report on disability.

[20] Camila Silva, Marcelo Medeiros Eler, and Gordon Fraser. 2018. A Survey on the Tool Support for the Automatic Evaluation of Mobile Accessibility. In *Proceedings of the 8th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-Exclusion (DSAI 2018)*. ACM, 286–293.

[21] Davi Bernardo Silva, Marcelo Medeiros Eler, Vinicius H.S. Durelli, and Andre Takeshi Endo. 2018. Characterizing mobile apps from a source and test code viewpoint. *Information and Software Technology* 101 (2018), 32–50.

[22] Christopher Vendome, Diana Solano, Santiago Linán, and Mario Linares-Vásquez. 2019. Can everyone use my app? An Empirical Study on Accessibility in Android Apps. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 41–52.

[23] Markel Vigo, Justin Brown, and Vivienne Conway. 2013. Benchmarking Web Accessibility Evaluation Tools: Measuring the Harm of Sole Reliance on Automated Tests. In *Proceedings of the 10th International Cross-Disciplinary Conference on Web Accessibility (W4A '13)*. ACM, New York, NY, USA, Article 1, 10 pages.

[24] Shunguo Yan and P. G. Ramachandran. 2019. The Current Status of Accessibility in Mobile Apps. *ACM Trans. Access. Comput.* 12, 1, Article 3 (Feb. 2019), 31 pages.