UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DAVI SIMÕES FREITAS

Uma infraestrutura assistida por robô para detecção de perda de dados em aplicativos Android.

Recife

2022

DAVI SIMÕES FREITAS

Uma infraestrutura assistida por robô para detecção de perda de dados em aplicativos Android.

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Área de Concentração: Engenharia de Software e Linguagens de Programação

Orientador (a): Breno Alexandro Ferreira de Miranda

Coorientador (a): Juliano Manabu Iyoda

Recife

2022

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

F866i Freitas, Davi Simões
Uma infraestrutura assistida por robô para detecção de perda de dados em aplicativos Android / Davi Simões Freitas. – 2022.
86 f.: il., fig., tab.

Orientador: Breno Alexandro Ferreira de Miranda.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2022.

Inclui referências.

1. Engenharia de software. 2. Teste de software. 3. Robótica. I. Miranda, Breno Alexandro Ferreira de (orientador). II. Título.

005.1 CDD (23. ed.) UFPE - CCEN 2022-152

Davi Simões Freitas

“Uma infraestrutura assistida por robô para detecção de perda de dados em aplicativos Android”

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Engenharia de Software e Linguagens de Programação.

Aprovado em: 23 de agosto de 2022.

BANCA EXAMINADORA

_____

Prof. Dr. Alexandre Cabral Mota
Centro de Informática / UFPE

_____

Prof. Dr. Márcio de Medeiros Ribeiro
Instituto de Computação / UFAL

_____

Prof. Dr. Breno Alexandro Ferreira de Miranda
Centro de Informática / UFPE
(**Orientador**)

RESUMO

Quando uma aplicação Android é interrompida por uma chamada ou quando a orientação do dispositivo é alterada (de retrato para paisagem ou vice-versa), o sistema operacional pode precisar destruir e recriar a atividade. Idealmente os dados e o estado da aplicação deveriam ser salvos (antes da destruição) e restaurados (após a recriação) para evitar perda de dados. Infelizmente, o sistema operacional Android não gerencia estes cenários nativamente e os desenvolvedores de aplicativos precisam explicitamente salvar e restaurar os dados da aplicação. Caso contrário, os usuários podem observar falhas de perda de dados (i.e., informações que estavam presentes antes da interrupção são perdidas após a recriação da atividade). Um trabalho recente (RIGANELLI et al., 2020) propôs uma abordagem automatizada para a detecção de perda de dados em aplicativos Android: casos de teste são gerados para explorar a interface do aplicativo e, durante a exploração, a aplicação é interrompida a partir do acionamento, via software, de uma rotação na tela. Capturas de tela e os dados do aplicativo antes e depois da rotação são comparados para verificar se dados foram perdidos. Tal proposta, entretanto, é i) pouco realista e ii) invasiva. Pouco realista porque a alteração na orientação da tela é acionada via software —— e não pelos sensores como aconteceria em um cenário real; e invasiva porque os comandos são enviados através de comunicação com o dispositivo via cabo USB ou dispositivo simulado. Este trabalho propõe a utilização de um braço robótico artesanal para proporcionar um ambiente de testes mais realista: os eventos de destruição e recriação das atividades são acionados pela rotação física do smartphone. A infraestrutura proposta também permite o envio de comandos sem a necessidade de conexão física com o dispositivo via cabo USB, um passo importante na direção de uma solução realista e não-invasiva. Para avaliar a viabilidade de utilização da infraestrutura proposta, uma avaliação empírica foi realizada considerando 77 aplicativos Android e 341 falhas de perda de dados foram identificadas. Todas as falhas identificadas foram reportadas aos desenvolvedores e, das 201 falhas que já foram analisadas, 180 (89,55%) foram confirmadas pelos desenvolvedores.

Palavras-chaves: teste de software; android; perda de dados; robótica para testes.

ABSTRACT

When an Android application is interrupted by a call or changes the device's orientation (from portrait to landscape or vice versa), the operating system may need to destroy and recreate the activity. Ideally, data and application state should be saved (before destruction) and restored (after recreated) to avoid data loss. Unfortunately, the Android operating system does not manage these scenarios natively, and application developers explicitly need to save and restore application data. Otherwise, users may experience data loss failures (i.e., information that was present before the outage is lost after the activity is recreated). Recent work (RIGANELLI et al., 2020) has proposed an automated approach to detecting data loss in Android applications: test cases are generated to explore the application's interface, and during exploration, the application is interrupted by triggering, via software, a rotation on the screen. Screenshots and application data before and after rotation are compared to see if data is lost. Such a proposal, however, is i) unrealistic and ii) invasive. Unrealistic because the change in screen orientation is triggered via software — and not by sensors as would happen in a real scenario, and invasive because commands are sent via communication with the device via USB cable or simulated device. This work proposes the use of a handcrafted robotic arm to provide a more realistic testing environment: the events of destruction and recreation of activities are triggered by the physical rotation of the smartphone. The proposed infrastructure also allows sending commands without the need to physically connect the device via a USB cable, an essential step towards a realistic and non-invasive solution. To assess the feasibility of using the proposed infrastructure, an empirical evaluation was performed considering 77 Android applications, and 341 data loss failures were identified. All identified failures were reported to the developers, and of the 201 bugs that have already been analyzed, 180 (89,55%) have been confirmed by the developers.

Keywords: software testing; android; data loss; robotics for testing.

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Os smartphones se tornaram uma ferramenta essencial na vida das pessoas por se tratar de um dispositivo portátil, de fácil uso e por possuir aplicativos e recursos que facilitam a vida cotidiana em várias dimensões (trabalho, entretenimento, compras e comunicação). Desenvolver aplicativos com qualidade para esses dispositivos traz grandes desafios. Entre os principais motivos está a grande diversidade de hardware e de versões do sistema operacional, o que faz com que o desenvolvedor precise testar seu aplicativo em mais de um ambiente de configuração (WEI; LIU; CHEUNG, 2016). Por exemplo, um desenvolvedor que lança um aplicativo para a plataforma Android tem que desenvolver em compatibilidade com até cinco versões para alcançar um número maior de usuários (ANDROIDSTUDIO, 2022). Neste cenário, o desenvolvedor precisa escolher entre ter mais de uma versão do aplicativo ou adiar a inclusão dos novos recursos do Android. Diante de tamanha variação de sistemas para testar, o desenvolvedor deve investir em testes que utilizem processos automatizados e que contemplem combinações de diferentes características de hardware e versões do sistema operacional.

Em aplicações Android, todas as interações com o usuário são realizadas pelo componente Activity. Nela, uma janela é criada e são implementadas funcionalidades relacionadas ao propósito do aplicativo. Quando o usuário navega pelo aplicativo, a Activity transita em diferentes estados. Dependendo da quantidade de recursos e o tipo do evento recebido, a Activity pode ser temporariamente destruída, com a finalidade de liberar recursos, para ser posteriormente reconstruída. Esses eventos que destroem as Activities e depois as reconstroem são chamados de eventos *stop-start*. Por exemplo, uma rotação do celular mudando a orientação de Retrato para Paisagem, ou uma chamada telefônica produzem o evento *stop-start*. A destruição e reconstrução da Activity ocorre com frequência e estes eventos podem resultar em problemas de perda de dados (*data loss*). Quando o desenvolvedor não implementa corretamente os métodos para guardar o estado da Activity antes da destruição para poder reconstruí-la corretamente depois, coloca-se o aplicativo em um estado inconsistente. Isso pode levar o aplicativo a perder dados inseridos pelo usuário, informações que estavam na tela, ou até mesmo travar o aplicativo. Por exemplo, no aplicativo Twitter, ao se digitar uma busca no teclado e mudar a orientação do celular de retrato para paisagem (provocando a destruição e reconstrução da Activity), perde-se o texto digitado. Um problema de perda de dados ocorre quando os dados são excluídos acidentalmente ou as variáveis de estado são atribuídas acidentalmente com

valores padrão ou iniciais (RIGANELLI et al., 2020).

Outro problema relacionado aos testes de dispositivos móveis é a forma de testar movimentos gestuais como rotações, chacoalhos e torções de punho. Métodos automatizados tradicionalmente utilizados para aplicativos *desktop* não conseguem testar essas particularidades dos dispositivos móveis. Para resolver esse desafio, algumas técnicas exercitam o celular enviando, via *software*, eventos falsos desses gestos (HANS, 2015; ROBOTIUM, 2022; UIAUTOMATOR, 2022). Tais testes são invasivos: os gestos são enviados ao dispositivo via cabo USB ou via instrumentação de código e são, no melhor caso, uma simulação de um gesto real feito diretamente no dispositivo. Testes em ambientes simulados e invasivos são adequados para as fases intermediárias do desenvolvimento do dispositivo, mas nenhum fabricante confia plenamente em testes em ambientes simulados antes de levar o produto para as prateleiras. Nas fases finais de desenvolvimento, os dispositivos são necessariamente submetidos a testes **não-invasivos**, com gestos feitos no dispositivo, e não via *software* em ambientes simulados. Para tal, usualmente, utilizam-se pessoas para executar tais testes: uma prática cara, tediosa e passível de erros.

Nos testes automatizados com suporte robótico, o testador adapta as atividades repetitivas para serem executadas em um smartphone por um robô, que reproduz o comportamento de uma pessoa. Essa abordagem tem se mostrado muito promissora, pois podem ser executadas com bastante precisão independentemente do número de repetições, sem sofrer a influência do erro humano por desatenção ou fadiga. No entanto, essas contribuições ainda precisam da experiência do testador ou de um bom conjunto de testes para criar os *scripts* de testes automatizados.

Data Loss Detector (DLD) é uma ferramenta que se propõe a identificar, automaticamente, problemas de perda de dados em dispositivos Android. DLD combina estratégias de exploração sistemática e aleatória, reduzindo a intervenção do testador a escolha de parâmetros para a estratégia de exploração e o tempo do teste. Em sua versão original, o DLD realiza os testes em um ambiente simulado (o *Android Virtual Device*, ou AVD) por meio de comandos que simulam inputs dos sensores e interações do usuário.

Nesse trabalho propomos uma infraestrutura assistida por robô para a detecção de perda de dados em aplicativos Android. Essa infraestrutura permite realizar testes menos invasivos e mais realistas do que as abordagens concorrentes, pois a aplicação interage diretamente com os sensores do *smartphone* ao invés de simular mudança de orientação por comandos ADBs.

O robô foi construído utilizando materiais de baixo custo (placa Arduino, motor de passos,

fonte de energia e chassis), um aspecto importante para viabilizar a adoção da ferramenta em ambientes de teste. A aplicação foi construída no ambiente de desenvolvimento do Arduino e a integração com o DLD foi realizada por meio comunicação serial utilizando a biblioteca PySerial em Python. Para verificar se a adição de rotações físicas suportadas pelo braço robótico e a nossa implementação do $\mathrm{R\text{-}DLD}$ influenciavam os resultados obtidos nos estudos realizados com rotação simulada, realizamos uma avaliação preliminar. Além disso, executamos uma replicação estendida do experimento do DLD, usando o braço robótico para a rotação do celular e testando uma quantidade maior de aplicativos. Nossa replicação foi realizada em 77 aplicativos escolhidos aleatoriamente de uma loja Android e encontramos 341 problemas de perda de dados em 67 aplicativos. Todos os *bugs* foram reportados aos desenvolvedores e obtemos respostas em 58,94% (201) dos casos. Deste, 89,55% (180) confirmaram a presença do problema de perda de dados e 35,82%(72) dos casos tiveram os bugs corrigidos.

A pesquisa foi realizada em colaboração com um parceiro industrial que utiliza robôs para automatizar testes em *smartphones* com sistema Android. Esses robôs interagem com o *smartphone* simulando ações semelhantes a um usuário real para realizar testes mais realista, por meio de toques na tela e interações com sensores. O teste de perda de dados poderia ser aplicado imediatamente nessa indústria parceira colaborando com a transferência de tecnologia entre a academia e a indústria.

A organização dos próximos capítulos é descrita a seguir. Conceitos fundamentais para o entendimento do problema e da solução proposta são apresentados no Capítulo 2. O Capítulo 3 apresenta a infraestrutura R-DLD, o seu processo de detecção de perda de dados utilizando o robô e os passos para a sua construção. O Capítulo 4 apresenta uma classificação dos *Bugs* reais e Alarmes falsos de acordo com as características dos alertas. O Capítulo 5 apresenta os detalhes da nossa avaliação empírica, incluindo as nossas questões de pesquisa e estratégia de seleção dos aplicativos testados. Os resultados do nosso estudo são apresentados e discutidos no Capítulo 6. Trabalhos relacionados são apresentados no Capítulo 7, seguidos por conclusão e trabalhos futuros.

2 BACKGROUND

Neste capítulo apresentamos conceitos básicos sobre o ciclo de vida de uma atividade Android e sobre o *Data Loss Detector* (DLD).

2.1 TESTE DE GUI PARA ANDROID

O ecossistema Android possui diversas ferramentas para a automação de testes de GUI. Essas ferramentas fornecem uma alternativa aos testes manuais, pois são mais confiáveis e podem interagir com todas as funcionalidades da aplicação utilizando a mesma interface do usuário. Esses testes podem ser usados para cenários específicos ou percorrer todo o fluxo da aplicação. São úteis para verificar regressão ou testar cenários complexos que envolvam combinações de hardware, aplicativo e versões de sistema operacional.

Espresso (ESPRESSO, 2022) fornece APIs para criar testes automatizados que simulam as interações dos usuários no aplicativo em teste. Esses testes podem ser criados de forma manual ou utilizando Espresso Test Recorder, que grava as interações em um dispositivo e, em seguida, gera o teste de GUI correspondente.

UI Automator (UIAUTOMATOR, 2022) fornece um conjunto de APIs para criar testes de UI que realizam interações em aplicações Android. UI Automator permite programar testes robustos sem que seja necessário conhecer os detalhes de implementação do aplicativo de destino (abordagem caixa preta) e permite operações no dispositivo em que o app está sendo executado por meio da classe UiDevice. O framework apresenta um desempenho melhor nas aplicações que implementam recursos de acessibilidade do Android.

Robotium (ROBOTIUM, 2022) é um framework para automação de testes Android com suporte para aplicativos nativos e híbridos. O Robotium funciona tanto em AVD como em smartphone real e pode executar testes em aplicações mesmo sem o código fonte (abordagem caixa preta).

O AndroidRipper (AMALFITANO et al., 2012) é uma técnica automatizada para testes de GUI que explora automaticamente a interface do aplicativo com o objetivo de interagir de maneira estruturada. Cada elemento da GUI possui um conjunto fixo de propriedades e aceita como entrada eventos gerados pelo usuário ou pelo sistema. O AndroidRipper analisa a GUI para identificar os possíveis eventos desses elementos e gera uma sequência de ações para um caso

de teste, nesse processo, o AndroidRipper cria uma árvore de estado com o conjunto de estados da GUI e transições encontradas. Nesse processo, enquanto explora a GUI, o AndroidRipper detecta falhas em tempo de execução.

2.2 CICLO DE VIDA DA ATIVIDADE

Uma atividade (ou *activity*) é um ponto de entrada para a interação de uma aplicação Android com o usuário (ANDROID, 2022a). Geralmente uma aplicação possui várias atividades: uma principal, que é iniciada ao abrir o aplicativo, e as secundárias, implementadas para realizar ações diferentes como, por exemplo, configurar um recurso ou acessar um serviço externo.

Uma atividade é um componente com reconhecimento de ciclo de vida, ou seja, do momento em que um aplicativo é lançado até a sua destruição, as instâncias das atividades transitam por vários estados e executam alguns *callbacks*. A classe *Activity* fornece um conjunto de sete callbacks (Figura 1): onCreate(), onStart(), onResume(), onPause(), onStop(), onDestroy() e onRestart().

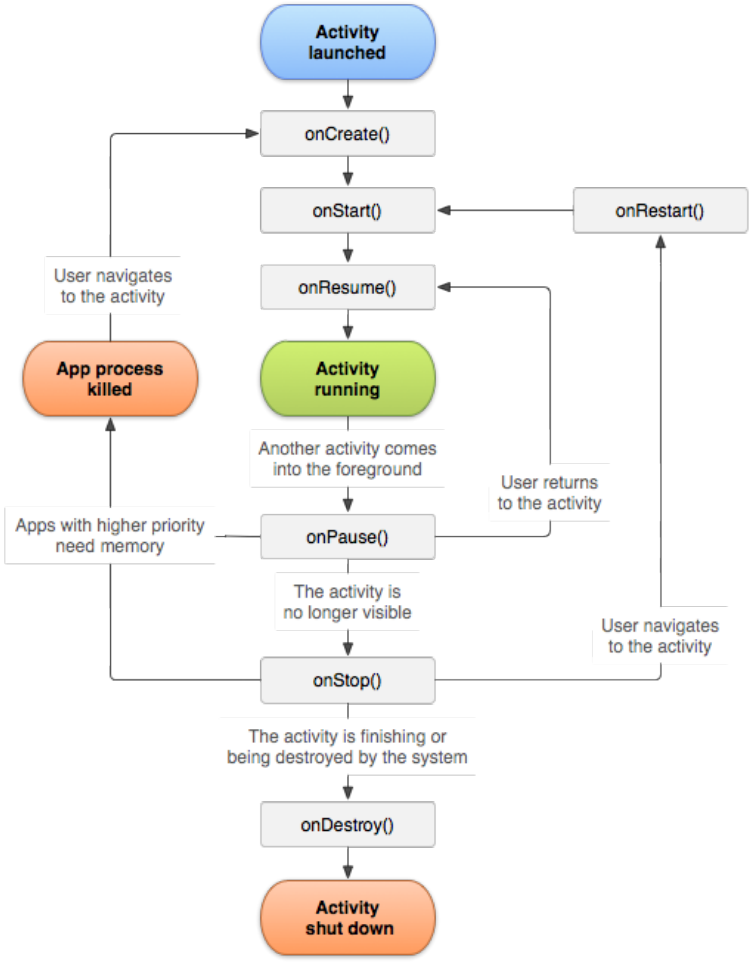
Cada *callback* permite realizar o trabalho específico adequado a determinada mudança de estado da aplicação, e implementá-los de forma correta evita prejudicar a experiência do usuário com travamentos, perdas de dados ou problemas de desempenho. No entanto, não é necessário implementar todos os métodos do ciclo de vida. Enquanto a implementação do onCreate() é obrigatória, a implementação dos outros métodos depende da complexidade e do comportamento da aplicação. Conforme a atividade entra em um novo estado, o sistema invoca cada um desses *callbacks*.

- onCreate() - É acionado assim que o sistema cria a atividade e insere o estado "Criado". Esse *callback* deve ser implementado e executa a lógica básica de inicialização do aplicativo. Recebe o parâmetro *savedInstanceState*, um objeto *Bundle* que contém o estado anteriormente salvo da atividade.

- onStart() - É acionado quando a atividade insere o estado "Iniciado", tornando a atividade visível ao usuário.

- onResume() - É acionado quando a atividade insere o estado "Retomado", em que o aplicativo fica em primeiro plano e o usuário pode interagir. A atividade permanece nesse estado até ocorrer uma interrupção (alternar entre aplicativos, ser interrompido por

Figura 1 – Ilustração simplificada do ciclo de vida da atividade



Fonte: (ANDROID, 2022a).

outro, etc), que a coloca no estado "Pausado"e, posteriormente, no estado "Retomado", caso a aplicação volte novamente para primeiro plano.

- onPause() - É acionado quando a atividade insere o estado "Pausado". Esse *callback* é chamado quando existe a primeira indicação de que o usuário está deixando a atividade. Normalmente é usado para liberar de forma rápida recursos que o usuário não está precisando como, por exemplo, uma câmera ou um sensor. No entanto, deve ser evitado implementar operações demoradas como, por exemplo, salvamento de dados, pois podem não ser concluídas antes da finalização do método.

- onStop() - É acionado quando a atividade insere o estado "Parado"e é o *callback* mais adequado para implementar a lógica para salvamento de dados ou liberar recursos desnecessários. A partir do estado "Parado", a atividade volta a interagir com o usuário ou

é encerrada. Se a atividade voltar para o primeiro plano, o sistema invocará onRestart(), caso contrário, onDestroy().

- onDestroy() - É acionado antes da atividade ser destruída. Ou seja, quando a atividade é finalizada pelo usuário e descartada completamente, ou quando o sistema está destruindo temporariamente a atividade devido a uma mudança na configuração como, por exemplo, mudança do *layout* da aplicação devido a mudança de orientação do dispositivo. Ou seja, durante a mudança de orientação a atividade é temporariamente destruída e depois reiniciada para acomodar os seus elementos na nova orientação do dispositivo.

- onRestart() - É acionado depois do *callback* onStop() quando a atividade saiu do foco e agora está sendo reexibida para o usuário. É seguido pelo *callback* onStart().

A probabilidade do Android eliminar o processo da aplicação depende do estado do processo e da atividade. A Tabela 1 apresenta a correlação entre o estado do processo, o estado da atividade e a probabilidade do sistema eliminar o processo, disponibilizado na documentação do Android.

Tabela 1 – Tabela com a probabilidade de eliminação das atividades Android.

Estado do processo	**Estado da atividade**	**Probabilidade de eliminação**
Em primeiro plano (com foco ou prestes a ter)	Criado Iniciado Retomado	Mínimo
Segundo plano (perde o foco)	Pausada	Médio
Segundo plano (não visível)	Parado	Máximo
Vazio	Destruído	Máximo

Fonte: Android *developer* (ANDROID, 2022a).

A mudança de configuração ativada pelo evento *stop-start* causada por uma mudança de orientação tem uma probabilidade maior de identificar um problema de perda de dados, pois a atividade é destruída pelo *callback* onDestroy e depois recriada sobre a nova orientação do dispositivo. Caso o desenvolvedor não tenha implementado os métodos para salvar o estado das variáveis, ou não tenha implementado de forma correta os componentes cientes do ciclo de vida, eles serão reiniciados com o valor padrão e ocorrerá a perda de dados.

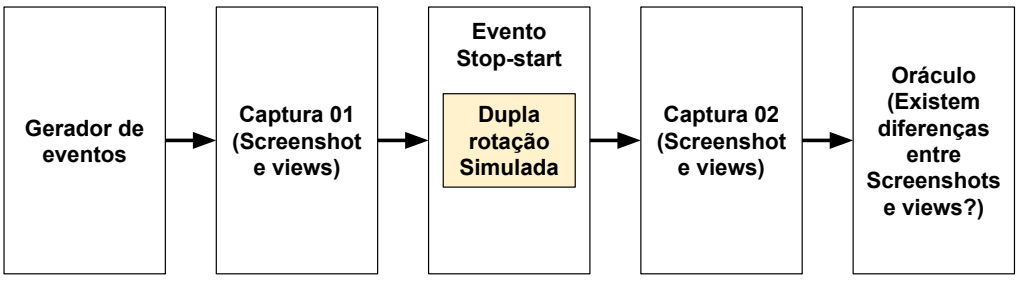
2.3 DATA LOSS DETECTOR

O Data Loss Detector (DLD) (RIGANELLI et al., 2020) é uma ferramenta para detectar perda de dados em aplicativos Android baseado no DroidBot (LI et al., 2017). Ele identifica problemas de perda de dados por meio do evento *stop-start* do tipo mudança de configuração, que é ativado por mudança de orientação do smartphone.

O DroidBot é um gerador de entradas de teste guiado por interface do usuário para aplicativos Android. Como a geração de entradas é guiada pela interface do usuário, não é necessário instrumentar o aplicativo ou ter acesso ao código fonte. O DroidBot pode ser utilizado para testar aplicativos no *Android Virtual Device* (AVD) ou em qualquer modelo de smartphone conectado via *Android Debug Bridge* (ADB). O modelo de transição de estado é gerado dinamicamente em tempo de execução e utilizado para criar entradas na interface. O Droidbot utiliza a estratégia de busca em profundidade, porém permite personalizar a exploração por meio de *scripts* ou dos módulos de geração de eventos. Atualmente o Droidbot possui um módulo, que utiliza técnicas de aprendizado profundo, para simular comportamentos mais humanos.

O DLD utiliza estratégias exploratórias e ações que maximizam a probabilidade de encontrar defeitos de perda de dados. O oráculo utiliza duas abordagens para identificar a perda de dados por meio da dupla rotação que força a destruição da atividade com a mudança da orientação. Esse processo ocorre em cinco etapas (Figura 2).

Figura 2 – Fluxo do DLD para detecção de perda de dados

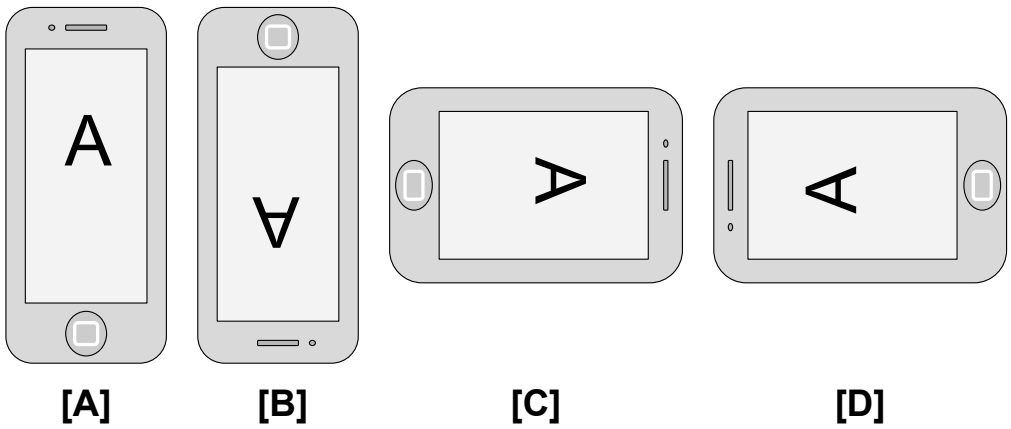


Fonte: Autor, 2022.

A primeira etapa gera um conjunto de eventos com a finalidade de modificar os valores padrão dos elementos da atividade como, por exemplo, a mudança, via digitação, de um campo que tinha valor padrão zero para o valor três. Depois, ocorre uma captura de estado da *activity* por meio de um *screenshot* e *views* (propriedades dos elementos da tela). Em

seguida, é realizada uma dupla rotação (Figura 3), por meio de comandos ADB, que move o smartphone da posição retrato (posição A), coloca a tela do smartphone na orientação paisagem esquerda (posição C) e depois retorna para retrato (posição A). Em seguida, é realizada uma nova captura do estado, mas dessa vez após o evento *stop-start* que força a destruição e a recriação da *activity*. Por último, o oráculo verifica se houve perda de dados comparando tanto os *screenshots* quanto as *views*. Caso existam divergências entre um desses artefatos, será gerado um alerta de perda de dados com as informações coletadas.

Figura 3 – Orientação do smartphone: A retrato, B retrato invertido, C paisagem esquerda e D paisagem direita.



Fonte: Autor, 2022.

A execução da ferramenta é realizada de forma automática após a definição de alguns parâmetros: o aplicativo em teste, o local para saída dos artefatos, a quantidade de eventos e o dispositivo em teste. Esse último pode ser um *Android Virtual Device* (AVD) ou um dispositivo real, e nos dois casos a rotação será realizada por meio de comandos *Android Debug Bridge* (ADB). O tempo de execução depende da quantidade de eventos passada como parâmetro e do dispositivo utilizado (AVD ou dispositivo real). Por exemplo, o experimento reportado no artigo do DLD (RIGANELLI et al., 2020) utilizou 2.250 eventos com um AVD e teve um tempo médio de 3 horas de execução.

A estratégia de exploração do DLD utiliza três métodos para aumentar a probabilidade de encontrar um problema de perda de dados:

1. Uma estratégia de exploração baseada em modelo tendencioso, que constrói um modelo da interface gráfica para representar os estados visitados e as ações executadas. A exploração visita os novos estados e testa incrementalmente os recém-descobertos. O DLD gera cinco tipos de ações durante a exploração:

a) *TouchEvent*, que executa um toque em uma visualização clicável.

b) *LongTouchEvent*, que executa um toque longo em uma visualização clicável.

c) *SetTextEvent*, que escreve texto dentro de uma visualização editável.

d) *KeyEvent*, que pressiona um botão de navegação.

e) *ScrollEvent*, que executa um *swipe* em uma exibição que permite *scroll*.

2. Ações de revelação de perda de dados, divididas em dois tipos: uma é executada sistematicamente toda vez que um novo estado abstrato é alcançado e a outra é executada probabilisticamente em cada estado abstrato. Quando um novo estado abstrato é descoberto, são realizadas as ações sistemáticas de revelação de perda de dados. Caso contrário, é feita uma ação probabilística de revelação de perda de dados que prioriza os eventos não executados na *activity*. As ações sistemáticas para revelar perda de dados possuem cinco passos:

   a) *Fill-in*: interage com todos os elementos da *activity* para inserir valores não-vazios diferentes dos valores padrão.

   b) *Save state*: salva o estado atual da *activity* para checar a posteriori se houve perda de dados.

   c) *Double screen rotation*: realiza um evento *stop-start* de mudança de orientação para forçar a recriação da *activity*. São feitas duas rotações para retornar à orientação inicial. A tela após a dupla rotação deve ser exatamente a mesma da tela inicial. Caso contrário, considera-se que ocorreu perda de dados.

   d) *Check state*: compara o estado atual com o estado salvo para determinar se ocorreu alguma perda de dados.

   e) *Scroll down*: executa uma ação de rolagem que pode fazer com que novos elementos apareçam para alcançar novos estados abstratos.

3. Uso de dois oráculos para detectar perda de dados. O DLD utiliza a estratégia de coletar o estado da GUI antes e depois de executar ações que podem identificar falhas de perda de dados e depois compará-los. Ele define duas estratégias de oráculo que podem ser usadas de forma independente ou conjunta: o oráculo baseado em *screenshot* e o oráculo baseado em propriedades.

a) Oráculo baseado em *screenshot*: O DLD faz uma captura da tela e corta o cabeçalho e o rodapé da imagem para eliminar as informações que mudam ao longo do tempo, como a hora e o nível da bateria. A comparação das imagens que representam os estados antes e após a dupla rotação são realizadas pixel por pixel. Para diminuir as ocorrências de falsos positivos relacionados ao piscar de um cursor, por exemplo, são desconsideradas diferenças de até 15 pixels para cada 10.000.

b) Oráculo baseado em propriedades: o DLD recupera todas as *views*, incluindo suas propriedades e organização hierárquica em um dicionário Python. A comparação é realizada entre os dicionários capturados antes e após a dupla rotação.

2.4 CONSIDERAÇÕES FINAIS

Neste capítulo apresentamos os principais conceitos sobre os *callbacks* do ciclo de vida de uma *activity* em aplicações Androids. Além disso, apresentamos a ferramenta DLD com suas características e estratégias de detecção de perda de dados. Com base nessas informações podemos compreender quando ocorre a perda de dados e como utilizar o evento *stop-start* do tipo mudança de orientação para identificar essas falhas. O próximo capítulo apresenta a infraestrutura $\mathrm{R\text{-}DLD}$, proposta como parte deste trabalho.

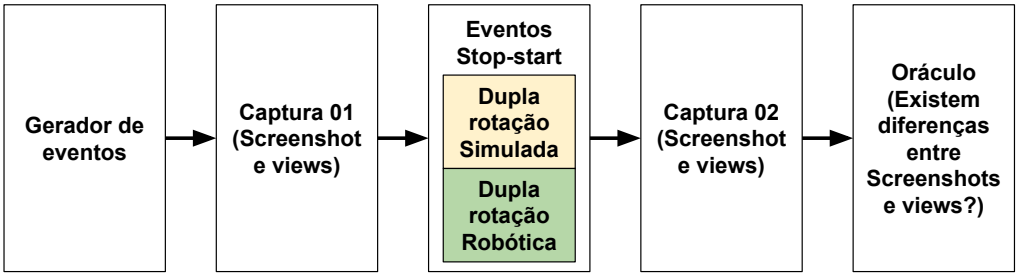
3 A INFRAESTRUTURA R-DLD

Neste capítulo descrevemos a infraestrutura R-DLD e o seu processo de detecção de perda de dados utilizando o robô. Além disso, apresentamos os detalhes da construção do robô, as modificações realizadas no DLD para suportar a estrutura robótica e uma visão ampla da infraestrutura R-DLD.

3.1 VISÃO GERAL SOBRE A ABORDAGEM

Embora o DLD permita trabalhar com dispositivos reais, a dupla rotação é realizada de forma simulada por comandos ADB, ao invés de utilizar o sensor de rotação do smartphone. Apesar de essa particularidade não impedir que a ferramenta encontre muitos tipos de perda de dados, falhas associadas aos sensores reais do dispositivo ou a interações mais complexas poderiam ser perdidas. Este trabalho propõe o R-DLD, uma extensão do DLD que permite expandir o escopo dos testes de perda de dados para incluir rotação real realizada por um braço robótico. A Figura 4 apresenta uma nova infraestrutura que permite realizar a detecção de perda de dados, tanto em ambiente simulado como no ambiente utilizando um dispositivo real.

Figura 4 – Fluxo do R-DLD: expansão do DLD para ambiente robótico.



Fonte: Autor, 2022.

3.2 DETALHES DA IMPLEMENTAÇÃO

O objetivo do R-DLD é estender a funcionalidade do DLD para permitir realizar rotações reais auxiliadas por um robô. Para isso, foi desenvolvida uma estrutura robótica, de baixo custo,

capaz de realizar mudanças de orientações em diferentes tipos de smartphones e integrada ao aplicativo DLD (Figura 5).

Figura 5 – Estrutura robótica para a detecção de perda de dados.

Fonte: Autor, 2022.

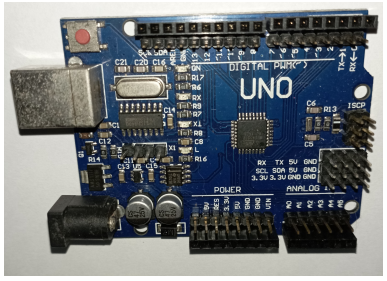
Artefatos utilizados para a construção do robô

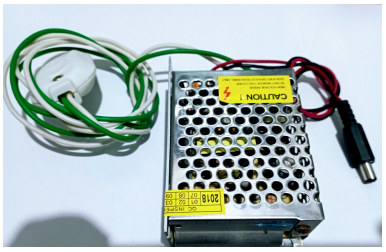
O robô foi construído na plataforma Arduino (MCROBERTS, 2015), com o objetivo de suportar quatro tipos de orientações do smartphone (Retrato, Retrato Invertido, Paisagem Esquerda e Paisagem Direita) acionados por meio de comunicação serial. Foram utilizados diferentes componentes da plataforma: uma parte composta de módulos pré-fabricados (*shields*) e uma outra desenvolvida no projeto para integrá-los.

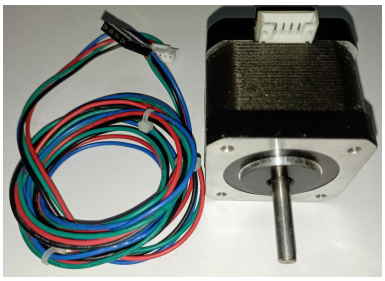
Para montar o robô, foram comprados: uma placa Arduino UNO, uma fonte 12 volts de 3 Watts, um motor de passos NEMA modelo 17HS15 e um *drive* para controlar o motor modelo DRV8825 (Figura 6). Alguns componentes tiveram que ser fabricados, como a garra para prender o smartphone e o chassis do robô (figuras 7 e 8).

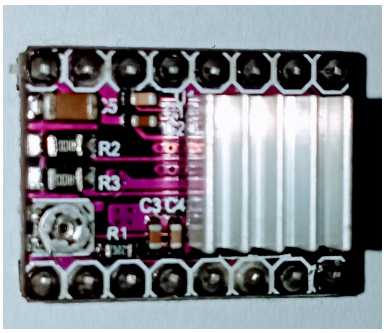
A garra foi confeccionada utilizando um suporte veicular de smartphone e adaptado um eixo para conectar ao motor de passos. Primeiro, eliminou-se o suporte que fixa o acessório ao volante do carro. Depois, abriu-se o suporte ao máximo para fazer a marcação de centro do furo do parafuso, pois o centro da peça não é o mesmo quando está conectado ao smartphone. Para isso, colocou-se um objeto para impedir o fechamento e realizou-se o furo com uma broca do diâmetro do parafuso (8mm). É importante estar com o suporte aberto para a broca

Figura 6 – [A] Arduino Uno, [B] Fonte 12 volts 3 Watts, [C] Motor de passo Nema17, [D] Driver DRV8825.

[A] 

[B] 

[C] 

[D] 

Fonte: Autor, 2022.

atravessar as duas partes da peça. Alguns ajustes foram feitos com o estilete na parte da frente do suporte para passar o parafuso e um parafuso M8x35mm foi utilizado para servir de eixo e fixá-lo no furo com uma porca. Esses passos podem ser vistos na Figura 7.

O chassi foi confeccionado com madeira e estrutura metálica para suportar os componentes e ter estabilidade durante a rotação. A Figura 8 apresenta um projeto em 3D desse componente desenhado no SketchUp. Os componentes foram interligados utilizando a documentação Arduino, mas com algumas modificações para adicionar a funcionalidade de calibração, por meio de um botão, e um ponto de conexão para um ventilador de arrefecimento do *driver*. Os pinos de controle do *driver* podem ser visto na Figura 9 e o diagrama de ligação, na Figura 10. O *driver* é uma interface para o motor de passos que permite definir, por meio de programação, o sentido de rotação, a velocidade, e a quantidade de passos para realizar um movimento.

Para posicionar o smartphone na orientação desejada durante o teste de perda de dados, foi implementado um programa em C (MONK, 2017) para se comunicar com o computador por meio de interface serial. Foi utilizado o ambiente de desenvolvimento do Arduino com a biblioteca AccelStepper[1] nessa aplicação. Como o motor de passos não recebe parâmetros com ângulos, calculou-se a quantidade de passos para cada posição da orientação. O motor utilizado necessita de 200 passos para uma volta completa, ou seja, cada passo corresponde a 1,8 grau. Assim, para atingir as orientações Retrato, Paisagem Esquerda, Retrato Invertido e Paisagem Direita (Figura 3), são necessários 0, 50, 100 e 150 passos respectivamente. A

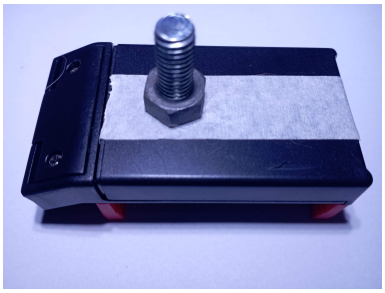
[1] <https://www.arduino.cc/reference/en/libraries/accelstepper/>

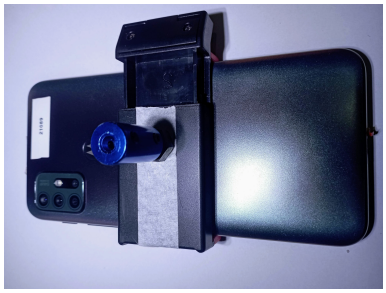
Figura 7 – Passos para construção da garra usando o suporte veicular para smartphone.

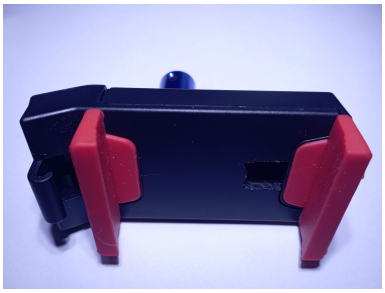
[1] 

[2] 

[3] 

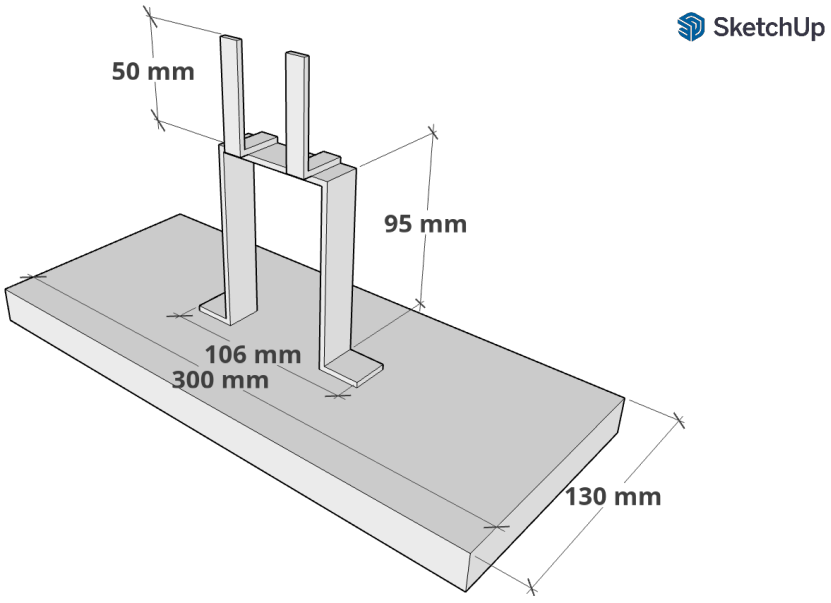
[4] 

[5] 

[6] 

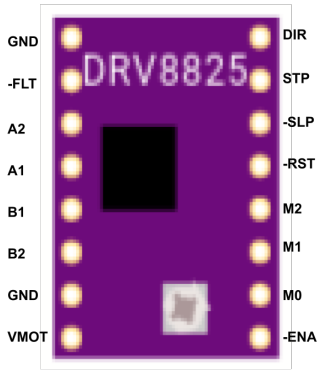
Fonte: Autor, 2022.

Figura 8 – Chassis do robô de rotação desenhado na ferramenta Sketchup.



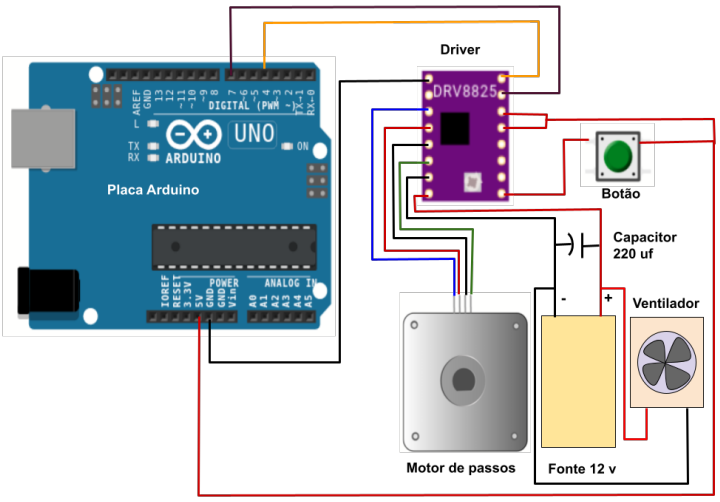
Fonte: Autor, 2022.

Figura 9 – Pinout driver DRV8825.



Fonte: Autor, 2022.

Figura 10 – Esquema elétrico do circuito de controle do robô.



Fonte: Autor, 2022.

posição inicial foi definida como Retrato e pode ser calibrada manualmente enquanto aperta-se o botão na placa.

Por ser um projeto de baixo custo, foi utilizado a calibração manual para inicializar o braço robótico. Essa atividade consiste em posicionar o smartphone, conectado a garra, na orientação retrato e apertar o botão de calibração para indicar a posição zero do motor de passo. Após essa calibração, é definido que o smartphone está na posição retrato e o motor no passo zero, assim, as outras posições serão incrementadas a partir desse ponto. Esse procedimento deve ser realizado todas as vezes que for desligado o braço robótico.

Para carregar a aplicação no Arduino UNO deve-se instalar a biblioteca AccelStepper por meio do gerenciador de bibliotecas e selecionar a placa no gerenciador de placas da IDE

Arduino. O Código 1 apresenta as definições das variáveis, no qual é incluído a biblioteca AccelStepper (linha 1) e configuradas as constantes de velocidade e aceleração do motor de passos (linhas 2–7). As configurações iniciais são apresentadas no Código 2, onde nele são configuradas a velocidade de conexão da porta serial (linha 10) e os parâmetros da biblioteca AccelStepper (linhas 13–14). Por último, o Código 3 apresenta a lógica para posicionar o motor de passo na orientação solicitada.

Código Fonte 1 – Definição das variáveis.

```
#include <AccelStepper.h>
const int VELOCIDADE_MOTOR = 400;
const int ACELERACAO_MOTOR = 200;
const int PINO_ENABLE = 10; // Definicao pino ENABLE

int input = 0;
AccelStepper motor1(1,7,4 ); // Definicao pinos STEP e DIR
```

Fonte: Autor, 2022.

Código Fonte 2 – Configuração inicial.

```
void setup() {
  Serial.begin(9600);
  pinMode(PINO_ENABLE, OUTPUT);
  // Configuracoes iniciais motor de passo
  motor1.setMaxSpeed(VELOCIDADE_MOTOR);
  motor1.setSpeed(VELOCIDADE_MOTOR);
  motor1.setAcceleration(ACELERACAO_MOTOR);
  print_msg();

}
```

Fonte: Autor, 2022.

Código Fonte 3 – Posição do motor de passos pela orientação do smartphone.

```
    if (input == '0')
    {
      motor1.moveTo(0);     //orientacao retrato
    }
    if (input == '1')
    {
      motor1.moveTo(50);    //orientacao paisagem esquerda
    }

    if (input == '2')
    {
      motor1.moveTo(100);   //orientacao retrato invertido
    }

    if (input == '3')
    {
      motor1.moveTo(150);   //orientacao paisagem direita
    }
```

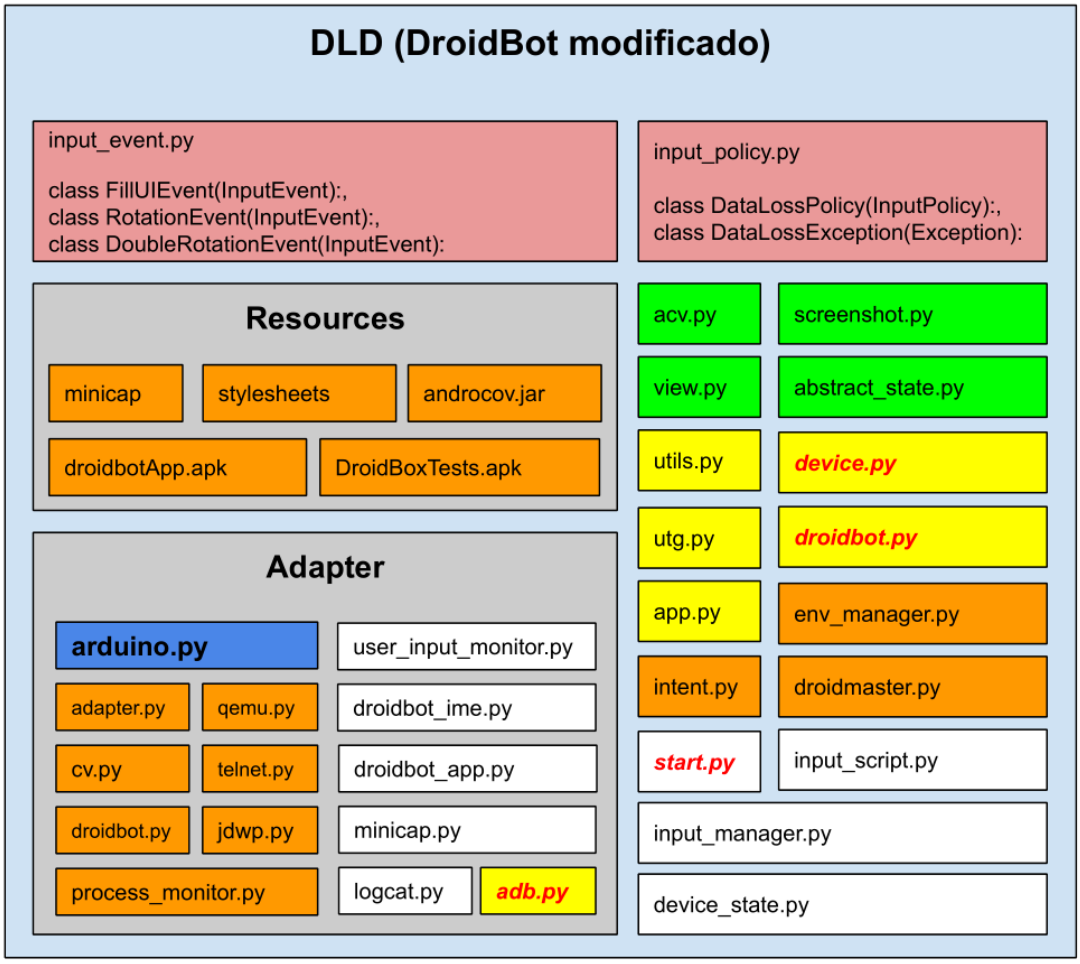
Fonte: Autor, 2022.

A aplicação responsável pelo controle de motor de passo possui aproximadamente 70 linhas de código e pode ser acessada na íntegra no repositório de replicação [2].

3.3 ARQUITETURA DO R-DLD E ALTERAÇÕES NO DLD

O R-DLD é uma extensão do DLD que, por sua vez, é composto pelo Droidbot e um conjunto de *scritps* e classes que lhe adiciona a capacidade de detectar perda de dados. A Figura 11 apresenta a arquitetura do R-DLD destacando as partes originais do Droidbot, as adicionadas e alteradas pela equipe do DLD e as criadas pelo R-DLD.

Figura 11 – Estrutura do R-DLD.



Fonte: Autor, 2022.

Partindo do princípio que o núcleo do DLD e do R-DLD é o Droidbot, será considerado, no que segue, para efeito de explicação, o Droidbot como projeto núcleo e apresentadas as modificações realizadas pelo DLD e depois, pelo R-DLD.

[2] <https://github.com/RoboticsDSF/R-DLD>

O Droidbot é composto pelo módulo Adapter, uma pasta Resources e um conjunto de classes e *scripts*. O módulo Adapter é responsável por fornecer uma abstração do dispositivo e do aplicativo em teste. A pasta de Resources possui arquivos auxiliares utilizados durante o teste. O conjunto de classes e *scripts* são responsáveis por controlar os eventos e as entradas durante o teste. A Figura 11 apresenta essas modificações. Os arquivos destacados na cor verde foram inseridos pelo DLD no Droidbot. Aos arquivos na cor rosa, foram adicionadas classes pelo DLD para tratar entradas e políticas relacionadas a perda de dados. Os arquivos na cor amarela foram incrementados com métodos para auxiliar na detecção de perda de dados. Os arquivos na cor branca tiveram pequenas alterações para trabalhar com as alterações criadas pelo DLD. Os arquivos na cor laranja não foram alterados, ou seja, são do Droidbot original. O arquivo destacado em azul foi adicionado pelo $\mathrm{R\text{-}DLD}$ para fazer a comunicação serial com a placa Arduino. Os arquivos com os nomes destacados em vermelho tiveram pequenas modificações pelo $\mathrm{R\text{-}DLD}$ para habilitar a funcionalidade do robô e a lógica para interceptar a mudança de orientação que seria enviada ao DLD para ser enviada ao robô.

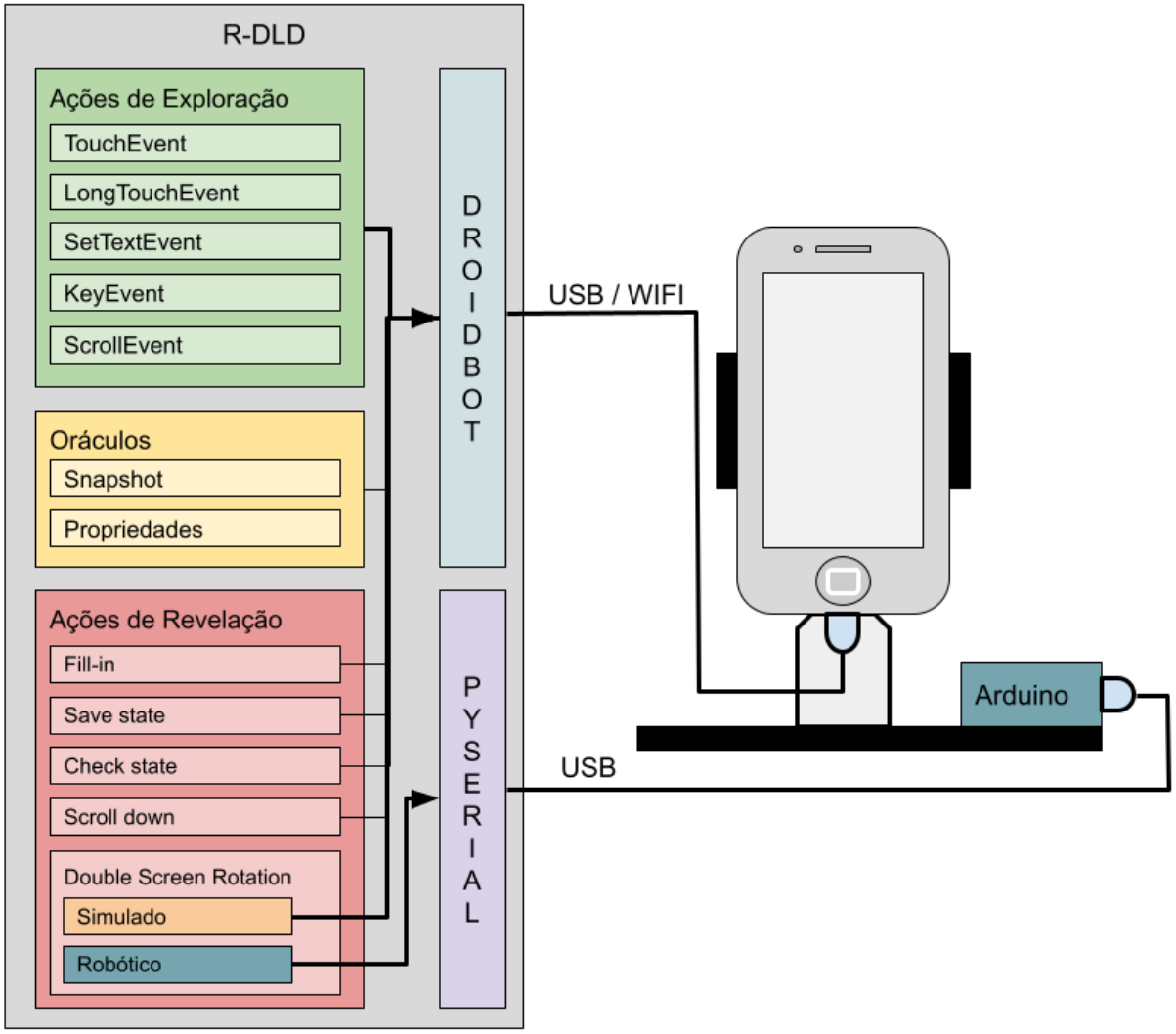
Foi criado o parâmetro *-robot*, para habilitar o robô no $\mathrm{R\text{-}DLD}$. Assim, quando esse parâmetro é utilizado, a classe ADB, que é responsável pela interação com smartphone, desvia o fluxo para a classe Arduino tratar as requisições de rotação. Por meio de comunicação serial, o Arduino controla o posicionamento do smartphone na orientação solicitada durante o teste. Como o DLD não possui suporte à comunicação serial, foi adicionada, ao módulo de instalação, a biblioteca PySerial[3] para se comunicar com o Arduino.

3.4 VISÃO AMPLA DA ARQUITETURA DO $\mathrm{R\text{-}DLD}$

O $\mathrm{R\text{-}DLD}$ estende a estrutura do DLD para suportar a infraestrutura robótica. A Figura 12 apresenta uma visão ampla da infraestrutura. Nela é possível observar que as ações de exploração e os oráculos são providos exclusivamente pelo DroidBot. No que diz respeito às ações de revelação, apenas a *Double Screen Rotation* possui um comportamento híbrido, em que uma parte é provida pelo DroidBot(*Double Screen Rotation* simulado) e outra pelo Arduino (Robótico) com auxílio da biblioteca PySerial. As outras ações de revelação são providas pelo DroidBot. Como o DroidBot utiliza o ADB para se comunicar com o smartphone é possível enviar todos os comandos por USB ou Wifi. Os detalhes da estratégia de exploração do DLD foram apresentados na Seção 2.3.

[3] <https://pypi.org/project/pyserial/>

Figura 12 – Visão ampla da arquitetura do R-DLD.



Fonte: Autor, 2022.

3.5 CONSIDERAÇÕES FINAIS

Neste capítulo apresentamos a infraestrutura R-DLD e discutimos as modificações na arquitetura e no código do DLD para suportar o robô. Além disso, apresentamos os detalhes da construção do robô e uma visão ampla da estrutura do R-DLD interagindo com o smartphone por meio do DroidBot.

4 CLASSIFICAÇÃO DOS *BUGS* REAIS E ALARMES FALSOS

Na avaliação do DLD (RIGANELLI et al., 2020) os alertas gerados pela ferramenta são classificados como *Bugs reais* ou Alarmes falsos. Essa forma de classificação, embora objetiva, não agrupa as falhas de acordo com as suas características, o que dificulta a criação de processos automatizados de classificação. Além disso, tanto os *Bugs reais* quanto os Alarmes falsos possuem diferentes características e soluções. Por exemplo, em uma perda de fragmento, há o desaparecimento de um componente, o que produz muitas alterações nos *screenshots* e nas *views*, e a solução para este problema seria implementar as ações cientes de ciclo de vida para o próprio componente. Já uma perda de texto acarreta pouco impacto visual nas *screenshots*, causa alteração em apenas uma propriedade e tem como solução implementar os métodos onSaveInstanceState() e onRestoreInstanceState() para salvar o estado da variável.

Por isso, para auxiliar as soluções de perda de dados, propomos uma classificação considerando os motivos da perda de dados. Com isso, no futuro, podemos automatizar, mesmo que parcialmente, o processo de correção das falhas de perda de dados.

4.1 CLASSIFICAÇÃO DOS *BUGS* REAIS

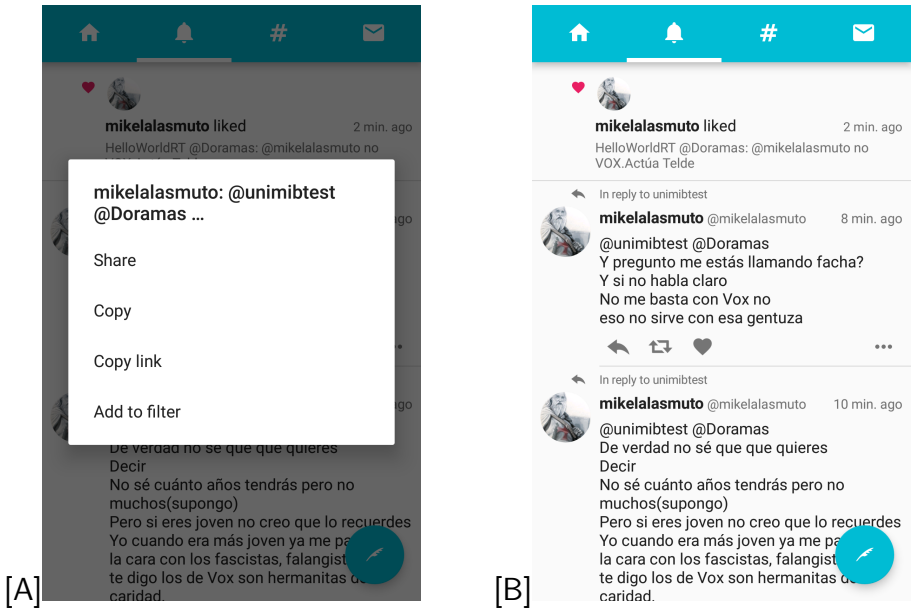
Perda de Fragmentos. Esse *bug* se manifesta quando uma seção modular de uma *activity* desaparece após a mudança de orientação. Quando esse tipo de perda de dados ocorre, gera-se uma exceção em ambos os oráculos. A Figura 13 apresenta um exemplo de perda de fragmento após a dupla rotação.

Perda de Propriedades. Esse *bug* se manifesta quando existe uma diferença entre as propriedades capturadas antes e após o evento de dupla rotação. A Figura 14 apresenta quatro capturas geradas por um *script* de análise do relatório DLD, em que a propriedade *focused* alterou de *False* para *True* após a dupla rotação.

Menu *popup* do aplicativo. Esse *bug* se manifesta quando o menu *popup* relacionado às funcionalidades do aplicativo desaparece. Alguns desenvolvedores criam menus *popups* customizados para adicionar atalhos a funcionalidades do aplicativo. A Figura 15 apresenta a perda de dados em que a funcionalidade desaparece após a dupla rotação.

Menu *popup* do Android. Esse *bug* se manifesta quando o menu *popup* relacionado às funcionalidades do sistema operacional Android (por exemplo, Copiar, Colar, Selecionar Tudo)

Figura 13 – Exemplo de perda de fragmento antes ([A]) e depois ([B]) da dupla rotação



Fonte: *Data loss detector* (RIGANELLI et al., 2020).

Figura 14 – Exemplos de perda de propriedades após a dupla rotação. A figura apresenta quatro ocorrências da mudança da propriedade *focused* após a dupla rotação.

```
2021-12-15 11-48-34 ['Before-> focused:False, After-> focused:True']
2021-12-15 11-49-21 ['Before-> focused:False, After-> focused:True']
2021-12-15 12-37-48 ['Before-> focused:False, After-> focused:True']
2021-12-15 13-43-56 ['Before-> focused:False, After-> focused:True']
```

Fonte: Autor, 2022.

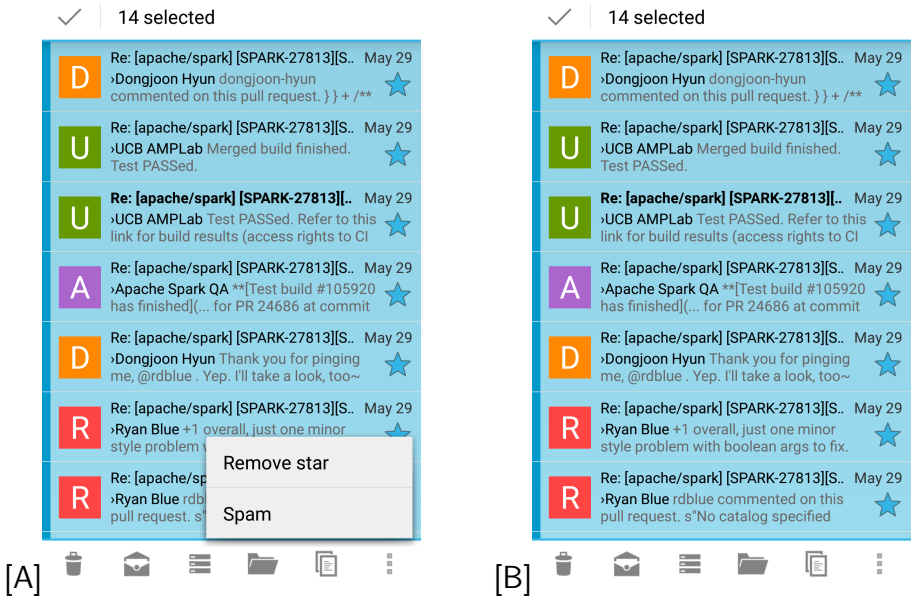
desaparece. A Figura 16 apresenta um exemplo desse *bug*.

Perda de Texto. Esse *bug* se manifesta quando algum texto digitado pelo usuário (por exemplo, durante o preenchimento de um formulário) desaparece após a dupla rotação. Quando esse tipo de perda de dados ocorre, o usuário precisa digitar todo o texto novamente. A Figura 17 apresenta um exemplo desse *bug*.

Perda de estado. Esse *bug* se manifesta quando algum elemento da *activity* volta ao estado original. Por exemplo, a *activity* volta ao estado original após um *zoom*, um *scroll*, ao expandir uma lista de elementos, entre outros. A Figura 18 contém um mapa em um nível de *zoom*, em que é possível observar informações de uma determinada coordenada. No entanto, após a dupla rotação, esse estado é reiniciado para uma opção padrão.

Barra de título. Esse *bug* se manifesta quando há alguma alteração na barra de título. Por exemplo, mudança ou desaparecimento do texto, alteração da cor, entre outros (Figura 19).

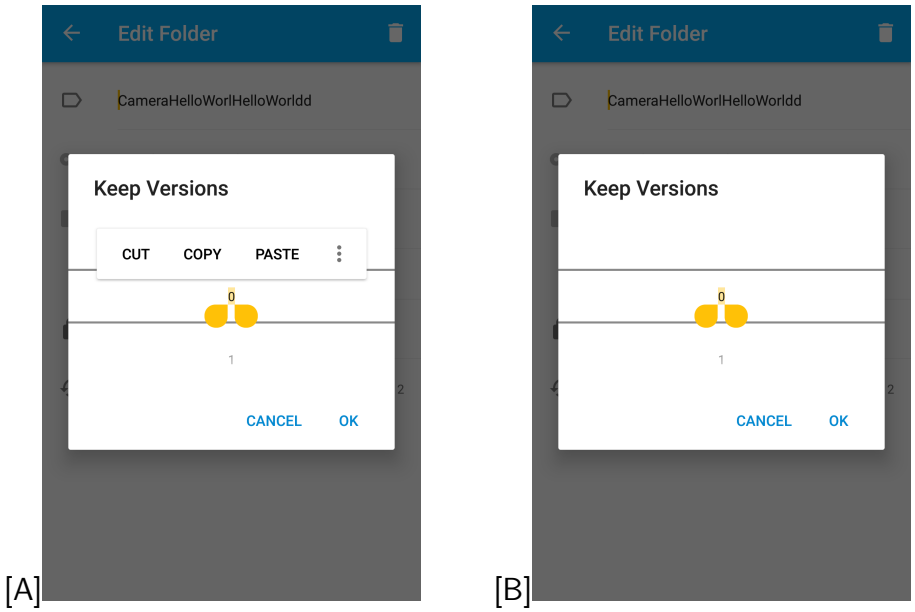
Figura 15 – Exemplo de perda do menu *popup* aplicativo antes ([A]) e depois ([B]) da dupla rotação



Fonte: *Data loss detector* (RIGANELLI et al., 2020)

.

Figura 16 – Exemplo de perda do menu *popup* Android antes ([A]) e depois ([B]) da dupla rotação

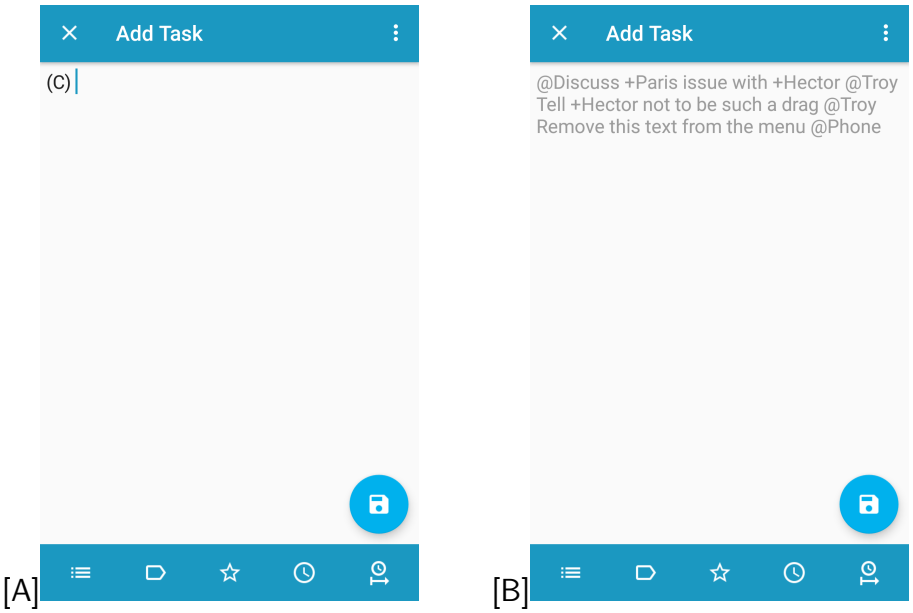


Fonte: *Data loss detector* (RIGANELLI et al., 2020).

Menu principal. Esse *bug* se manifesta quando os elementos do menu principal desaparecem ou são alterados após a dupla rotação (Figura 20).

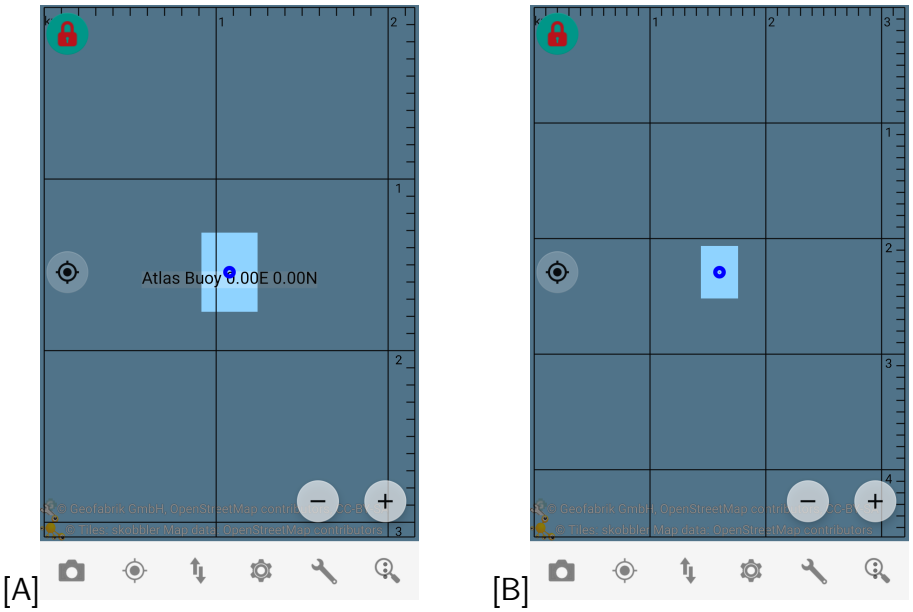
Perda de elementos da *activity*. Esse *bug* se manifesta quando algum elemento da *activity* (por exemplo, um botão, um menu) desaparece após a dupla rotação A Figura 21 apresenta um exemplo desse *bug*.

Figura 17 – Exemplo de perda de texto antes ([A]) e depois ([B]) da dupla rotação



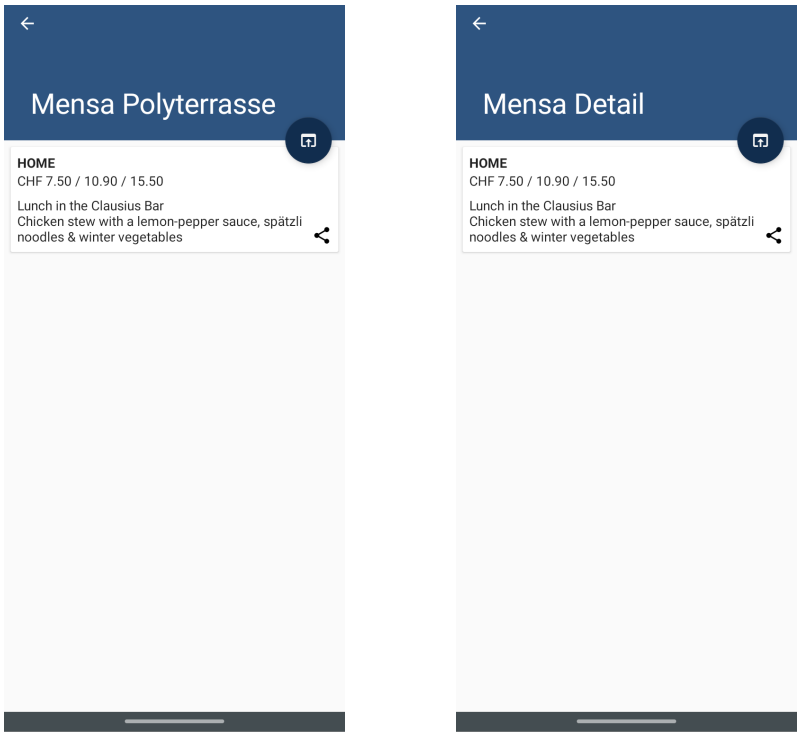
Fonte: *Data loss detector* (RIGANELLI et al., 2020).

Figura 18 – Exemplo de perda de perda de estado antes ([A]) e depois ([B]) da dupla rotação



Fonte: *Data loss detector* (RIGANELLI et al., 2020).

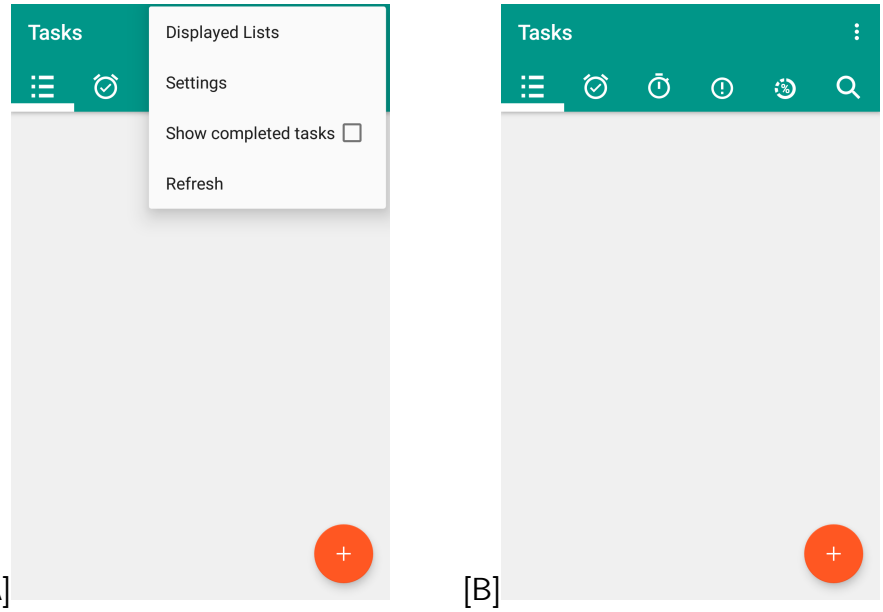
Figura 19 – Exemplo de perda de informações da Barra de título antes ([A]) e depois ([B]) da dupla rotação



[A] [B]

Fonte: Autor, 2022.

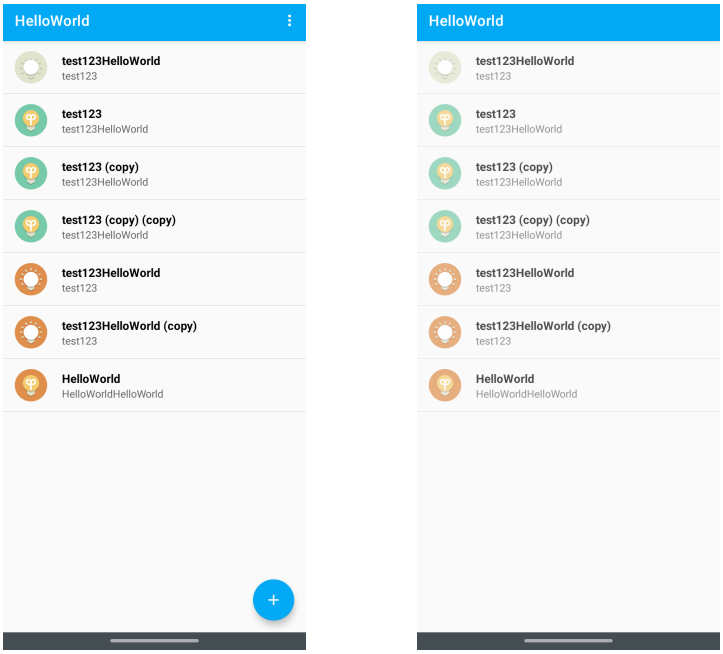
Figura 20 – Exemplo de perda do menu principal antes ([A]) e depois ([B]) da dupla rotação



[A] [B]

Fonte: *Data loss detector* (RIGANELLI et al., 2020).

Figura 21 – Exemplo de perda de elementos da *activity* antes ([A]) e depois ([B]) da dupla rotação



[A] [B]

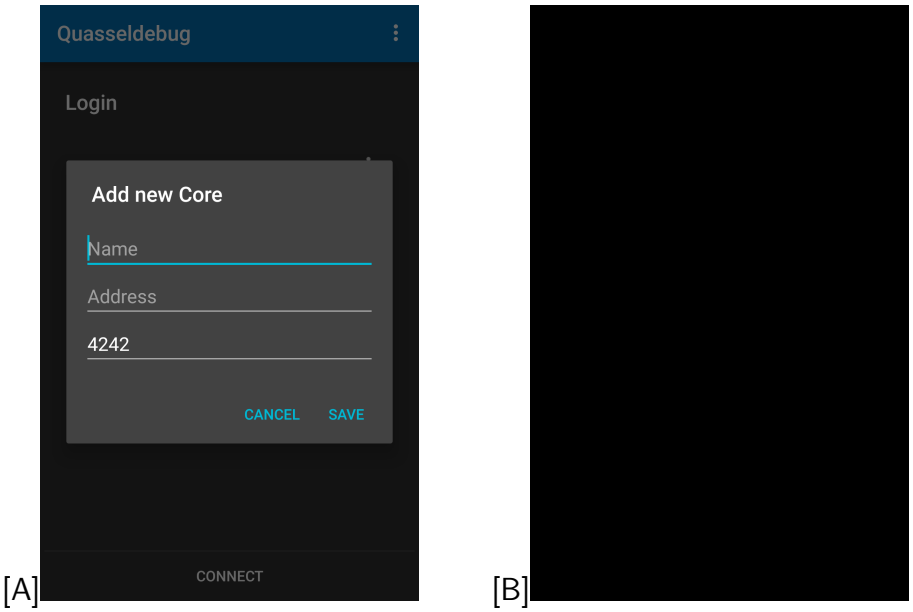
Fonte: Autor, 2022.

4.2 CLASSIFICAÇÃO DOS ALARMES FALSOS

Os Alarmes falsos são avisos de bugs reportados pela ferramenta R-DLD, mas que não representam, de fato, uma perda de dados. Tais alarmes podem acontecer por um erro na captura das telas ou por um comportamento esperado do aplicativo que é classificado, incorretamente, como uma perda de dados (por exemplo, uma animação, um vídeo ou um temporizador podem ocasionar alarmes falsos devido a diferença entre as capturas na verificação do oráculo).

Erro de captura. Esse Alarme falso se manifesta quando ocorrem erros durante a captura das telas e/ou das propriedades. É sinalizada por uma diferença entre *views*, *screenshot* ou ambos. Esse erro pode ocorrer durante a primeira ou a segunda captura. É caracterizado por uma captura vazia (no caso das *views*), ou por uma tela preta em uma das capturas (no caso das *screenshot*). A Figura 22 apresenta um exemplo desse *Alarme falso*.

Figura 22 – Exemplo de Alarme falso (Erro de captura) antes ([A]) e depois ([B]) da dupla rotação



Fonte: *Data loss detector* (RIGANELLI et al., 2020).

Cursor. Esse Alarme falso se manifesta quando são sinalizadas diferenças entre os *screenshots* gerados pelo piscar do cursor (Figura 23).

***Toast*.** Esse Alarme falso se manifesta quando são capturadas mensagens temporizadas entre as *screenshots* (Figura 24).

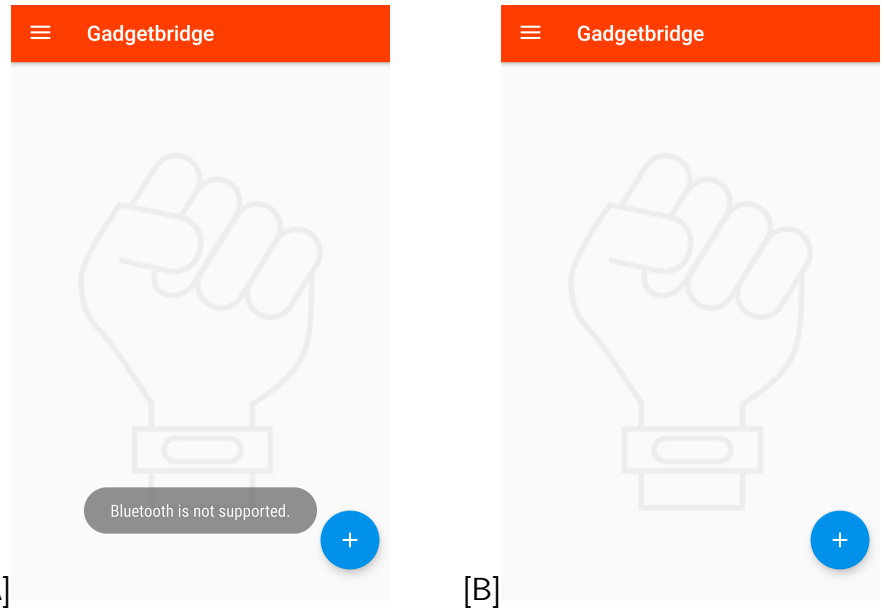
Atualização da tela. Esse Alarme falso se manifesta quando são sinalizadas diferenças entre os *screenshots* gerados por descoloração ou sinalização de carregamento de dados (Figura 25).

Figura 23 – Exemplo de Alarme falso (cursor) antes ([A]) e depois ([B]) da dupla rotação

[A] [B]

Fonte: *Data loss detector* (RIGANELLI et al., 2020).

Figura 24 – Exemplo de Alarme falso (*toast*) antes ([A]) e depois ([B]) da dupla rotação.



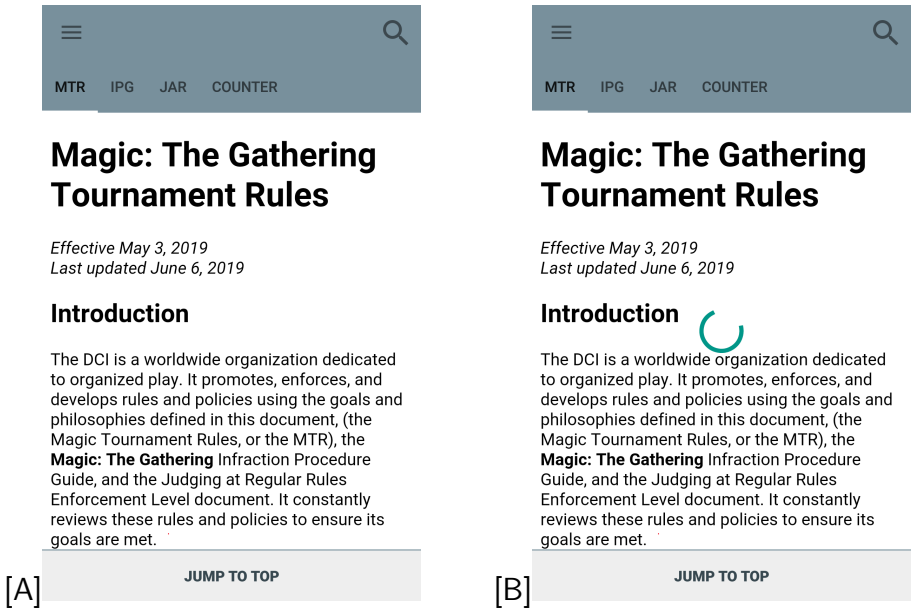
[A] [B]

Fonte: *Data loss detector* (RIGANELLI et al., 2020).

Esse Alarme falso pode ter sua ocorrência diminuída se for aumentando o tempo entre os eventos usando o parâmetro *-interval*, que tem valor padrão de três segundos.

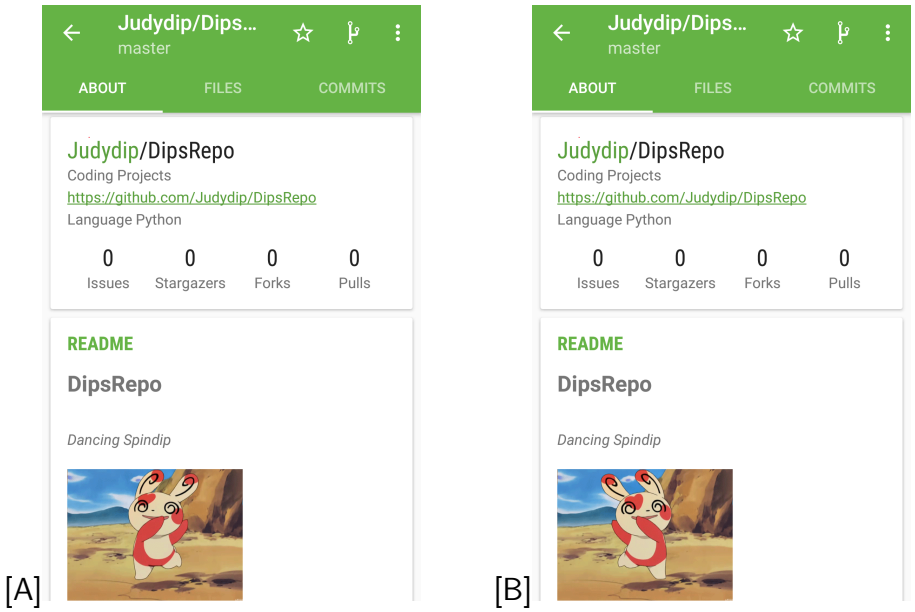
Animação. Esse Alarme falso se manifesta quando a *Activity* contém um vídeo, um GIF animado ou um elemento que se altera ao passar o tempo. A (Figura 26) apresenta uma animação na parte inferior da imagem que induz a ferramenta a marcar essa captura de tela como perda de dados.

Figura 25 – Exemplo de Alarme falso (atualização da tela) antes ([A]) e depois ([B]) da dupla rotação.



Fonte: *Data loss detector* (RIGANELLI et al., 2020).

Figura 26 – Exemplo de Alarme falso (animação) antes ([A]) e depois ([B]) da dupla rotação.

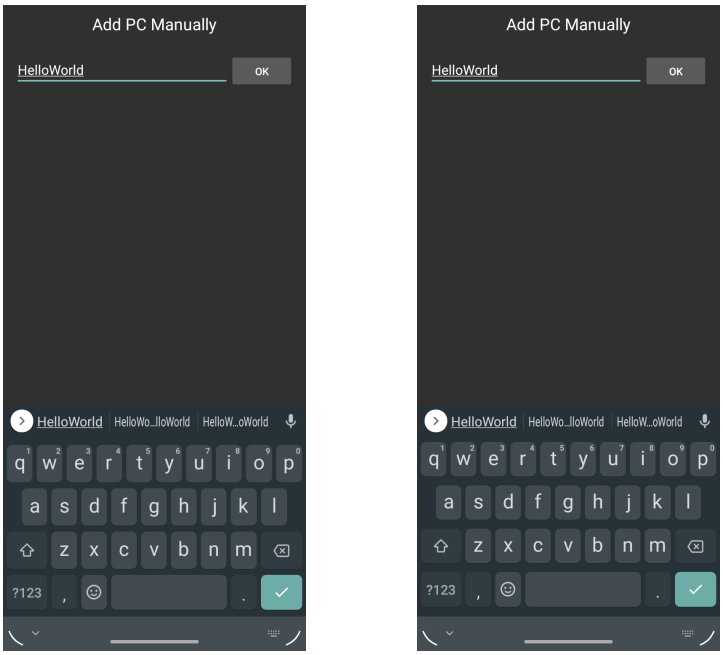


Fonte: *Data loss detector* (RIGANELLI et al., 2020).

Teclado. Esse Alarme falso se manifesta quando existe um deslocamento do teclado entre as imagens. Isso ocorre devido à animação do aparecimento do teclado que, às vezes, não é concluída antes da captura da tela. Como a inserção de texto é realizada por comandos ADBs, é uma boa prática instalar um teclado falso para evitar esse tipo de Alarme falso. A Figura 27 apresenta uma sutil diferença na altura das teclas após a dupla rotação.

Tempo. Esse Alarme falso se manifesta quando a *activity* possui algum elemento com a

Figura 27 – Exemplo de Alarme falso (teclado) antes ([A]) e depois ([B]) da dupla rotação.

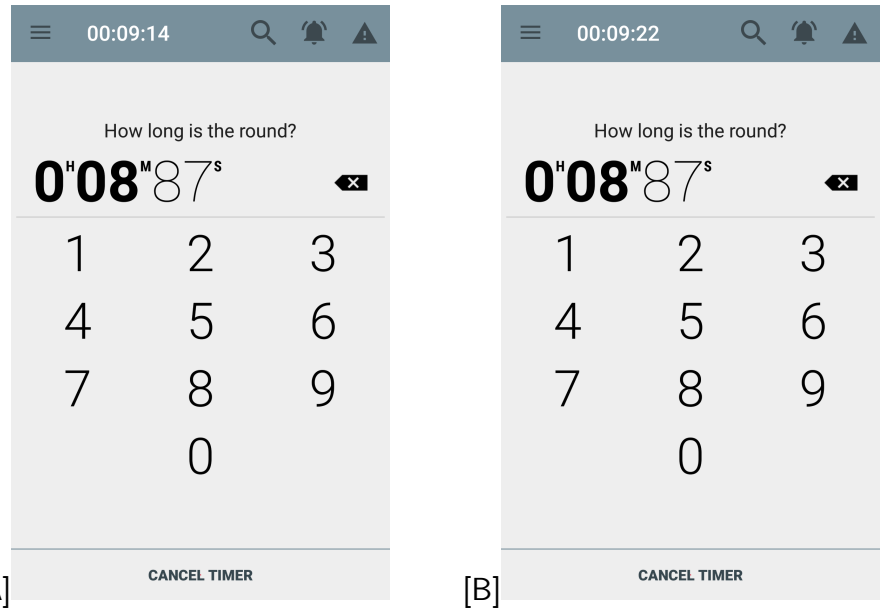


[A] [B]

Fonte: Autor, 2022.

função de medir o tempo. Alguns aplicativos realizam atividades temporizadas que aparecem na *Activity*, como o exemplo mostrado na Figura 28.

Figura 28 – Exemplo de Alarme falso (tempo) antes ([A]) e depois ([B]) da dupla rotação.



[A] [B]

Fonte: *Data loss detector* (RIGANELLI et al., 2020).

4.3 CONSIDERAÇÕES FINAIS

Nesse capítulo apresentamos uma subclassificação dos *Bugs* reais e Alarmes Falsos de acordo com as características identificadas nos alertas do $\mathrm{R\text{-}DLD}$. O objetivo é agrupar os problemas de perda de dados e facilitar a estratégia de divulgação aos desenvolvedores.

5 AVALIAÇÃO EMPÍRICA

A avaliação empírica do nosso estudo foi inspirada na avaliação do DLD (Seção 2.3). Por isso, a Seção 5.1 apresenta uma breve descrição do experimento e dos resultados obtidos pelo DLD. O processo de avaliação preliminar da nossa infraestrutura para investigar a consistência dos resultados com e sem o suporte do robô é apresentado na Seção 5.2. O processo de seleção dos aplicativos, a descrição dos passos para o experimento, a avaliação dos relatórios e a divulgação dos problemas aos desenvolvedores são apresentados na Seção 5.3. Por último, a Seção 5.4 apresenta as questões de pesquisa do nosso estudo de replicação.

5.1 AVALIAÇÃO EMPÍRICA DO DLD

Para a avaliação empírica do DLD (RIGANELLI et al., 2020) os autores comparam os estados e as telas do aplicativo antes e depois de um evento de dupla rotação para detectar perda de dados. A dupla rotação inicia o celular na orientação Retrato, gira para Paisagem Esquerda e, em seguida, retorna para a orientação Retrato. O evento de dupla rotação (ao invés de apenas uma rotação) facilita a detecção de perda de dados durante a comparação das telas: como a disposição dos elementos da GUI pode variar bastante nas orientações Retrato e Paisagem, é mais fácil comparar duas telas capturadas em uma mesma orientação.

O DLD utiliza uma métrica para escolher a *activity* e interage com os elementos da GUI por meio de ações para modificar os valores padrão. Em seguida, o DLD i) captura o estado da tela, ii) faz a dupla rotação e iii) captura o estado novamente para posterior comparação. A comparação é feita de duas maneiras: via análise de similaridade entre as telas (comparação dos *screenshot* antes e depois da dupla rotação); e via comparação das propriedades das telas antes e depois.

5.1.1 Questões de pesquisa

O estudo original investigou cinco questões de pesquisa.

- *RQ0 - Qual o valor do parâmetro ε(épsilon) que apresenta a melhor exploração?*

  A variável ε possui um range entre 0 e 1 para controlar a capacidade de explorar uma atividade. Quando $\varepsilon = 0$, implica uma exploração sistemática pura, enquanto 1 implica

uma exploração aleatória pura. Os autores do DLD realizaram um experimento com três aplicativos para definir qual é o melhor valor dessa variável.

- *RQ1 - Qual é a eficácia do DLD com problemas de perda de dados?*
  - *RQ1.1 - Qual é a capacidade de descoberta de perda de dados do DLD?*
  - *RQ1.2 - Qual é a taxa de Alarmes falsos relatadas pelo DLD?*
- *RQ2 - O DLD é mais eficaz do que as técnicas do estado da arte?*
  - *RQ2.1 - Qual é a eficácia relativa do DLD, ALARic e Quantum?*

    ALARic (RICCIO; AMALFITANO; FASOLINO, 2018) é uma ferramenta automatizada que explora o ciclo de vida da atividade para encontrar problemas de perda de dados.

    QUANTUM (ZAEEM; PRASAD; KHURSHID, 2014) é um gerador automático de casos de teste com foco em falhas de GUI.
  - *RQ2.2 - Quais são os principais fatores que determinam a eficácia do DLD?*
- *RQ3 - Qual é o trade-off entre os oráculos baseados em screenshots e propriedades?*
- *RQ4 - As falhas de perda de dados são relevantes para os desenvolvedores?*

5.1.2 Experimento

O experimento utilizou o *Data Loss Repository* como *benchmark*[1], que contém 110 falhas de perda de dados em 48 aplicativos e 54 *releases*. O DLD foi então comparado com as ferramentas ALARic e Quantum. Cada aplicativo foi submetido a três rodadas de testes com 3 horas de duração, em um dispositivo AVD Nexus 5 com Android 6.0 e 2GB de RAM, totalizando 42 dias de computação ininterrupta.

5.1.3 Resultados do experimento do DLD

A questão de pesquisa RQ0 está relacionada com a escolha do ε que promove a melhor exploração do aplicativo em teste. O DLD tem uma probabilidade ε de escolher um evento aleatoriamente e uma probabilidade $1 - \varepsilon$ de escolher um evento que nunca foi executado

[1] DataLossRepository: https://gitlab.com/learnERC/DataLossRepository

no estado atual com base no modelo GUI disponível. No experimento realizado com três aplicativos com diferentes quantidades de *activities*, o valor escolhido para esse parâmetro foi 0.1.

A efetividade do DLD em revelar problemas de perda de dados está associada à RQ1. Para responder a essa pergunta, todos os problemas reportados foram manualmente classificados para distinguir os problemas reais de perda de dados dos Alarmes falsos. Segundo os autores, uma perda de dados é considerada Alarme falso quando: (i) o estado do aplicativo após a dupla rotação é capturado muito cedo, enquanto a atividade ainda está sendo recriada, fazendo com que o oráculo falhe em sua verificação; ou (ii) a diferença relatada pelo DLD não pode ser considerada uma perda de dados.

A RQ1.1 explora a capacidade de descoberta de perda de dados. O DLD detectou 83 das 110 falhas conhecidas do *benchmark* (75%) e 35 falhas adicionais que não faziam parte do *benchmark*, mas foram reportadas aos desenvolvedores através de relatórios de bugs. Além disso, quando executado nas versões mais recentes dos aplicativos, o DLD revelou 232 novas falhas de perda de dados. Sobre as falhas presentes no *benchmark* e que não foram reveladas pelo DLD os autores citam três causas:

- Sequência de eventos com baixa probabilidade de a ferramenta executar. A ferramenta precisaria ter gerado um número maior de eventos para descobrir tal falha.
- Configuração do ambiente: seria necessária uma configuração específica para gerar a falha.
- Ações não suportadas: exigem a execução de operações que estão fora do escopo do DLD como, por exemplo, executar outro aplicativo durante o teste.

A RQ1.2 explora o índice de Alarmes falsos. O DLD relatou apenas 1 *activity* como Alarme falso para cada 4 aplicativos testados. No *benchmark* avaliado, a porcentagem varia entre um mínimo de 0% e um máximo de 24%, com um valor médio de 10,4% e mediana de 2,7%.

A RQ2 compara o DLD com ALARic e Quantum e verifica a eficácia entre as abordagens. A RQ2.2 investiga quais os principais fatores que determinam a eficácia do DLD. No primeiro quesito, o DLD superou o ALARic em termos de capacidade de descoberta de perda de dados e revelou 189 *activities* defeituosas, contra 71 *activities* reveladas por ALARic. Em relação ao Quantum, a comparação foi realizada de forma diferente, pois ele não está disponível publicamente. Assim, foi realizado um experimento por três horas com 4 dos 6 aplicativos testados

pelo Quantum e os resultados foram comparados. Os autores mencionam que é difícil realizar uma comparação justa entre as duas abordagens porque não conhecem o esforço de construir o modelo de GUI feito manualmente e exigido pelo Quantum. No entanto, os resultados sugerem que o DLD é eficaz mesmo em comparação com técnicas que usam modelos de GUI feitos manualmente. No segundo quesito, os fatores que determinaram a eficácia do DLD foram a estratégia exploratória e o oráculo baseado em *screenshot*. Os autores identificaram que o ALARic encontrou poucas falhas porque a estratégia exploratória não atingiu as *activities* com perda de dados e devido à falta de um oráculo baseado em *screenshot*.

A comparação entre os dois oráculos foi realizada na RQ3 e constatou que 73,1% das perdas de dados foram reveladas em ambos os oráculos, sugerindo que a maioria das falhas de perda de dados estão ligadas a propriedades que perdem seus valores e a problemas visíveis no aplicativo. Em 26,9% dos casos a perda de dados é capturada por apenas um dos oráculos: 9,2% dos casos são detectados apenas pelo oráculo baseado em *screenshot*; e 17,8% dos casos são detectados apenas pelo oráculo baseado em propriedades. Com relação aos Alarmes falsos, 73,6% dos casos foram reportados pelos dois oráculos, 21,1% dos casos foram reportados apenas pelo oráculo baseado em *screenshot*, 5,3% foram ocasionados por rotação incompleta e 0,1% foram reportados apenas pelo oráculo baseado em propriedades.

A relevância do problema de perda de dados para os desenvolvedores foi avaliada na RQ4 por meio de um novo experimento com três execuções de três horas e com a versão atualizada dos aplicativos do *benchmark*. Foram encontradas 195 falhas de perda de dados e reportadas aos desenvolvedores. Os autores obtiveram 98 respostas, das quais 88 confirmaram a falha reportada e apenas em 10 casos os desenvolvedores rejeitaram o relatório. Para os casos em que a falha de perda de dados não foi confirmada, os desenvolvedores dos aplicativos alegaram que a falha era uma falha de estrutura, que a falha não era reproduzível ou simplesmente ignoraram o relatório.

Uma boa prática citada no artigo do DLD (RIGANELLI et al., 2020) para evitar problemas de perda de dados, foi mover a ação dos componentes que são cientes de ciclo de vida para o próprio componente no caso de componentes, fragmentos e diálogos. Ou salvar as variáveis da aplicação antes da destruição e restaurá-las após a criação da *activity* por meio da implementação dos métodos onSaveInstanceState() e onRestoreInstanceState().

5.2 VALIDAÇÃO PRELIMINAR DO BRAÇO ROBÓTICO

Para verificar se a adição de rotações físicas suportadas pelo braço robótico e a nossa implementação do R-DLD influenciavam os resultados obtidos nos estudos realizados com rotação simulada, realizamos uma avaliação preliminar (verificação de sanidade).

Nossa avaliação preliminar tem o propósito de responder a seguinte questão de pesquisa: *A rotação real realizada pelo* R-DLD *altera os resultados dos testes de perda de dados realizados com rotações simuladas?*

Para a nossa avaliação preliminar utilizamos um *benchmark* de falhas de perda de dados (RIGANELLI et al., 2019). O *benchmark* é composto por 48 aplicativos desenvolvidos para Android, testados nas APIs 22 e 23, com perda de dados documentadas e seus respectivos casos de teste. Os casos de teste são implementados em Java e executados com o Appium (HANS, 2015) em um smartphone Moto G30. Cada caso de teste (i) realiza uma série de passos para colocar o aplicativo no estado em que foi documentada a perda de dados; (ii) realiza a dupla rotação; e (iii) confirma a perda de dados através do oráculo. Como uma perda de dados pode se manifestar em mais de uma atividade, alguns aplicativos possuem mais de um caso de teste, totalizando 110. No entanto, não existe uma padronização em identificar a perda de dados. Por exemplo, 12 testes do *benchmark* não possuem oráculo; em alguns, a perda de dados é identificada pela exceção; e, em outros, pela falha do teste. Para esse estudo, foram descartados: (i) os casos de testes que não possuem oráculos; (ii) casos de testes que identificam perda de dados através das exceções levantadas; (iii) casos de testes que possuem erros na implementação.

Como o experimento foi realizado em um dispositivo real e o *benchmark* em um dispositivo virtual (AVD), foi necessário realizar duas etapas de verificação para selecionar os aplicativos. Essas etapas de verificação prévia foram necessárias, pois alguns testes do *benchmark* utilizam coordenadas absolutas para encontrar um elemento da tela, enquanto outros usam o identificador do elemento. O uso de coordenadas pode fazer um teste falhar devido ao tamanho da tela ou resolução. Na primeira etapa foi executada a suíte de testes, sem qualquer alteração, no AVD com configuração semelhante ao dispositivo real do experimento, com a finalidade de eliminar os testes que sofrem a influência da diferença entre a versão do AVD do *benchmark* e o dispositivo do experimento. Por exemplo, um teste que realiza um toque em ponto específico da tela pode ativar outra funcionalidade se a resolução da tela for diferente devido à disposição dos elementos da *activity*. Na segunda etapa, os testes que passaram na etapa anterior foram

executados três vezes no AVD para verificar inconsistências nos resultados, ou seja, testes que geram resultados diferentes sob as mesmas condições. O ambiente foi então configurado seguindo a documentação do *benchmark* em um computador com processador AMD FX 8320E com 16 GB de memória RAM. Nesse processo, foram selecionados treze aplicativos e dezoito casos de teste para serem utilizados no experimento.

Os dois tratamentos (rotação simulada e real) foram executados no smartphone MI5X com os dezoito casos de testes selecionados. Na rotação simulada (primeiro tratamento), não houve nenhuma alteração no código do teste, apenas a execução realizada no AVD. Na rotação real realizada pelo robô (segundo tratamento), houve uma pequena alteração no código dos testes com o auxílio de um *script* que substitui as instruções de rotação simulada por outras a serem realizadas pelo robô. O caso de teste original utiliza uma pausa de um segundo após cada mudança de orientação para o smartphone realizar a ação (linhas 46 e 48 do Código Fonte 4). Essa instrução foi substituída por uma pausa de dois segundos após solicitar a mudança de orientação ao braço robótico (linha 53 do Código Fonte 5), assim, o braço robótico tem esse tempo para realizar os dois movimentos. Embora o uso da *Thread.sleep* não seja uma prática recomendada, foi utilizada para manter o padrão de tempo com o caso de teste original.

Para ilustrar as alterações realizadas no código dos testes o Código Fonte 4 apresenta um exemplo de um caso de teste original e o Código Fonte 5 apresenta o código do mesmo teste após a alteração para que a mudança de orientação seja realizada pelo robô. Quando o caso de teste adaptado para o robô é executado, um processo inicia um *script* em Python que faz a comunicação com a placa Arduino para a mudança de orientação. O robô muda a orientação de retrato para paisagem esquerda, aguarda um segundo e volta para a orientação retrato. Por fim, o caso de teste continua para verificar o oráculo.

Código Fonte 4 – Caso de teste original.

```
    @Test
38  public void TestRun() throws IOException, InterruptedException {

40      Thread.sleep(1000);
        driver.findElementByAccessibilityId("Settings").click();
42      Thread.sleep(1000);
        driver.findElementByClassName("android.widget.RelativeLayout").click();
44      Thread.sleep(1000);
        driver.rotate(ScreenOrientation.LANDSCAPE);
46      Thread.sleep(1000);
        driver.rotate(ScreenOrientation.PORTRAIT);
48      Thread.sleep(1000);

50      WebElement menu = driver.findElementByClassName("android.widget.TextView"
    );
        Assert.assertEquals("Select Theme", menu.getText());
52  }
```

Fonte: (RIGANELLI et al., 2019)

.

Código Fonte 5 – Caso de teste alterado para rodar com o robô.

```
42  @Test
    public void TestRun() throws IOException, InterruptedException {
44
        Thread.sleep(1000);
46      driver.findElementByAccessibilityId("Settings").click();
        Thread.sleep(1000);
48      driver.findElementByClassName("android.widget.RelativeLayout").click();
        Thread.sleep(1000);
50
        pro =run.exec("python2.7 /home/davi/projetos_git/Teste_smartphone/arduino
    .py");
52      read = new BufferedReader(new InputStreamReader(pro.getInputStream()));
        Thread.sleep(2000);
54

56      WebElement menu = driver.findElementByClassName("android.widget.TextView"
    );
        Assert.assertEquals("Select Theme", menu.getText());
58  }
```

Fonte: Autor, 2022.

Os dados foram coletados a partir das saídas dos *scripts* de automação, sendo consolidados em uma planilha (Tabela 2) com o código do caso de teste e o tratamento (rotação simulada ou real). Por último, foi calculada a taxa de sucesso para cada tratamento conforme a equação $TS = \frac{QtdS}{QtdT}$, onde TS representa a taxa de sucesso, $QtdS$ é a quantidade de testes que executaram com sucesso e $QtdT$ é a quantidade total de testes.

A Tabela 2 apresenta os casos de teste utilizados do *benchmark* e os resultados do experimento com os dois tratamentos. O tratamento com a rotação real suportada pelo R-DLD

apresentou os mesmos resultados alcançados pelo tratamento com a rotação simulada, indicando que a nossa implementação do R-DLD não altera os resultados observados em estudos com a rotação simulada. Diante desse resultado, prosseguimos com a realização de um estudo de replicação utilizando novos aplicativos Android (detalhado nas próximas Seções).

Tabela 2 – Resultados dos testes com rotação simulada e real.

Caso de teste	**Rotação Simulada**	**Rotação Real (Robô)**
AntennaPod2023	pass	pass
AntennaPod2028	pass	pass
CalendarNotifications200	pass	pass
CalendarNotifications202	pass	pass
Diary47	pass	pass
Diary52	pass	pass
Equate11	pass	pass
FirefoxFocus1974	pass	pass
LoopHabitTracker239	pass	pass
LoopHabitTracker239b	pass	pass
OpenTasks658	pass	pass
PortKnocker9	pass	pass
SimpleSolitare3	pass	pass
Simpletask843	pass	pass
Syncthing804	pass	pass
Taskbar20	pass	pass
WiFiAnalyzer36	pass	pass
WiFiAnalyzer78	pass	pass

Fonte: Autor, 2022.

5.3 SELEÇÃO DOS APLICATIVOS DA LOJA F-DROID

O objetivo desse experimento é explorar a ocorrência de perda de dados em um conjunto mais abrangente de aplicativos Android para reportar e propor soluções aos desenvolvedores. A loja Android escolhida foi a F-Droid, pois hospeda mais de 2.500 projetos gratuitos e de código aberto, além de disponibilizar um link para baixar a versão compilada do aplicativo. Essas atividades foram realizadas por meio de *scripts* em Python para automatizar o processo de coleta de dados e execução do experimento.

5.3.1 Web scraping

O processo de coleta de dados foi realizado utilizando técnicas de *web scraping* para obter nome, descrição, versão do Android, e links para o repositório e para a versão compilada. No primeiro momento, foram coletados todos os links, nas dezessete categorias de software, totalizando 3.631 entradas. Após eliminar, de modo manual, as entradas repetidas e os links quebrados esse número reduziu para 2.461.

5.3.2 Análise do manifesto dos aplicativos

O próximo passo foi fazer uma análise do *Manifest* dos aplicativos para verificar se existe algum bloqueio das orientações da tela. A análise utiliza uma biblioteca para ler o arquivo AndroidManifest.xml que está integrado na versão compilada do aplicativo e procura instruções específicas de configuração. Foi utilizada a biblioteca Androguard para buscar por ocorrências da instrução *screenOrientation* nas atividades do aplicativo. Pode ocorrer três tipos de bloqueios: (i) Bloqueio total da orientação do aplicativo, nesse caso o R-DLD não consegue acionar o evento *stop-start*, esses aplicativos foram descartados. (ii) Bloqueio parcial da mudança de orientação do aplicativo, nesses casos algumas atividades poderão ter um bloqueio. O R-DLD não consegue diferenciar uma atividade bloqueada de uma normal e poderia explorar o aplicativo por horas sem conseguir executar o evento *stop-start*. Esses aplicativos terão menor prioridade para serem escolhidos. (iii) Aplicativo sem bloqueio de mudanças na orientação, esses aplicativos terão maior prioridade para serem escolhidos.

Outra instrução procurada foi a configChanges, que é utilizada para passar a responsabilidade da mudança de configuração para o desenvolvedor (ANDROID, 2022b), através da implementação do método onConfigurationChanged() para tratar a mudança de configuração. Na prática, é comum o desenvolvedor não implementar esse método e utilizar somente essa instrução para impedir a destruição da atividade durante a mudança de orientação, o que pode ocasionar efeitos colaterais.

A análise do *Manifest* revelou que 30,8% dos aplicativos possuem apenas uma *activity*. Esse número aumenta para 55,6%, quando contabilizamos aplicativos com até 3 *activities*, e para 70,5% quando contabilizamos aplicativos com até 5 *activities*. Já em relação às mudanças de configurações, 35,1% dos projetos estão com o parâmetro configChange habilitado no *Manifest*, indicando que o desenvolvedor assume a responsabilidade de tratar eventos como

mudança de orientação por meio do método onConfigurationChanged. Além disso, 22,8% dos aplicativos possuem algum tipo de bloqueio de orientação. Isso pode ocorrer em todas as *activities* ou em apenas parte delas. Ao contabilizar somente os bloqueios na *activity* principal, essa porcentagem diminui para 13,8%. Como, em geral, a maior parte das funcionalidades do aplicativo estão na *activity* principal, a ocorrência desse bloqueio descarta o aplicativo automaticamente. No entanto, quando ocorre em outro local, o aplicativo poderá ser analisado em outro momento. O critério screenOrientation foi utilizado para eliminar os aplicativos que tinham bloqueios em alguma *activity*. Assim, a quantidade de projetos aptos foi reduzida para 1.899.

O próximo critério tem a finalidade de evitar projetos abandonados pelos desenvolvedores. Selecionados aplicativos que tiveram pelo menos uma atualização em 2021. Ao analisar os repositórios utilizados pelo F-Droid, observou-se que 78,4% correspondem ao Github, 11,3% ao Gitlab e 10,3% a outros repositórios. Um *script* para obter os *commits* de cada projeto foi desenvolvido para Github por maximizar a quantidade de aplicativos avaliados, enquanto os outros repositórios foram descartados. Ao aplicar essas restrições, 733 projetos foram selecionados para serem submetidos à análise do R-DLD, o que corresponde a 29,8% do número inicial de projetos.

5.3.3 Avaliação usando R-DLD

Nessa etapa, foram submetidos ao $\mathrm{R\text{-}DLD}$ os aplicativos não desclassificados na etapa anterior. O experimento teve a mesma quantidade de eventos (2250) utilizado na avaliação do DLD, porém foram realizadas com rotação real com o objetivo de descobrir defeitos reais em um conjunto maior de aplicativos.

O processo de execução foi automatizado para funcionar de forma contínua, respeitando o tempo necessário para a instalação e desinstalação no smartphone com Android. Foram escolhidos, de forma aleatória, 77 aplicativos para serem analisados no experimento ($\approx 10\%$ do número de aplicativos candidatos).

O experimento foi executado no $\mathrm{R\text{-}DLD}$ com um smartphone Motorola modelo MotoG30 (memória de 4GB, armazenamento 128GB, resolução 1600x720 e Android 11). Inicialmente foram realizadas algumas configurações no dispositivo para permitir a instalação dos aplicativos e a comunicação por USB. Também foi instalado um *Null Keyboard* para impedir que o teclado apareça no momento da entrada de texto. Essa decisão foi tomada a partir da observação de

Alarmes falsos relacionados à interferência do teclado durante as nossas análises iniciais.

Ao iniciar o experimento com os 2.250 eventos observamos que o tempo de processamento foi 50% maior (4,5 horas) que o tempo reportado na avalição do DLD (3 horas). A investigação concluiu que esse acréscimo é decorrente da diferença entre utilizar um simulador (AVD) e um dispositivo real, não estando relacionado com o uso do robô de rotação. Ou seja, caso o experimento do DLD tivesse sido executado em um smartphone real, ao invés do AVD, o tempo para processar os 2250 eventos também seria 50% maior (4,5 horas).

O experimento gerou 77 relatórios e teve uma média de 192 rotações, 82 perdas de dados, 71% de cobertura das *activities* e 18,5% de *fatal exceptions* por aplicativo. O tempo médio de cada execução foi de quatro horas e meia, totalizando aproximadamente 350 horas de computação.

5.3.4 Avaliação dos relatórios

Avaliamos manualmente as informações dos relatórios para categorizar os alarmes como sendo verdadeiro positivo (*Bug real*) ou falso positivo (*Alarme falso*). Todas as perdas de dados marcadas com *Bug real* são reportadas aos desenvolvedores por meio da abertura de *issues* com a descrição dos problemas utilizando os artefatos produzidos pela ferramenta R-DLD.

Alarmes falsos tiveram ocorrência maior nas perdas de dados reportadas por *screenshot* e estavam associadas à animação, vídeos ou a mensagens temporizadas (*toast*). Também podem ocorrer por diferenças de tonalidade do *screenshot* decorrente da demora na atualização da tela, ou por causa do fechamento do aplicativo causado por uma exceção.

Uma perda de dados é classificada como *Bug real* quando uma variável é destruída, reinicializada ou atribuída a um valor padrão. Quando reportadas por *screenshot*, a perda de dados se apresenta como um fragmento, texto ou mensagem que desaparece, um elemento que altera a cor ou perde o estado, entre outros. Essas perdas sempre estão associadas a grandes diferenças entre os arquivos das *views*.

5.3.5 Reportando issues

Após a classificação dos relatórios, prosseguimos com a etapa de reportar os *bugs*, cujo objetivo é apresentar o problema ao desenvolvedor. A abertura de *issue* é uma forma de reportar uma falha ao desenvolvedor com um relato que envolve a descrição do problema, os passos

para reproduzir, o comportamento esperado, o modelo do dispositivo e a versão do aplicativo. São utilizados os *screenshots* reportados pelo $\mathrm{R\text{-}DLD}$ para facilitar a reprodução da falha pelo desenvolvedor.

5.4 QUESTÕES DE PESQUISA DO ESTUDO DE REPLICAÇÃO

A avaliação empírica do DLD foi realizada utilizando um dispositivo virtual (AVD) e, portanto, sem interferência dos sensores do hardware, das tecnologias de conexões, da quantidade de memória e do poder de processamento do dispositivo alvo. Nosso estudo de replicação foi realizado em um contexto diferente — com o suporte de uma infraestrutura robótica e com rotações físicas em um dispositivo real — para promover testes menos invasivos e mais realistas. O estudo original possui cinco questões de pesquisas: (i) Uma questão de pesquisa para escolher a estratégia de exploração definida pela variável ε. Para efeito de replicação foi utilizado o mesmo valor dessa variável (*epsilon* $= 0.1$). (ii) Uma questão para comparar o DLD com as abordagens concorrentes (ALARic e QUANTUM). Como nosso objetivo não foi comparar, essa questão foi descartada. (iii) Uma questão sobre a eficácia do DLD, que originou a RQ1. (iv) Uma questão sobre a descoberta de *bugs* por tipos de oráculos, que originou a RQ2. (v) Uma questão sobre a relevâncias da perda de dados para os desenvolvedores, que originou a RQ3. Assim, foram definidas as seguintes questões de pesquisa:

- RQ1 - Qual a relação entre *Bugs reais* e Alarmes falsos identificados pela infraestrutura $\mathrm{R\text{-}DLD}$? Essa questão de pesquisa verifica se a proporção de *Bugs reais* e Alarmes falsos identificados pelo $\mathrm{R\text{-}DLD}$ é semelhante àquela observada no estudo de avaliação do DLD (RIGANELLI et al., 2020).

- RQ2 - A detecção de perda de dados obedece à mesma proporção identificada pelo DLD para os oráculos *screenshot-based* e *property-based*? Essa questão de pesquisa ajuda a esclarecer se os oráculos têm um comportamento semelhante ao observado no estudo de avaliação do DLD (RIGANELLI et al., 2020) quando alteramos os aplicativos e consideramos mudança de orientações reais.

- RQ3 - O problema de perda de dados é relevante para os desenvolvedores? Essa questão de pesquisa tem a finalidade de coletar informações dos desenvolvedores a respeito das

falhas de perda de dados reportadas nos aplicativos e verificar se existe interesse em corrigi-las.

5.5 REPLICAÇÃO DO EXPERIMENTO

Todos os artefatos produzidos no experimento estão no pacote de replicação disponível em <https://github.com/RoboticsDSF/R-DLD>. O repositório foi dividido em três partes: (i) Aplicação possui o código fonte do DLD com as alterações para suporte por robô. (ii) Construção do robô possui os detalhes para montar a estrutura robótica com o diagrama elétrico e o código fonte do Arduino. (iii) Avaliação possui o relatório dos aplicativos avaliados utilizando o R-DLD, planilhas com a classificação em *Bugs reais* ou *Alarmes falsos*, a planilha com a abertura das *issues* e as respostas dos desenvolvedores.

5.6 CONSIDERAÇÕES FINAIS

Nesse capítulo apresentamos os detalhes da avaliação empírica do nosso trabalho. Apresentamos uma breve descrição da avaliação empírica do DLD e dos resultados obtidos. Em seguida, apresentamos os resultados de uma avaliação preliminar para verificar se a adição de rotações físicas suportadas pelo braço robótico e a nossa implementação do R-DLD influenciavam os resultados obtidos nos estudos realizados com rotação simulada. Por último, relatamos o processo de seleção dos aplicativos, as questões de pesquisa para o nosso estudo de replicação.

6 RESULTADOS

Nesse capítulo apresentamos os resultados do estudo de replicação utilizando o R-DLD, iremos responder as questões de pesquisas reportadas no Capítulo 5 e realizar uma breve discussão sobre as características dos *Bugs reais* e Alarmes falsos reportados no experimento. Por fim, apresentaremos as ameaças a validade do estudo de replicação.

6.1 RQ1: QUAL A RELAÇÃO ENTRE *BUGS REAIS* E ALARMES FALSOS IDENTIFICADOS PELA INFRAESTRUTURA R-DLD?

Em nosso estudo de replicação analisamos 77 aplicativos utilizando o R-DLD, com o tempo médio de execução de 4h e 31 min por aplicativo, e avaliamos os 3.589 alertas de perda de dados que foram gerados. Seis desses aplicativos não reportaram nenhum alerta de *bug*, como pode ser visto na Tabela 3. No entanto, em três casos o R-DLD ficou preso na tela de tutorial e para contornar isso seria necessário uma configuração inicial, que avançasse até a atividade principal, antes de iniciar a avaliação. Seis aplicativos reportaram alertas de *bug*, mas foram classificados como Alarmes falsos Tabela 4.

Tabela 3 – Aplicativos que não apresentaram bugs

Aplicativo	**Diagnóstico**
TripleCamel	Adiciona funcionalidade em outro aplicativo. Possui uma interface simples, apenas com instruções para uso.
AF Weather	Widget
Shelter	A configuração inicial possui vários tutoriais que impossibilitaram o R-DLD de acessar a atividade principal.
Track and Graph	A configuração inicial possui um tutorial que impossibilitou o R-DLD de acessar a atividade principal.
Keymapper	A configuração inicial possui um tutorial e várias configurações que impossibilitaram o R-DLD de acessar a atividade principal.
AccA	O aplicativo precisa de acesso *root* ativado para funcionar.

Fonte: Autor, 2022.

Após uma análise manual de cada alerta, 2.160 (60,2%) alertas foram classificados como *Bugs reais*, e 1.429 (39,8%) foram classificados como Alarmes falsos. As Figuras 29 [A] e [B]

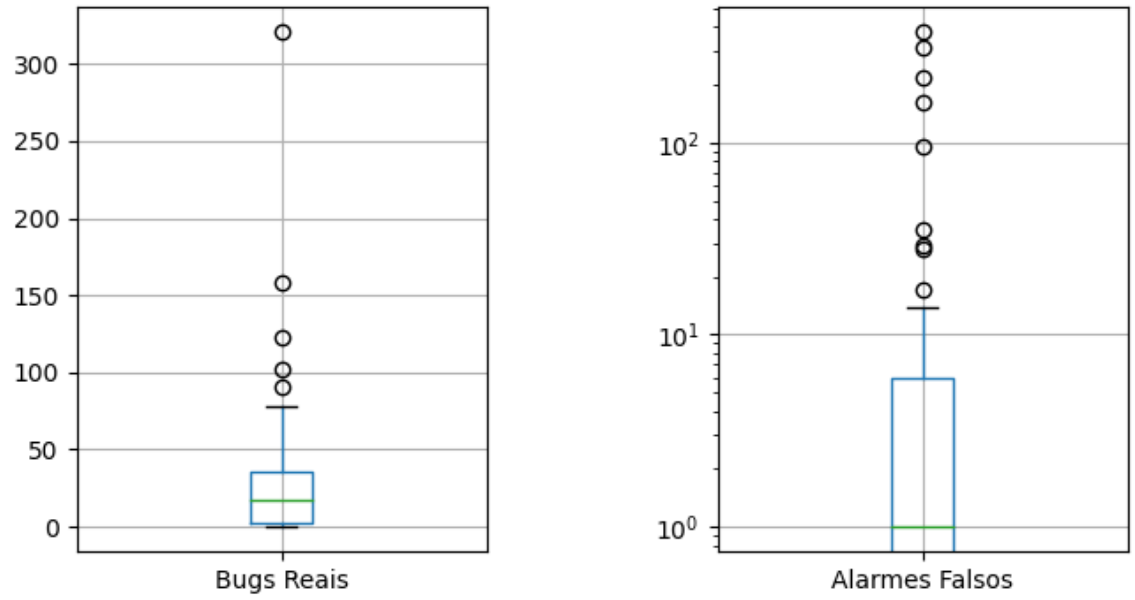
Tabela 4 – Aplicativos que apresentaram alertas de *bug*, mas foram classificados como Alarmes falsos

Aplicativo	**Alertas**	**Perda de dados**	**Diagnóstico (Alarmes Falsos)**
Compass	1	0	Atualização
Termux:Boot	1	0	Erro na captura
Egyptian Mouse Pounce	163	0	Animação
DSA Assistant	6	0	Mensagem temporizada

Fonte: Autor, 2022.

apresentam os diagramas de caixa (*boxplots*) desses dados. O eixo y é a quantidade de *Bugs reais* e Alarmes falsos, respectivamente, e os pontos são os aplicativos.

Figura 29 – [A] Análise da distribuição dos *Bugs reais*. [B] Análise da distribuição dos Alarmes falsos em escala logarítmica.



[A] [B]

Fonte: Autor, 2022.

Ao analisar o motivo da quantidade de Alarmes falsos, observamos que nove aplicativos apresentaram valores atípicos (*outliers*). Esse comportamento pode ser observado na Figura 29[B], cujos valores são apresentados em escala logarítmica. Enquanto a maior parte dos aplicativos apresentaram até dez Alarmes falsos, cinco aplicativos estão na faixa entre dez e cem alarmes e os outros quatro, acima de cem Alarmes falsos. Uma investigação mais detalhada revelou que os quatro aplicativos que geraram uma quantidade excessiva de Alarmes falsos (juntos geraram 1.169 Alarmes falsos e encontraram 72 *Bugs reais*) são aplicativos que

possuem animações. A Tabela 5 apresenta o número de *Bugs reais*, de Alarmes falsos e o diagnóstico com os motivos dos Alarmes falsos para os aplicativos considerados *outliers*. Embora o aplicativo Moonlight (linha 6 da Tabela 5) possua animação, ela só ocorre quando acontece o carregamento dos dados. Provavelmente esses Alarmes falsos poderiam ser minimizados se fosse ajustado um tempo maior entre as capturas do oráculo.

Tabela 5 – Diagnósticos dos aplicativos *outliers*.

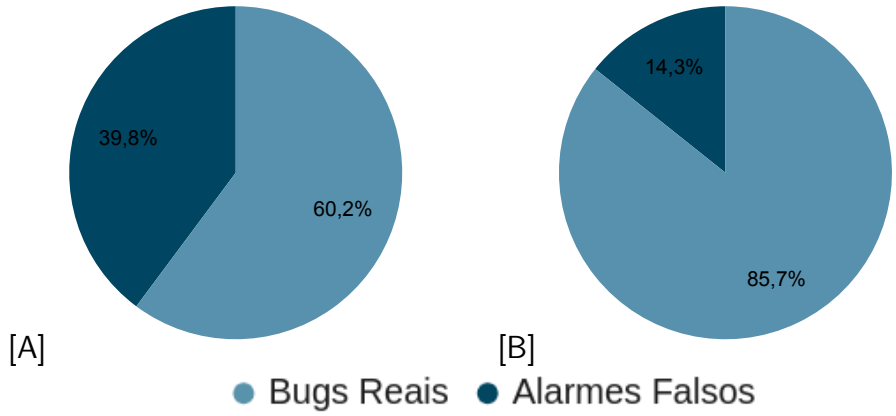
Aplicativo	***Bugs reais***	**Alarmes Falsos**	**Diagnóstico**
Material Files	97	28	Cursor piscando
Wikipedia	35	28	Teclado renderizando
VocableTrainer	20	28	Teclado renderizando
A Time Tracker	49	17	Alterou a hora
Goodtime	6	312	Animação no relógio
Moonlight	45	98	Animação de carregamento de dados da *activity*
OpenPods	0	163	Animação durante o uso do aplicativo
Baby Dots	1	379	Animação no aplicativo
Fiddle Assistant	18	219	Animação no aplicativo

Fonte: Autor, 2022.

Quando foram considerados todos os aplicativos, incluindo os casos com animação, a proporção é de 1,38 *Bugs reais* para cada Alarme falso. No entanto, quando excluímos esses aplicativos (Tabela 5 aplicativos com animação e mais de 100 Alarmes falsos), a proporção sobe para 6 *Bugs reais* para cada Alarme falso. A Figura 30 apresenta essa diferença nos dois casos.

Também foram analisados os percentuais de cobertura das atividades durante o experimento. A Figura 31 [A] apresenta o diagrama de caixa, onde o eixo y é o percentual de cobertura das atividades e os pontos são os aplicativos. A maior parte dos aplicativos tiveram cobertura acima de 40%, com mediana de aproximadamente 70%. Mesmo os aplicativos que tiveram cobertura abaixo de 40%, conseguiram encontrar uma quantidade não negligenciável de *Bugs reais*, com total de 351 *Bugs reais*. A quantidade de Alarmes falsos para os aplicativos de baixa cobertura foi de 470. No entanto, esse número elevado foi ocasionado pelo aplicativo Babydots, que apresentou 379 Alarmes falsos e revelou apenas 1 *Bug real*.

Figura 30 – [A] apresenta a classificação de *Bugs reais* e Alarmes falsos com todos aplicativos analisados. [B] apresenta a mesma classificação sem os quatro aplicativos que possuem animações.

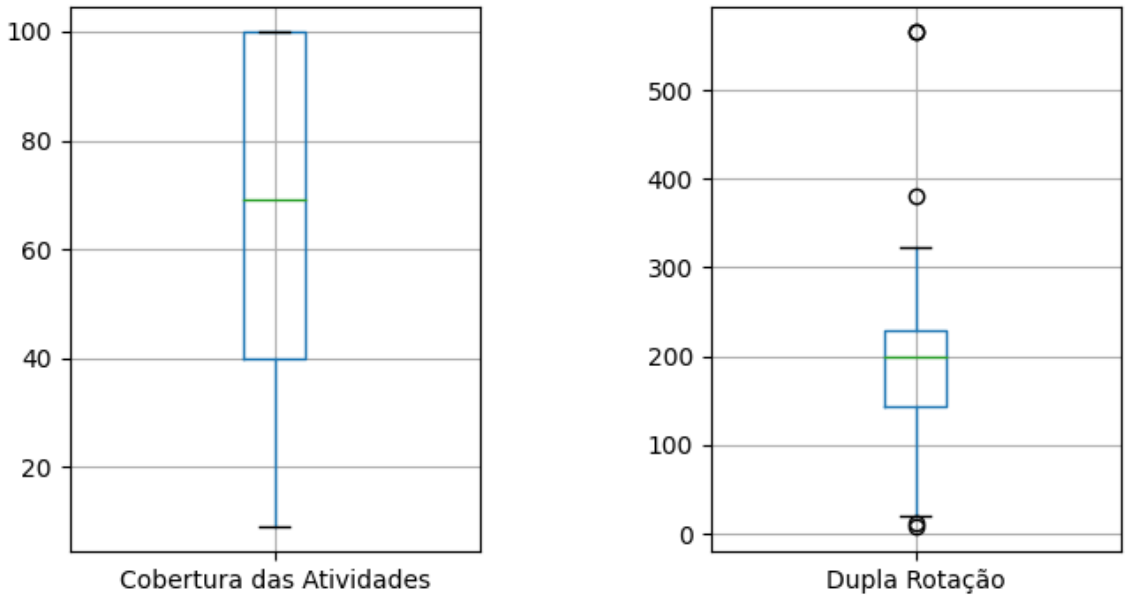


Fonte: Autor, 2022.

Outro ponto analisado foi a quantidade de verificações do oráculo, sinalizado pela dupla rotação no relatório gerado. A Figura 31 [B] mostra o diagrama de caixa dessa análise, onde o eixo y é a quantidade de verificações do oráculo e os pontos são aplicativos. Cinco aplicativos comportaram-se como *outliers*: dois abaixo do limite inferior e três acima do limite superior da distribuição. A maior parte dos aplicativos tiveram entre 145 e 230 verificações dos oráculos com a mediana em 200 verificações.

Uma análise mais detalhada dos resultados destes aplicativos confirmou a intuição que a quantidade de verificações realizadas não está, necessariamente, correlacionada com a quantidade de *Bugs reais* revelados. Para alguns aplicativos, o oráculo pode ser aplicado centenas de vezes e gerar poucos alertas, mas que correspondem a *Bugs reais*; e, para outros aplicativos, centenas de alertas podem ser gerados, mas todos relacionados a Alarmes falsos. A Tabela 6 apresenta a quantidade de alertas, *Bugs reais* e Alarmes falsos gerados para os aplicativos considerados *outliers* com base na quantidade de verificações realizadas. A variação na quantidade de duplas rotações entre os aplicativos está relacionado com a estratégia de exploração do DLD em interagir com todos os elementos da atividade para inserir valores diferentes do padrão antes de testar o oráculo (seção 2.3). Em alguns casos, o $\mathrm{R\text{-}DLD}$ pode ficar preso em uma atividade se for preciso executar uma sequência muito específica de toques na tela para mudar de atividade. Por exemplo, no aplicativo Babydots se for lançado um evento de toque em um ponto específico da tela o aplicativo é bloqueado precisando executar uma sequência de ações para desbloqueá-lo. Nesses casos, o $\mathrm{R\text{-}DLD}$ pode ficar aguardando uma condição (por exemplo, inserir valores diferentes do padrão) para testar o oráculo, e acaba consumindo muitos eventos.

Figura 31 – [A] Distribuição da cobertura das atividades. [B] Distribuição da verificação do oráculo por aplicativo (dupla rotação).



[A] [B]

Fonte: Autor, 2022.

Tabela 6 – *Outliers* da dupla rotação na análise de perda de dados.

Projetos	**Dupla rotação**	**Alertas**	***Bugs Reais***	**Alarmes Falsos**
Compass	566	1	0	1
StepandHeightcounter	12	12	12	0
bubble	8	3	3	0
Baby Dots	380	380	1	379
Mindustry	566	1	1	0

Fonte: Autor, 2022.

Resposta para RQ1

A relação do DLD foi de 4 *Bugs reais* para cada Alarme falso. Ao considerar todos os alertas, o R-DLD obteve 1,38 *Bugs reais* para cada Alarme falso. No entanto, ao retirar os aplicativos com animações (Tabela 5, classificados com animação e com mais de 100 Alarmes falsos), a proporção sobe para 6 *Bugs reais* para cada Alarme falso. Essa diferença pode estar relacionada com o tipo de aplicativo avaliado, uma vez que a remoção de cinco aplicativos com animação provocou uma mudança expressiva dessa proporção.

6.2 RQ2: A DETECÇÃO DE PERDA DE DADOS OBEDECE À MESMA PROPORÇÃO IDENTIFICADA PELO DLD PARA OS ORÁCULOS *SCREENSHOT-BASED* E *PROPERTY-BASED*?

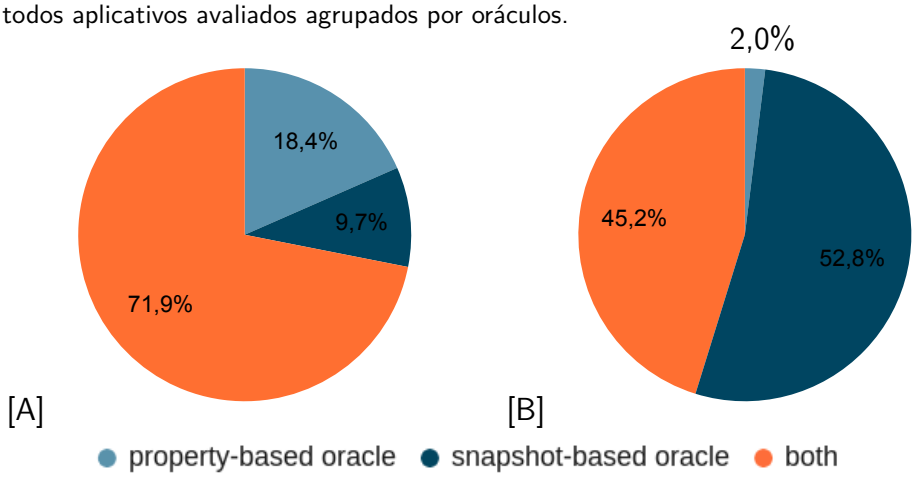
A avaliação empírica do DLD (RIGANELLI et al., 2020) utilizou um oráculo baseado em *screenshots* e outro baseado em propriedades. Um alerta de *bug* pode ser acionado por um dos oráculos ou por ambos. Os autores do DLD realizaram a classificação manual dos alertas em *Bugs reais* e Alarmes falsos e reportaram que 73,1% dos casos dos *Bugs reais* foram identificados por ambos os oráculos, 17,8% dos casos foram identificados apenas pelo oráculo de propriedades e 9,2% dos casos foram identificados apenas pelo oráculo baseado em *screenshots*. Com relação aos Alarmes falsos, 73,6% ocorreram em ambos os oráculos, 21,1% nos oráculos baseados em *screenshots* e 0,1% nos oráculos baseados em propriedades. Em 5,3% dos casos o oráculo falhou devido a erros em capturar informações corretas ocasionados pela demora na atualização dos aplicativos. Como solução, os autores sugeriram ajustes nos parâmetros do DLD para reduzir essas falhas.

Análise do experimento com todos os aplicativos

Considerando todos os aplicativos, sem excluir os aplicativos que possuem animações, os resultados observados em nosso estudo foram similares aos resultados reportados pelos autores do DLD nos casos de *Bugs reais*. No entanto, quando comparados com os Alarmes falsos, há uma distribuição diferente dos dados. A Figuras 32A e 32B apresentam a distribuição dos oráculos relacionados aos *Bugs reais* e Alarmes falsos, respectivamente, considerando todos os aplicativos. Com relação aos *Bugs reais*, ambos oráculos conseguiram identificar perda de dados em 71,9% dos casos. Para o oráculo baseado em propriedades, esse valor foi de 18,4% e para o oráculo baseado em *screenshots*, 9,7%. Na avaliação do DLD, os resultados foram 73,1%, 17,8% e 9,2%, respectivamente. Em relação aos Alarmes falsos, a Figura 32B mostra que ambos os oráculos conseguiram identificar a perda de dados em 45,2% dos casos. No oráculo baseado em propriedades, esse valor foi de 2,0% e para o oráculo baseado em *screenshots*, 52,8%. Na avaliação do DLD, os resultados foram 73,6%, 0,1% e 21,1%, respectivamente.

No caso de Alarmes falsos, há uma categoria adicional relacionada a erros na captura da informação, com o valor de 5,3% no estudo do DLD. Esse erro está relacionado com a lentidão do aplicativo em renderizar a *activity*, que induz o oráculo a acusar diferença entre as capturas

Figura 32 – As figuras A e B apresentam a proporção de *Bugs reais* e Alarmes falsos, respectivamente, de todos aplicativos avaliados agrupados por oráculos.



Fonte: Autor, 2022.

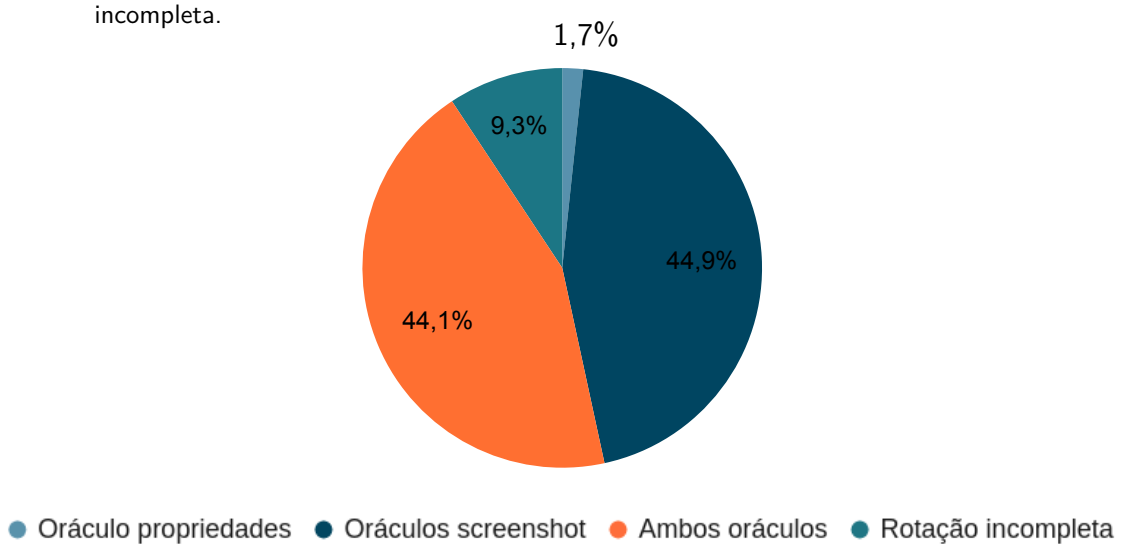
por conta das informações desatualizadas. Esse problema pode ser reduzido ajustando-se o parâmetro de tempo para o oráculo. Para efeito de comparação, foram atribuídos os Alarmes falsos relacionados ao tempo de atualização da tela para essa categoria.

A Figura 33 apresenta a distribuição nas quatro categorias. Os oráculos erraram juntos em 44,1% dos casos; o oráculo de propriedades errou em 1,7% dos casos; o oráculo baseado em *screenshots* em 44,9% dos casos; e as rotações incompletas foram responsáveis por 9,3% dos casos. A rotação incompleta acontece quando a captura do *screenshot* ocorre antes de concluir a mudança de orientação, pode estar relacionada com o nível de processamento do *smartphone* no momento da captura. A distribuição dos oráculos em relação a Alarmes falsos teve um comportamento diferente daquele observado no estudo do DLD, tanto em quantidade como em proporção. As diferenças observadas estão relacionadas com as características dos aplicativos avaliados.

Análise do experimento excluindo os aplicativos com animação

Os aplicativos com animações geram vários Alarmes falsos durante as análises pois, no momento da verificação, é sinalizada uma perda de dados devido às diferenças no posicionamento de algum elemento da *activity*. Apesar do oráculo baseado em *screenshots* ser mais sensível a esse tipo de comportamento, dependendo do tipo de animação esse tipo de Alarme falso também pode ser sinalizado por ambos os oráculos. Porém, esse Alarme falso nunca ocorre apenas no oráculo baseado em propriedades, pois é um problema essencialmente visual. Por

Figura 33 – A figura apresenta a proporção Alarmes falsos separados por oráculo e o critério da rotação incompleta.



Fonte: Autor, 2022.

exemplo, no aplicativo Babydot, todas as ocorrências foram geradas pelo oráculo baseado em *screenshots*; e no aplicativo Goodtime, alguns alertas ocorreram no oráculo de *screenshots* e outros, em ambos.

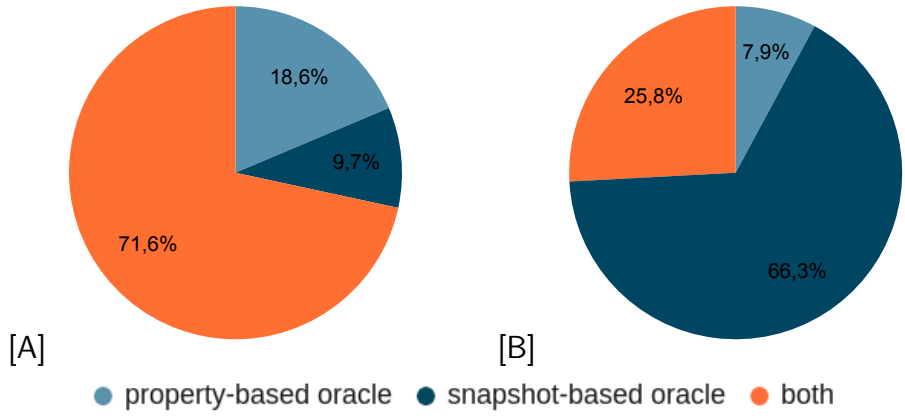
Quando a análise é realizada excluindo os quatro aplicativos com animações, o comportamento dos oráculos em relação aos *Bugs reais* continua semelhante ao estudo original. Contudo, há uma grande redução dos Alarmes falsos. A Figura 30B apresenta essa proporção ao retirar os aplicativos com animações: o percentual de Alarmes falsos reduz de 39,8% para 14,3% e o percentual de *Bugs reais* detectados aumenta de 60,2% para 85,7%.

Os quatro aplicativos com animação revelaram apenas 25 perdas de dados, por isso a exclusão não gerou grandes mudanças na distribuição dos oráculos (Figura 34A). Ambos oráculos descobriram bugs reais em 71,6% dos alertas, seguidos pelo oráculo de propriedades com 18,6%, e do oráculo baseado em *screenshots*, com 9,7% dos casos. Na avaliação do DLD estes valores foram 73,1%, 17,8% e 9,2%, respectivamente.

A maior parte dos alertas provocados pelas animações foi detectada pelos oráculos de propriedades e *screenshots* juntos. Ao retirar os aplicativos com animação, os casos de detecção do oráculo de *screenshots* tornaram-se a maioria, com 66,3%, seguidos pela detecção simultânea dos oráculos, com 25,8% e, por último, o oráculo de propriedades, com 7,9% (Figura 34B).

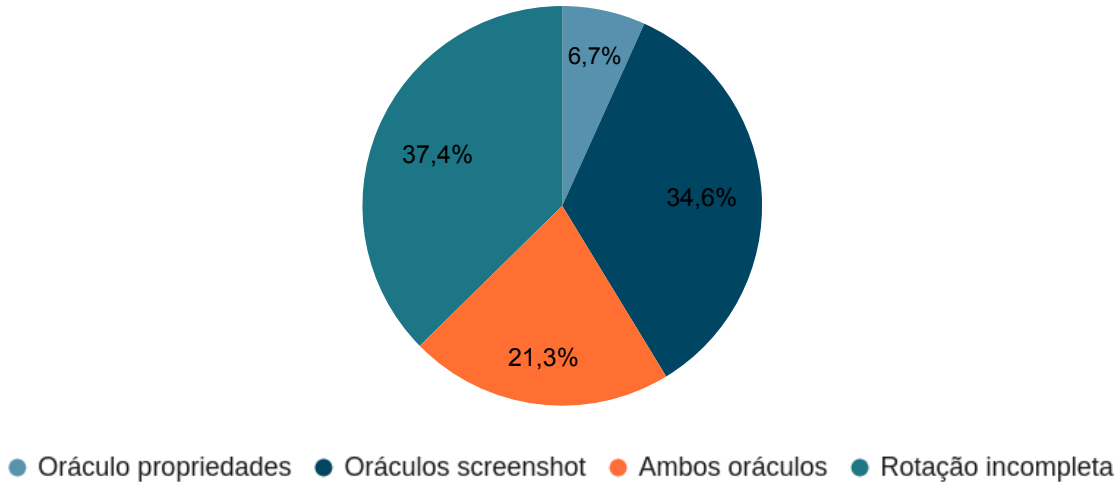
Ao utilizar o critério com as rotações incompletas (Figura 35), os casos de rotação incompleta tornaram-se a maioria, com 46,9%, seguidos de *screenshots* com 33,7%, ambos oráculos simultaneamente 14,3%, e oráculo de propriedades com 5,1%. Na avaliação do DLD estes valores foram 5,3%, 21,1%, 73,6% e 0,1%, respectivamente.

Figura 34 – As figuras A e B apresentam a proporção de *Bugs reais* e Alarmes falsos, respectivamente, retirando os aplicativos com animações.



Fonte: Autor, 2022.

Figura 35 – A figura apresenta a proporção Alarmes falsos separados por oráculo e o critério da rotação incompleta sem considerar os aplicativos com animações.



Fonte: Autor, 2022.

Diferenças entre os aplicativos nos dois estudos (DLD e R-DLD*)*

Em nosso estudo de replicação os casos de *Bugs reais* reportados com e sem animação tiveram um comportamento semelhante ao estudo original. No entanto, houve uma diferença significativa em relação aos Alarmes falsos, tanto em quantidade como em proporção. As diferenças observadas podem estar relacionadas com as características dos aplicativos analisados, principalmente nos casos dos aplicativos que possuem animações. Por exemplo, aplicativos que carregam dados da internet podem demandar dos oráculos um tempo que é diferente (maior) do que o tempo padrão (Nosso estudo de replicação não teve o objetivo de ajustar o parâmetro

de tempo entre os eventos para encontrar um número maior de *bugs*). Além disso, o estudo do DLD não reportou nenhum aplicativo com muitos alertas de *bugs*, o que sugere que não avaliaram casos de aplicativos com animações.

A grande quantidade de Alarmes falsos não implica, necessariamente, em maior esforço manual durante a etapa de análise dos alertas. Como esse tipo de alerta normalmente acontece em uma *activity* específica, possui um ou dois estados abstratos associados e são detectados por apenas um oráculo, é possível aplicar filtros nos alertas gerados e agrupar as capturas para uma análise manual ou automatizada mais eficiente.

Resposta para RQ2

Quando consideramos todos os aplicativos, os *Bugs reais* tiveram uma distribuição semelhante ao DLD. Ao realizar essa mesma análise sem os aplicativos com animação, essa distribuição permaneceu semelhante. No entanto, observamos diferenças em relação aos Alarmes falsos em relação ao DLD. O $\mathrm{R\text{-}DLD}$ obteve uma proporção maior de Alarmes falsos no oráculo de *screenshot-based* considerando o agrupamento dos aplicativos com e sem animação. Essa diferença pode estar relacionada com as características dos aplicativos analisados, pois a maior parte dos Alarmes falsos estão relacionados a atualização de tela, cursor e mensagem *toast*.

6.3 RQ3: O PROBLEMA DE PERDA DE DADOS É RELEVANTE PARA OS DESENVOLVEDORES?

Após analisar manualmente todos os 3.589 alarmes e classificá-los como *Bugs reais* e Alarmes falsos, verificamos que muitos alarmes estavam repetidos, algumas vezes reportando a falha de forma idêntica, e outras, com pequenas variações. Por exemplo, um formulário com a *string* "test" e o mesmo formulário com "teste123" reportados em alarmes diferentes. O agrupamento das falhas semelhantes resultou em um total de 341 *bugs*. Apenas 5 falhas não foram reportadas pois estavam associadas a projetos que foram arquivados (Bubble e TrebleShot). As demais 336 falhas encontradas foram reportadas aos desenvolvedores por meio de abertura de *issues* nos repositórios dos projetos no GitHub. Devido à quantidade de *bugs* por projeto, foram criadas uma ou mais *issues* para reportá-las, mas sempre sinalizando a relação com as demais, o que resultou em um total de 87 *issues* abertas.

Para contextualizar os desenvolvedores com o assunto de *perda de dados*, as descrições

das *issues* contemplaram os conceitos de ciclo de vida da atividade, o problema de perda de dados, a técnica de detecção do R-DLD e foram informadas possíveis soluções descritas na documentação do Android. Após essa contextualização, foi relatada, para cada bug, a sequência de passos para reproduzir o *bug*, o resultado esperado, o ambiente de configuração e alguns *links* para documentação do Android. Além disso, foram inseridos alguns artefatos produzidos durante a execução do R-DLD, por exemplo, *screenshots* antes e depois da dupla rotação, propriedades que foram alteradas e os toques na tela.

Até o presente momento, foram respondidas 52 *issues* (58,94% das *issues* reportadas), e 47 (89,55% das *issues* respondidas) tiveram um *feedback* positivo, mostrando o interesse do desenvolvedor sobre o problema.

Entre as issues que tiveram o *feedback* negativo (apenas 5), os desenvolvedores relataram não ter interesse em resolver a *issue* por ser um problema muito específico e com pouca probabilidade de ser reproduzido por um usuário. Outros, atribuíram o problema ao Android ou informaram que o projeto está sem manutenção, ou simplesmente fecharam a *issue* sem dar uma resposta.

Para as issues que tiveram *feedback* positivo, 18 foram resolvidas pelos integrantes dos projetos. Em alguns casos, uma solução parcial foi desenvolvida em pouco tempo, principalmente quando a falha reportada comprometia alguma funcionalidade importante do aplicativo. Em um caso, em particular, o desenvolvedor estava prestes a liberar uma nova versão reestruturada do aplicativo e os *bugs* já haviam sido corrigidos, quase que simultaneamente à abertura da *issue*. Em 25 casos os desenvolvedores deixaram as *issues* em aberto esperando um momento oportuno para resolvê-las. Em 4 ocorrências os desenvolvedores admitiram o problema de *perda de dados*, mas informaram que não tinham interesse em resolver e deixaram a *issue* aberta. A Tabela 7 apresenta os detalhes das *issues* que foram abertas.

Tabela 7 – Tabela com os resultados da abertura de *issue*.

ID	Aplicativo	Falhas	Status	Situação
1	Wikipedia	12	accepted	open
2	10-bitClockWidget	2	accepted	fix
3	Material Files	3	accepted	fix
4	Material Files*	3	rejected	closed
5	Transistor	1	accepted	fix
6	Transistor*	2	accepted	open
7	GetBack GPS	1	accepted	fix
8	GetBack GPS*	8	waiting for reply	
9	EteSync	10	accepted	open
10	EteSync*	1	accepted	open

Tabela 7 – Tabela com os resultados da abertura de *issue*.

ID	Aplicativo	Falhas	Status	Situação
11	Deedum	1	accepted	wontfix
12	HTTP Shortcuts	25	accepted	fix
13	HTTP Shortcuts*	2	accepted	fix
14	Simple Thank You	5	rejected	closed
15	GTFSOffline	2	waiting for reply	
16	Moonlight	3	accepted	fix
17	Moonlight*	2	accepted	fix
18	OpenPods	1	waiting for reply	
19	OkcAgent	1	waiting for reply	
20	Unit Converter Ultimate	1	waiting for reply	
21	Baby Dots	1	accepted	fix
22	NoProvider2Push	1	waiting for reply	
23	Jiten-webview	1	waiting for reply	
24	Kõnele	4	accepted	open
25	Mindustry	1	rejected	closed
26	Updater For Spotify	1	waiting for reply	
27	Gerberoid	2	accepted	open
28	VocableTrainer	10	waiting for reply	
29	VocableTrainer*	1	accepted	bug
30	DokuwikiAndroid	11	accepted	fix
31	Mensa	3	accepted	open
32	PanicTrigger	5	waiting for reply	
33	FakeStandby	1	accepted	bug
34	Fiddle Assistant	2	accepted	open
35	Zandy	1	waiting for reply	
36	Zandy*	16	waiting for reply	
37	A Time Tracker	1	accepted	open
38	A Time Tracker*	12	accepted	open
39	Chibe	4	waiting for reply	
40	Wirebug	1	waiting for reply	
41	Traductor Softcatala Android	2	waiting for reply	
42	Snapcast	4	rejected	closed
43	StepandHeightcounter	1	accepted	wontfix
44	StepandHeightcounter*	5	waiting for reply	
45	Vigilante	2	accepted	fix
46	Vigilante*	3	accepted	closed
47	Odeon	2	accepted	open
48	Stanley	1	waiting for reply	
49	Stanley*	5	waiting for reply	
50	Squeezer	1	waiting for reply	
51	Squeezer*	6	waiting for reply	
52	Activity Manager	2	accepted	fix
53	Activity Manager*	6	accepted	wontfix
54	Missed Notifications Reminder	1	accepted	open
55	Missed Notifications Reminder*	4	waiting for reply	
56	ScreenshotTile	5	accepted	fix
57	Headi	4	accepted	fix
58	JRPN_Android	7	accepted	wontfix
59	NLWeer	1	accepted	bug
60	NLWeer*	5	accepted	bug
61	A minimalist keyboard	2	accepted	bug
62	EtchDroid	2	accepted	open

Tabela 7 – Tabela com os resultados da abertura de *issue*.

ID	Aplicativo	Falhas	Status	Situação
63	Privacy Friendly Food Tracker	1	waiting for reply	
64	Privacy Friendly Food Tracker*	5	waiting for reply	
65	CanIDrive	2	accepted	fix
66	CanIDrive*	2	accepted	fix
67	DNG Processor	4	waiting for reply	
68	Status Bar Speedometer	2	waiting for reply	
69	Ichaival	1	accepted	fix
70	Ichaival*	7	waiting for reply	
71	Acastus	4	waiting for reply	
72	Shorty	8	rejected	closed
73	Split It Easy	4	waiting for reply	
74	eSpeak NG Text-to-Speech	1	waiting for reply	
75	eSpeak NG Text-to-Speech*	8	waiting for reply	
76	Privacy Friendly Weather App	5	accepted	bug
77	Inflation Calculator	3	accepted	fix
78	Easer	1	accepted	bug
79	Easer*	7	waiting for reply	
80	Japanese Traditional Time	4	accepted	open
81	Nextcloud services	4	accepted	open
82	Mupen64Plus-AE	10	accepted	bug
83	Privacy Friendly Werewolf	1	waiting for reply	
84	Privacy Friendly Werewolf*	11	waiting for reply	
85	SpotIt	2	waiting for reply	
86	CleanTimer	1	waiting for reply	
87	Goodtime	2	accepted	open
	Total	**87**	**336**	

Fonte: Autor, 2022.

* Alguns aplicativos tiveram mais de um *issue* reportados aos desenvolvedores devido a quantidade de falhas encontradas.

Resposta para RQ3

Os 341 bugs foram reportadas em 87 *issues*, em que foram respondidas 52 (58,94%) *issues* pelos desenvolvedores. Dos respondidos, 47 *issues* tiveram os bugs de perda de dados confirmados. Essas 47 *issues* aceitas correspondem a 180 bugs e representam 89,55% dos bugs respondidos. Dessa forma, é possível inferir que o problema de perda de dados é relevante aos desenvolvedores.

6.4 DISCUSSÃO

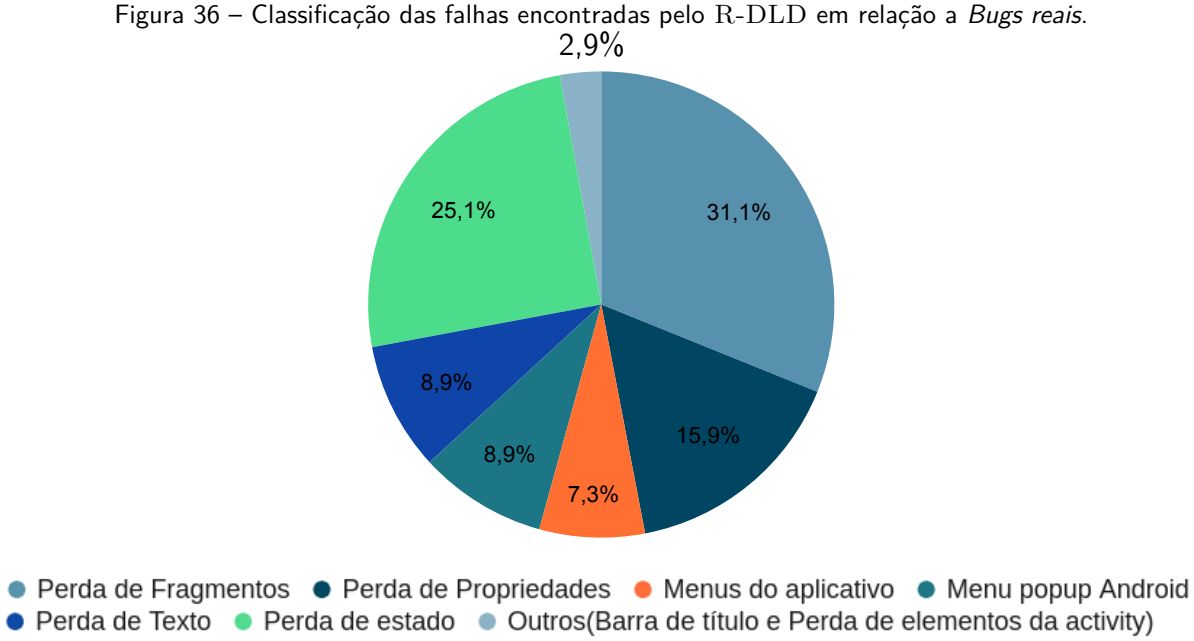
Nessa seção iremos apresentar as características dos *Bugs reais* e Alarmes Falsos identificadas no experimento de replicação. Depois iremos discutir a viabilidade da utilização de robótica para a automação de testes de perdas de dados.

6.4.1 Características dos alertas gerados pelo R-DLD

Nesta seção discutimos as características dos alertas gerados pelo R-DLD utilizando as categorias apresentadas anteriormente na Seção 4.

A Figura 36 apresenta os *Bugs reais* encontrados nos aplicativos avaliados (desconsiderando os alertas repetidos), agrupados por categoria de perda de dados.

Figura 36 – Classificação das falhas encontradas pelo R-DLD em relação a *Bugs reais*.



Fonte: Autor, 2022.

A principal ocorrência foi a perda de fragmento com 31,1%, seguida pela perda de estado com 25,1% e propriedades, com 15,9%. O Problema de perda de texto inserido pelo usuário foi encontrado em 8,9% dos aplicativos avaliados. Problemas como menu *popup* do Android e menus do aplicativo ocorreram em 8,9% e 7,3%, respectivamente. Outros casos, como perda de elementos da *activity* e problemas na barra de título, ocorreram em 2,9% dos casos.

A perda de fragmento está relacionada com a forma como o desenvolvedor implementa os diálogos do aplicativo, fazendo uso de práticas não recomendadas. A perda de dados pode

ocorrer, por exemplo, ao se instanciar um diálogo diretamente na *activity*. Uma boa prática é mover a ação dos componentes que são cientes do ciclo de vida para o próprio componente. Ou seja, ao invés de instanciar diretamente um diálogo, deve-se criar uma classe que estende o DialogFragment e implementar os métodos para o correto funcionamento do diálogo.

Tabela 8 – Quantidade de falhas agrupadas por tipo e oráculos

Tipo de perda de dados	**Quantidade**	**Oráculo Propriedades**	**Oráculo Screenshot**	**Ambos Oráculos**
Perda de Fragmentos	473	0	0	473
Perda de Propriedades	397	347	0	50
Menu popup aplicativo	115	0	88	27
Menu popup Android	165	0	40	125
Perda de Texto	154	10	16	128
Perda de estado	638	35	53	550
Barra de título	134	2	9	123
Menu principal	65	0	0	65
Perda de elementos da activity	19	4	3	12

Fonte: Autor, 2022.

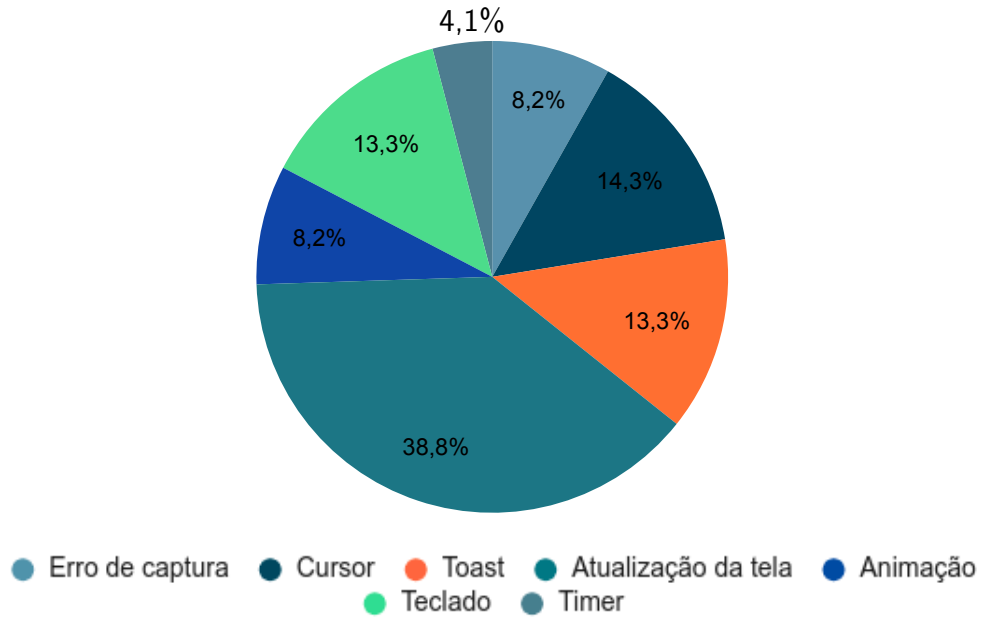
A perda de estado está associada à dificuldade do aplicativo em retornar ao ponto exato antes do evento de dupla rotação. Muitas vezes esse erro é causado por pequenos *scrolls* nas telas antes e depois da rotação que podem mostrar uma parte maior de um componente ou um componente que estava oculto. Quando o primeiro caso ocorre, aciona-se o oráculo de *screenshot* e, no segundo, os oráculos de *screenshot* e propriedades juntos.

A Tabela 8 permite analisar o comportamento dos oráculos para as diferentes categorias de perda de dados (todos os alertas). A perda de fragmento é revelada por ambos oráculos. Já a perda de estado tem prevalência em ambos os oráculos, mas também pode ocorrer com menor probabilidade nos oráculos de propriedades e *screenshot* isoladamente.

Com relação aos Alarmes falsos, a Figura 37 apresenta a distribuição dos alarmes, agrupados por categoria (desconsiderando os alertas repetidos). O R-DLD reportou Alarmes falsos de atualização de tela em 38,8% dos aplicativos. Outras fontes de falhas na classificação foram Cursor, Teclado e Mensagem *toast*, com 14,3%, 13,3% e 13,3%, respectivamente. Erro de captura, *timer* e animação representaram 8,2%, 4,1% e 8,2%, respectivamente.

O Alarme falso de atualização de tela é ocasionado pela demora do aplicativo em atuali-

Figura 37 – Classificação das falhas encontradas pelo R-DLD em relação a Alarmes Falsos.



Fonte: Autor, 2022.

zar a *activity*, muitas vezes devido ao carregamento de dados ou processamento elevado do smartphone. Para diminuir a interferência desse tipo de falha, os autores do DLD sugerem ajustes no tempo entre os oráculos, pois aplicativos que fazem uso de carregamento de dados ou processamentos extras podem gerar mais Alarmes falsos.

A Tabela 9 apresenta a distribuição dos Alarmes falsos por oráculos (todos os alertas). Embora as animações tenham gerado 1.022 Alarmes falsos do total de 1.429 reportados pelo R-DLD, apenas 8,2% dos aplicativos avaliados apresentaram Alarmes falsos.

Tabela 9 – Quantidade Alarmes falsos agrupados por tipo e oráculos

Tipo de Alarmes falsos	**Quantidade**	**Oráculo Propriedades**	**Oráculos Screenshot**	**Ambos Oráculos**
Erro de captura	34	6	3	25
Cursor	70	3	60	7
Toast	93	0	93	0
Atualização da tela	133	4	113	16
Animação	1.022	0	468	554
Teclado	60	0	16	44
Tempo	17	15	2	0

Fonte: Autor, 2022.

6.4.2 Estudo de viabilidade de utilização de robótica para a automação de testes de perdas de dados

Quais os custos de configurar o ambiente de testes? Em um contexto onde a equipe já trabalha com robôs, inserir a atividade de teste de perda de dados é uma tarefa simples, pois o esforço fica na integração do DLD com o sistema de controle do robô. No entanto, quando é necessário construir toda a infraestrutura, o custo de implantação pode variar bastante. Caso a equipe opte por adquirir um robô industrial com a capacidade de realizar diversos tipos de movimentos em diferentes graus de liberdade, com controle de velocidade (por exemplo, um robô da Denso Robotics[1]), este custo poderá chegar a algumas dezenas de milhares de dólares. O braço robótico artesanal construído como parte deste trabalho para rotacionar o smartphone tem um custo bem menor (a versão apresentada na Figura 5 custou aproximadamente R$ 300,00). Durante o experimento de replicação, com duração aproximada de 350 horas, o braço robótico não apresentou defeitos nem precisou realizar calibrações adicionais devido ao número de mudanças de orientações. A calibração descrita no Capítulo 3 é realizada ao ligar o equipamento, não sendo necessário realizá-la durante o teste.

Quais as dificuldades para integrar a abordagem robótica em uma ferramenta sem suporte para tal? A integração da ferramenta de testes automatizados com o robô é uma tarefa relativamente simples que consiste em interceptar os comandos ADBs e substituí-los por instruções robóticas. Nesse caso, a instrução robótica é um comando para rotacionar um objeto preso em um suporte. No entanto, isso depende de alguma API que a ferramenta disponibilize ou do código fonte. A integração com o DLD foi possível porque os desenvolvedores disponibilizaram o código fonte. Outro ponto crítico é a velocidade do braço robótico em realizar uma mudança de orientação. Durante o experimento de validação do braço robótico verificamos que esse tempo foi inferior a 2 segundos para cada mudança de orientação. Por padrão o DLD utiliza o tempo de 3 segundos entre eventos o que permite o braço robótico realizar as duas mudanças de orientações sem interferir na verificação do oráculo.

Existem vantagens em utilizar dispositivos reais em comparação aos dispositivos emulados? Um AVD é uma simulação de um perfil de hardware para executar no emulador do Android em que os componentes são abstrações dos dispositivos reais. Embora se assemelhe a um dispositivo físico, um AVD possui limitações. Por exemplo, não é possível executar testes

[1] <https://www.densorobotics.com/>

que envolvam ligações telefônicas, conexão *bluetooth* ou algoritmos de retificação de imagem (requer uso da câmera do dispositivo). Além disso, durante o ciclo de vida de um smartphone, é comum ter versões diferentes de hardware, algumas vezes para corrigir *bugs*, outras, por conta de alteração de componentes. Por isso, realizar testes somente em simulador pode não detectar *bugs* que ocorram ao interagir com o hardware.

6.5 AMEAÇAS À VALIDADE

A principal ameaça interna do nosso estudo é a classificação manual dos alertas em *bug* verdadeiro ou Alarme falso. A classificação, embora seja um procedimento simples em que se verifica a existência de diferenças entre as propriedades ou os *screenshots*, não é uma atividade trivial. Ela requer o conhecimento do comportamento da aplicação, para verificar se o que está sendo considerado uma perda de dados não é, na verdade, um comportamento esperado da aplicação. Por exemplo, em casos em que a atividade possui um temporizador, os oráculos de *screenshots*, propriedades ou ambos podem causar uma perda de dados, no entanto, as diferenças observadas representam o comportamento esperado da aplicação. Nesses casos, o alerta gerado deve ser considerado um Alarme falso. A principal ameaça externa diz respeito à generalização dos resultados. Nosso estudo foi realizado em uma amostra de 77 aplicativos coletados de uma loja de aplicativos do ecossistema Android. O controle para essa ameaça só pode ser alcançado através da realização de estudos adicionais, considerando diferentes aplicativos, diferentes domínios e perguntas de pesquisa adicionais.

6.6 CONSIDERAÇÕES FINAIS

Neste capítulo apresentamos os resultados do estudo de replicação utilizando a ferramenta R-DLD. Relatamos a análise preliminar dos dados, os resultados do nosso estudo e as comparações das proporções de *Bugs reais* e Alarmes falsos em relação ao estudo do DLD. Por fim, apresentamos o processo de abertura de *issues*, as respostas das questões de pesquisas e as ameaças à validade.

7 TRABALHOS RELACIONADOS

Os trabalhos relacionados à infraestrutura do R-DLD foram divididos em duas categorias: teste de perda de dados e uso de robôs para testes em smartphones.

7.1 TESTES DE PERDA DE DADOS

Nesta seção, iremos apresentar trabalhos que realizam testes de perda de dados na GUI por meio de ferramentas automatizadas.

Zaeen *et al.* (ZAEEM; PRASAD; KHURSHID, 2014) apresentam a ferramenta QUANTUM, um gerador automático de casos de teste que incluem oráculos de teste para explorar a GUI em aplicativos móveis. A ferramenta foi criada com base em um estudo nos registros de *bugs* dos aplicativos. Selecionaram 106 *bugs* reproduzíveis de 13 aplicativos para identificar oportunidades de geração automática de casos de testes. Cada *bug* foi categorizado manualmente com a finalidade de identificar o tipo de oráculo que poderia ser usado e agrupado em categorias. O grupo de *App Agnostic*, que é composto pelas categorias dos oráculos de Rotação, Ciclo de vida da atividade e Erros de gestos foi escolhido para implementar a ferramenta QUANTUM. A avaliação foi realizada com 6 aplicativos. A ferramenta gerou 60 testes baseados em um modelo de GUI criados manualmente e encontrou um total de 22 *bugs*. Embora esse trabalho tenha o foco em falhas de GUI, ele também explora problemas de *perda de dados* quando explora o ciclo de vida da atividade.

Amalfitano *et al.* (AMALFITANO et al., 2018) realizaram dois experimentos *Blackbox*, com uma abordagem não-invasiva, para identificar ocorrências de falhas na GUI usando a mudança de orientação em dispositivos reais. O *termo falhas de GUI* encontrados nesse artigo é semelhante ao termo *perda de dados* reportado no DLD e R-DLD. O primeiro experimento optou por escolher aplicativos com código fonte para analisar as causas das falhas, reportá-las aos desenvolvedores e elencar soluções. O conjunto de dados foi extraído da loja F-Droid e apresentou 439 falhas de GUI nos 68 aplicativos analisados. O segundo experimento foi realizado com aplicativos mais populares, que tiveram boa classificação e mais de 50 milhões de *downloads*. Teve a finalidade de verificar se os aplicativos de grandes empresas tinham problemas de falhas na interface. Foram selecionados 10 aplicativos no Google Play que apresentaram um total de 140 falhas. Nos dois casos, o processo de detecção foi realizado de forma manual por alunos

de pós-graduação, que interagiam com o dispositivo e verificavam se existiam alterações na tela após a dupla rotação. A cada *bug* encontrado, foi criado um *script* usando o Robotium (ROBOTIUM, 2022) para reproduzi-lo. Por fim, foi observado que falhas na GUI ocorrem em aplicativos de pequeno e grande porte, e a mudança de orientação é uma excelente estratégia para identificar esses *bugs*.

Riccio *et al.* (RICCIO; AMALFITANO; FASOLINO, 2018) apresentam ALARic, uma ferramenta para testes automatizados que combina técnicas de exploração dinâmica de aplicativos com três eventos específicos (*Double Orientation Change (DOC)*, *Background Foreground (BF)* e *Semi-Transparent Activity Intent (STAI)*) para explorar o ciclo de vida da atividade. A arquitetura da ferramenta é composta por duas partes: ALARic Engine é responsável por implementar a lógica de negócios da abordagem de teste, no qual recebe como entrada um arquivo de configuração que é utilizado para configurar o processo de teste; e o Test Executor, que é responsável por executar as atividades de teste ao interagir com um dispositivo físico e ou um AVD, sendo composto por dois componentes *Robot* e *Driver*. O primeiro interage com o dispositivo disparando eventos e descrevendo as GUIs em tempo de execução, e o segundo, desacopla a lógica de negócios implementada no ALARic Engine. O experimento foi realizado com 15 aplicativos e encontraram 106 *Bugs reais*, e concluíram que o evento DOC tem maior probabilidade de expor problemas vinculados ao ciclo de vida da atividade.

Riganelli *et al.* (RIGANELLI et al., 2020) desenvolveram a ferramenta Data Loss Detector (DLD) para identificar perda de dados em dispositivos Android com uma abordagem invasiva. Ou seja, DLD usa inserção de comandos ADBs para simular a interação com o dispositivo. A estratégia do DLD foi expandir o DroidBot (LI et al., 2017) para explorar a perda de dados após a dupla rotação, adicionando um oráculo para verificar a diferença entre *screenshots* e outro, para verificar a diferença entre propriedades. O processo de detecção de *perda de dados* é automático e consiste em parametrizar a ferramenta com um aplicativo, o caminho para salvar os artefatos e a quantidade de eventos para definir o critério de parada dos testes. No experimento apresentado, foram utilizados 2.250 eventos que tiveram um tempo médio em AVD de três horas. Os principais benefícios ao utilizar a ferramenta foram a abordagem *Blackbox* e a geração automática de testes, que utiliza uma estratégia exploratória para cobrir as *Activities* e outra de interação para alterar os valores padrão dos elementos. Para validar a ferramenta, comparou-se com outras abordagens de detecção de perda de dados utilizando AVD.

Guo *et al.* (GUO et al., 2022a; GUO et al., 2022b) apresentam o iFixDataloss, uma ferramenta

para detectar e corrigir problemas de perda de dados em aplicativos Android. O iFixDataloss realiza uma análise estática em que cria um grafo de transição do aplicativo em testes e uma análise dinâmica. E utiliza como entrada a análise estática para limitar as ações com uma estratégia de exploração guiada. Para cada *activity* do aplicativo, são executados testes e eventos predefinidos que abrangem vários cenários de perda de dados. Ao encontrar um problema de *perda de dados*, o iFixDataloss usa um modelo para gerar um *patch* de correção. O processo de correção consiste em identificar as variáveis, analisando o código e o arquivo XML que descrevem a GUI para rastrear o fluxo de dados. Caso esses dados sejam utilizados em chamadas de API, o iFixDataloss os considera como dados persistentes. Esses dados foram divididos em três categorias: valores de *widgets* editáveis que precisam ser armazenados durante a execução do aplicativo, valores de *widgets* editáveis que precisam ser armazenados para uma única execução do aplicativo, e valores de *widgets* não-editáveis. Para cada categoria, existe uma forma de corrigi-las com um *patch* seguindo as recomendações do Android. Caso ocorra um problema de *perda de dados*, o iFixDataloss utiliza um modelo de *patch* para salvar e restaurar as variáveis, seguindo o critério mais adequado. Após criar os *patchs* para cada variável, é realizada uma nova verificação nas *activities* corrigidas a fim de verificar se ainda ocorrem problemas de *perda de dados* ou se as correções ocasionaram problemas de travamento.

Os trabalhos de Amalfitano e Riganelli se apresentam como uma abordagem *Blackbox* para detecção de perda de dados. No entanto, um detecta a *perda de dados* de forma manual, usando técnicas não-invasivas, e o outro, automatizada, com técnicas invasivas. Já o trabalho de Guo (iFixDataloss) apresenta uma abordagem *greybox*, pois inicia com uma análise estática do código, para limitar a estratégia de exploração, e depois realiza uma análise sistemática do aplicativo com ações que priorizam acionar transições na atividade. A análise sistemática é realizada de forma automatizada utilizando técnicas invasivas. O $\mathrm{R\text{-}DLD}$ explora os benefícios da robótica para produzir testes mais semelhantes ao uso pelos usuários de smartphones. Ou seja, $\mathrm{R\text{-}DLD}$ é uma abordagem *Blackbox* menos invasiva, já que a rotação é física (feita pelo robô), mas a interação com a GUI é via ADB.

7.2 USO DE ROBÔ PARA TESTES EM SMARTPHONES

Robôs são amplamente utilizados para muitas tarefas repetitivas e o interesse em usá-los no contexto de teste de dispositivos móveis vem crescendo recentemente (MACIEL et al., 2022).

Como utilizamos robôs para a descoberta de *perda de dados*, também apresentamos alguns trabalhos que descrevem a aplicação de robôs para atividades de teste de software.

Banerjee, Yu e Aggarwal (BANERJEE; YU; AGGARWAL, 2018a) utilizam robôs para testar algoritmos de *hand jitter reduction* (HJR) simulando movimentos semelhantes aos usuários do mundo real. Algoritmos HJR são usados para suavizar ruídos das imagens das câmeras provocados pelos tremores da mão do usuário durante a utilização. A motivação deste trabalho é desenvolver cenários com diferentes tipos de movimentos que não são normalmente cobertos em testes HJR e utilizando-se robôs para identificar os limites de tolerância dos algoritmos. Os movimentos do robô foram gerados de acordo com os possíveis usos da câmera e foram inseridos ruídos para simular ações do cotidiano do usuário, como fazer gravações andando ou tremer ao tirar uma fotografia. O robô é essencial para permitir a comparação entre os algoritmos por causa da sua capacidade de repetir com precisão um conjunto de movimentos programados. Assim, é possível comparar o desempenho do algoritmo em condições controladas.

Banerjee e Yu (BANERJEE; YU, 2018) também utilizaram um braço robótico para automatizar cenários de testes para algoritmos de reconhecimento facial em dispositivos móveis. Vários testes devem ser desenvolvidos, pois estes algoritmos são usados como validação de segurança para desbloquear um smartphone. Assim, usando um braço robótico, é possível realizar testes e comparar os algoritmos em cenários distintos simulando o comportamento do usuário desbloqueando o smartphone. Por exemplo, simular o desbloqueio do smartphone com o usuário em movimento, com oclusão facial ou em diferentes ângulos. A capacidade de executar com precisão as tarefas permitiu uma melhor comparação entre os algoritmos de reconhecimento facial, pois diminui o erro humano pela fadiga ou falta de atenção.

Em outro trabalho, Banerjee, Yu e Aggarwal (BANERJEE; YU; AGGARWAL, 2018c) utilizaram braços robóticos para automatizar testes em algoritmos de rastreamento de objetos (*touch-to-track*). A utilização do robô permitiu expandir os cenários de testes ao reproduzir o mesmo teste variando a iluminação, o ângulo e adicionando ruídos como vibrações na horizontal ou vertical para simular limitações dos usuários. A capacidade de reproduzir os movimentos com precisão, independente do número de repetições, permitiu comparar algoritmos *Touch-to-track*.

Em (BANERJEE; YU; AGGARWAL, 2018b), utilizou-se um braço robótico para automatizar testes de algoritmos de retificação de textos utilizando a câmera do smartphone. O principal benefício apontado pelo autor em utilizar robôs é o aumento da cobertura, a redução dos testes manuais e a melhor eficiência e escalabilidade do teste.

Em (BANERJEE; YU, 2019), apresenta-se uma solução para criar um teste integrado automatizado de captura de imagem utilizando o robô para eliminar os testes manuais. A solução manual precisava que o engenheiro de teste iniciasse e sincronizasse os testes automatizados. Nessa tarefa, existiam dois artefatos: um *script* de teste e o *script* do robô. O *script* de teste era iniciado no computador para se comunicar com o smartphone e, em paralelo, por meio do painel de controle do robô, era iniciado o script do robô. Esse cenário precisava da atenção do engenheiro de teste para mantê-los sincronizados. Para resolver esse problema, foi proposta uma estratégia de automação de ponta a ponta, que utiliza um *middleware* para abstrair a complexidade do acionamento do robô para o *script* de teste. Os testes de captura de movimento realizados usando essa abordagem foram facilmente executados e permitiram produzir análises comparativas entre diferentes abordagens devido à precisão dos movimentos e a facilidade de repeti-los.

Em (BANERJEE; YU, 2020), apresenta-se uma proposta para automatização de testes usando braço robótico para testes de reconhecimento facial em smartphones. O objetivo é realizar testes de autenticação facial 3D em dispositivos móveis sob condições reais envolvendo movimento do dispositivo, iluminação, distância da face, detecção de olhar, falta de foco, etc. Utilizar robôs para testes de autenticação facial permite realizar atividades repetitivas com precisão, eliminando erros humanos ocasionados por cansaço físico ou falta de atenção. Além disso, permitiu implementar situações do mundo real de forma rápida e precisa como realizar testes com o smartphone em movimento, sob diversos ângulos e distâncias da face do usuário.

K. Mao *et al.* (MAO; HARMAN; JIA, 2017) apresentam a ferramenta Axiz, um gerador de testes robóticos para aplicativos móveis que utiliza um dispositivo real em uma abordagem de teste caixa-preta. Sua arquitetura é composta por dois componentes de alto nível: o gerador de testes robótico e o executor de testes robóticos. O primeiro analisa o aplicativo e gera um caso de teste usando *Evolucionary Search*. O segundo, o executor de testes robóticos, converte o *script* de testes em comandos executados pelo robô para interagir com a interface ciber-física do dispositivo de forma semelhante ao usuário. Além disso, processa imagens de uma câmera usando técnicas de visão computacional para detecção de objetos e comparação com o oráculo. Os pontos em comuns que o Axiz tem com o $\mathrm{R\text{-}DLD}$ são: a utilização do robô para executar testes, a utilização da interface ciber-física do dispositivo e a geração automática de testes. No entanto, para criar casos de testes eficientes, a ferramenta Axiz necessita de uma suíte de testes pré-definidos para gerar um modelo realista. Isso pode gerar

dois problemas: o primeiro está relacionado ao viés introduzido pela falta de diversidade de testes; e o segundo, é o viés introduzido pelos vícios de programação da equipe de testes. O R-DLD gera testes realistas sem a necessidade de acessar uma suíte de testes, pois utiliza o DLD para identificar os elementos da *activity* e o conjunto de interações possíveis, e gera eventos aleatórios respeitando esses requisitos.

J. Qian *et al.* (QIAN et al., 2020) apresentam a ferramenta RoScript, um sistema de testes robóticos baseado em *script* que utiliza uma abordagem não intrusiva para testes de GUI em dispositivos com tela de toque. Diferente das abordagens intrusivas, que obtém informações da GUI por meio de acesso ao sistema operacional do dispositivo, utiliza-se de visão computacional para identificar os elementos e o estado da GUI. E, por meio de um robô, interage com o dispositivo usando a interface ciber-física. A criação de *script* pode ser realizada de forma manual ou auxiliada por um componente que identifica gestos do ser humano no dispositivo em teste e traduz em uma sequência de comandos para ser reproduzido pelo robô. Esses gestos são identificados usando visão computacional processando vídeos com os passos da execução do teste e convertido para comandos suportados pelo robô. Esse processo se diferencia das abordagens tradicionais que utilizam aplicativos instalados no dispositivo em testes para gravar esses passos.

Craciunescu *et al.* (CRACIUNESCU et al., 2018) apresentam um robô portátil, de baixo custo, desenvolvido para teste de dispositivos móveis que funciona em telas do tipo resistivo e capacitivo. Por utilizar técnicas de processamento de imagens, consegue ter *feedback* das ações executadas e ser robusto para encontrar elementos na tela mesmo com reposicionamento do dispositivo. A construção do robô foi baseada em robôs delta que, segundo os autores, são mais rápidos e precisos se comparados aos robôs cartesianos. O robô utiliza o algoritmo SURF (*Speeded Up Robust Features*) para detecção de elementos do smartphone.

Juang e Cheng (JUANG; CHENG, 2017) utilizam visão computacional e um braço robótico para realizar testes em smartphone, com objetivo de criar um sistema robótico que aceite um caso de teste e execute semelhante a um humano em um dispositivo. Utilizando duas câmeras 2D, calcula-se a distância dos objetos de forma semelhante à visão humana. E consegue-se controlar um braço robótico em um espaço 3D sem nenhuma calibração por meio da teoria *fuzzy*. No caso de falha no toque, há um processo de ajuste simples para ajustar para a área desejada. Com auxílio de outra câmera e utilizando técnicas de reconhecimento ótico de caracteres (*Optical Character Recognition* (OCR)), identifica letras, números e ícones na tela do smartphone.

Vermaa *et al.* (VERMAA et al., 2017) apresentam um robô de baixo custo que permite realizar ações no smartphone utilizando o *touch screen*. A solução engloba interações de toque simples, toque duplo e rolagem de tela. O robô foi construído com um projeto de coordenadas cartesianas usando uma placa Arduino, dois motores de passos Nema 17 e outros componentes necessários para o seu funcionamento. A interação com o smartphone é realizada por um dispositivo de toque desenvolvido com um servo motor e uma caneta para tela *touch*. Esse sistema robótico aparentemente replica uma ação que foi realizada em um dispositivo. No artigo, não fica claro se existe alguma ferramenta que auxilia a criação do *script* ou se o desenvolvedor precisa passar as coordenadas manualmente. Embora esse sistema tenha um mecanismo para identificar se a ação foi executada corretamente, ele depende de ter coordenadas precisas para realizar a ação. Isso pode ser um problema se o aparelho acidentalmente mudar de posição. Além disso, o fato de realizar o toque na posição correta não garante que seja a ação que o testador queira. Por exemplo, ele pode tocar em outro aplicativo que está naquela posição porque o ícone mudou de posição.

Zhang *et al.* (ZHANG et al., 2020) apresentam uma técnica de detecção de GUIs isomórficas para implementar testes robóticos automatizados. O processo consiste em reconhecer os elementos da GUI usando visão computacional, construir uma estrutura para representá-los, extração dos vetores de recursos por meio de operações de convolução e agrupamento, e identificação das GUIs isomórficas por entropia relativa.

Pan *et al.* (PAN et al., 2020) utilizam um robô para interagir com o dispositivo móvel por meio de processamento de imagens para realizar operações de clique na tela. Eles usam tecnologia de projeção sem fio, ao invés de detecção por câmera, para projetar a tela do smartphone no computador. E realizam o reconhecimento dos elementos do GUI por meio de OCR. Os autores relatam que a utilização da projeção sem fio melhora a taxa de reconhecimento e economiza custos quando comparado com a tecnologia que utilizam câmeras.

7.3 CONSIDERAÇÕES FINAIS

Neste capítulo apresentamos duas categorias de trabalhos relacionados a ferramenta R-DLD. Entre os trabalhos que envolvem robôs, todos citam benefícios em automatizar os testes manuais, a maior parte deles utilizaram uma abordagem *Blackbox* não-invasiva para interagir com o smartphone semelhante à utilização de um usuário. Entre os principais benefícios citados se destaca a capacidade de repetir os movimentos com precisão e de realizar ações semelhantes

ao usuário. Nos trabalhos sobre perda de dados, somente Amalfitano utilizou uma abordagem *Blackbox* em um dispositivo real, mas os testes foram realizados de forma manual. Os outros trabalhos utilizaram AVD ao invés de um smartphone real e seguiram uma abordagem invasiva, porém automatizada.

O R-DLD agrega os benefícios de utilizar robô em smartphone real com a abordagem dos testes automatizados e possui uma solução menos invasiva do que os trabalhos citados. Pois uma parte dos comandos enviados ao smartphone (mudança de orientação) foram substituídos por interação com o robô. Esse trabalho é o único que utiliza rotações reais para detectar perda de dados. Em nossa avaliação empírica, o R-DLD revelou 341 bugs em 77 aplicativos analisados.

8 CONCLUSÃO E TRABALHOS FUTUROS

Nosso trabalho propôs uma ferramenta automatizada, acionada por robô, para realizar testes de perda de dados em um ambiente real. Tal proposta é mais realista e menos invasiva do que os trabalhos recentemente publicados (RICCIO; AMALFITANO; FASOLINO, 2018; RIGANELLI et al., 2020; GUO et al., 2022b). Mais realista porque as mudanças de orientações são acionadas pela rotação física do smartphone, interagindo com os sensores como em um cenário real. E menos invasiva, pois uma parte da interação (mudança de orientação) com o smartphone é realizada por um robô artesanal de baixo custo. Ou seja, são interações que não são enviadas por comandos ADBs via USB ao smartphone. Para avaliar nossa proposta, realizamos uma avaliação empírica com 77 aplicativos escolhidos aleatoriamente de uma loja Android e encontramos 341 problemas de perda de dados. Reportamos todos os *bugs* aos desenvolvedores e tivemos respostas em 58,94%, com o reconhecimento do problema em 89,55% dos casos. Além disso, propusemos novas classificações para causas de perda de dados e Alarmes falsos.

Como trabalho futuro, pretendemos investigar a viabilidade de propor uma abordagem totalmente não-invasiva: poderíamos substituir o oráculo de *screenshot* por técnicas de visão computacional para identificar diferenças entre as atividades após um evento de dupla rotação; já a interação com o smartphone seria executada com um robô cartesiano com uma caneta de toque para interagir com o dispositivo. Tudo isso controlado por técnicas de visão computacional para reconhecer elementos na tela do smartphone e traduzir em coordenadas. Uma alternativa seria utilizar um braço robótico segurando um smartphone com uma caneta de toque fixo em um suporte. Nesse cenário, o smartphone seria movimentado para executar as ações de toques na tela, permanecendo a caneta fixa. Essa solução possui a vantagem de utilizar apenas um robô para realizar os eventos de toque e a mudança de orientação. No entanto, o robô precisaria ser mais robusto e capaz de realizar movimentos mais complexos.

Pretendemos investigar outras formas de provocar um evento *stop-start* para detectar a perda de dados, como, por exemplo, alternar entre aplicações em condições de estresse associado a um aumento do processamento. Nessas condições, o sistema Android necessitaria de recursos e destruiria o processo do aplicativo para depois retomá-lo. Essa funcionalidade é particularmente útil quando existe o bloqueio da mudança de orientação no aplicativo. Assim, o tipo do evento *stop-start* poderia ser escolhido de acordo com as características da atividade: a mudança de orientação seria o evento principal, mas sob certas condições o R-DLD acionaria

outros eventos para forçar a destruição e a retomada da atividade.

Outro ponto que pretendemos melhorar é a estratégia de exploração das atividades. A solução atual não permite selecionar as atividades que queremos testar como, por exemplo, selecionar uma atividade específica, excluir aquelas com bloqueio de mudança de orientação, entre outras. Durante a seleção dos aplicativos, foi observado que muitos deles têm bloqueio de rotação em uma ou mais atividades. Esses aplicativos foram descartados pois a ferramenta precisa da mudança de orientação ativada para detectar a perda de dados.

Também é possível melhorar a forma de classificar os alertas em *Bugs reais* e Alarmes falsos por meio de técnicas de inteligência computacional seguindo o exemplo de trabalhos anteriores (MIRANDA et al., 2020; CABRAL et al., 2022). Uma possibilidade de trabalho futuro é a investigação da viabilidade de utilizar algoritmos de classificação binária para classificar os alertas gerados pelo R-DLD em *Bugs reais* ou Alarmes falsos.

REFERÊNCIAS

AMALFITANO, D.; FASOLINO, A. R.; TRAMONTANA, P.; CARMINE, S. D.; MEMON, A. M. Using gui ripping for automated testing of android applications. In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.: s.n.], 2012. p. 258–261.

AMALFITANO, D.; RICCIO, V.; PAIVA, A. C. R.; FASOLINO, A. R. Why does the orientation change mess up my android application? from gui failures to code faults. *Software Testing, Verification and Reliability*, v. 28, n. 1, p. e1654, 2018. E1654 stvr.1654. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1654>.

ANDROID. *The Activity Lifecycle*. 2022. [Online; accessed 12-February-2022]. Disponível em: <https://developer.android.com/guide/components/activities/activity-lifecycle>.

ANDROID. *Handle configuration changes*. 2022. [Online; accessed 12-February-2022]. Disponível em: <https://developer.android.com/guide/topics/resources/runtime-changes>.

ANDROIDSTUDIO. *API level distribution chart*. 2022. [Aplication; accessed 15-June-2022].

BANERJEE, D.; YU, K. Robotic arm-based face recognition software test automation. *IEEE Access*, v. 6, p. 37858–37868, 2018.

BANERJEE, D.; YU, K. Integrated test automation for evaluating a motion-based image capture system using a robotic arm. *IEEE Access*, v. 7, p. 1888–1896, 2019.

BANERJEE, D.; YU, K. 3d face authentication software test automation. *IEEE Access*, v. 8, p. 46546–46558, 2020.

BANERJEE, D.; YU, K.; AGGARWAL, G. Hand jitter reduction algorithm software test automation using robotic arm. *IEEE Access*, v. 6, p. 23582–23590, 2018.

BANERJEE, D.; YU, K.; AGGARWAL, G. Image rectification software test automation using a robotic arm. *IEEE Access*, v. 6, p. 34075–34085, 2018.

BANERJEE, D.; YU, K.; AGGARWAL, G. Object tracking test automation using a robotic arm. *IEEE Access*, v. 6, p. 56378–56394, 2018.

CABRAL, L.; MIRANDA, B.; LIMA, I.; D'AMORIM, M. Rvprio: A tool for prioritizing runtime verification violations. *Software Testing, Verification and Reliability*, Wiley Online Library, v. 32, n. 5, p. e1813, 2022. Disponível em: <https://doi.org/10.1002/stvr.1813>.

CRACIUNESCU, M.; MOCANU, S.; DOBRE, C.; DOBRESCU, R. Robot based automated testing procedure dedicated to mobile devices. In: *2018 25th International Conference on Systems, Signals and Image Processing (IWSSIP)*. [S.l.: s.n.], 2018. p. 1–4.

ESPRESSO. *API to test GUI*. 2022. [Online; accessed 15-June-2022]. Disponível em: <https://developer.android.com/training/testing/espresso>.

GUO, W.; DONG, Z.; SHEN, L.; TIAN, W.; SU, T.; PENG, X. Detecting and fixing data loss issues in android apps. In: *ISSTA 2022*. [S.l.: s.n.], 2022.

GUO, W.; DONG, Z.; SHEN, L.; TIAN, W.; SU, T.; PENG, X. A tool for detecting and fixing data loss issues in android apps. In: *ISSTA 2022*. [S.l.: s.n.], 2022.

HANS, M. *Appium Essentials*. [S.l.]: Packt Publishing, 2015. v. 1.

JUANG, J.-G.; CHENG, I.-H. Application of character recognition to robot control on smartphone test system. *Advances in Mechanical Engineering*, v. 9, n. 3, p. 1687814017693181, 2017.

LI, Y.; YANG, Z.; GUO, Y.; CHEN, X. Droidbot: a lightweight ui-guided test input generator for android. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. [S.l.: s.n.], 2017. p. 23–26.

MACIEL, L.; OLIVEIRA, A.; RODRIGUES, R.; SANTIAGO, W.; SILVA, A.; CARVALHO, G.; MIRANDA, B. A systematic mapping study on robotic testing of mobile devices. In: IEEE. *2022 48th Euromicro Conference Series on Software Engineering and Advanced Applications (SEAA)*. 2022. p. 475–482. Disponível em: <https://doi.org/10.1109/SEAA56994.2022.00079>.

MAO, K.; HARMAN, M.; JIA, Y. Robotic testing of mobile apps for truly black-box automation. *IEEE Software*, v. 34, n. 2, p. 11–16, 2017.

MCROBERTS, M. *Arduino Básico*. [S.l.]: Novatec, 2015. v. 2.

MIRANDA, B.; LIMA, I.; LEGUNSEN, O.; D'AMORIM, M. Prioritizing runtime verification violations. In: IEEE. *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 2020. p. 297–308. Disponível em: <https://doi.org/10.1109/ICST46399.2020.00038>.

MONK, S. *Programação com Arduino*. [S.l.]: Bookman, 2017. v. 2.

PAN, Z.; CHEN, J.; YAO, L.; CHEN, Z. Research on functional test of mobile app based on robot. In: *2020 IEEE 5th International Conference on Signal and Image Processing (ICSIP)*. [S.l.: s.n.], 2020. p. 960–964.

QIAN, J.; SHANG, Z.; YAN, S.; WANG, Y.; CHEN, L. Roscript: A visual script driven truly non-intrusive robotic testing system for touch screen applications. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020. (ICSE '20), p. 297–308. ISBN 9781450371216. Disponível em: <https://doi.org/10.1145/3377811.3380431>.

RICCIO, V.; AMALFITANO, D.; FASOLINO, A. R. Is this the lifecycle we really want? an automated black-box testing approach for android activities. In: *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. [S.l.: s.n.], 2018. p. 68–77.

RIGANELLI, O.; MOBILIO, M.; MICUCCI, D.; MARIANI, L. A benchmark of data loss bugs for android apps. In: IEEE. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. [S.l.], 2019. p. 582–586.

RIGANELLI, O.; MOTTADELLI, S. P.; ROTA, C.; MICUCCI, D.; MARIANI, L. Data loss detector: automatically revealing data loss bugs in android apps. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. [S.l.: s.n.], 2020. p. 141–152.

ROBOTIUM. *Android test automation framework.* 2022. [Online; accessed 15-June-2022]. Disponível em: <https://developer.android.com/training/testing/other-components/ui-automator>.

UIAUTOMATOR. *UI testing framework*. 2022. [Online; accessed 15-June-2022]. Disponível em: <https://developer.android.com/training/testing/other-components/ui-automator>.

VERMAA, P.; CHAUHANA, D. S.; RAMASWAMYA, R.; KUMAR, C. L. Multitouch testing robot. *International Journal of Control Theory and Applications*, v. 10, n. 31, p. 219–223, 2017. ISSN 0974-5572.

WEI, L.; LIU, Y.; CHEUNG, S.-C. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2016. p. 226–237.

ZAEEM, R. N.; PRASAD, M. R.; KHURSHID, S. Automated generation of oracles for testing user-interaction features of mobile apps. In: IEEE. *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. [S.l.], 2014. p. 183–192.

ZHANG, T.; LIU, Y.; GAO, J.; GAO, L. P.; CHENG, J. Deep learning-based mobile application isomorphic gui identification for automated robotic testing. *IEEE Software*, v. 37, n. 4, p. 67–74, 2020.