Este é o ciclo de vida que realmente queremos? Uma abordagem automatizada de teste de caixa preta para atividades do Android

Vincenzo Riccio
Universidade de Nápoles Federico II
Nápoles, Itália
vincenzo.riccio@unina.it

Domenico Amalfitano
Universidade de Nápoles Federico II
Nápoles, Itália
domenico.amalfitano@unina.it

Anna Rita Fasolino
Universidade de Nápoles Federico II
Nápoles, Itália
anna.fasolino@unina.it

RESUMO

O Android é hoje o sistema operacional móvel mais popular do mundo e a demanda por qualidade para aplicativos móveis Android cresceu junto com sua disseminação. O teste é uma abordagem bem conhecida para garantir a qualidade dos aplicativos de software, mas os aplicativos Android têm várias peculiaridades em comparação com os aplicativos de software tradicionais que devem ser levados em consideração pelos testadores. Vários estudos apontaram que os aplicativos móveis sofrem de problemas que podem ser atribuídos ao manuseio incorreto do ciclo de vida da atividade, por exemplo, travamentos, travamentos, desperdício de recursos do sistema. Portanto, o ciclo de vida das Atividades que compõem um aplicativo deve ser devidamente considerado por meio de abordagens de teste. Neste artigo, propomos o ALARic, uma técnica de teste totalmente automatizada baseada em eventos de caixa preta que explora um aplicativo em teste para detectar problemas vinculados ao ciclo de vida da atividade do Android. ALARic foi implementado em uma ferramenta. Conduzimos um experimento envolvendo 15 aplicativos Android reais que mostraram a eficácia do ALARic em encontrar falhas de GUI e travamentos vinculados ao ciclo de vida da atividade. No estudo, o ALARic provou ser mais eficaz na detecção de falhas do que o Monkey, a ferramenta de teste automatizada do Android de última geração.

Formato de Referência

ACM: Vincenzo Riccio, Domenico Amalfitano e Anna Rita Fasolino. 2018. Este é o ciclo de vida que realmente queremos? Uma abordagem automatizada de teste de caixa preta para atividades Android. Em (ISSTA Companion/ECOOP Companion'18), 16 a 21 de julho de 2018, Amsterdã, Holanda. ACM, Nova York, NY, EUA, 10 páginas. https://doi.org/10.1145/3236454.3236504

1. INTRODUÇÃO

O número de usuários de tecnologia móvel e smartphones está crescendo constantemente e deve ultrapassar 2,5 bilhões em 2019 [18]. Hoje, o Android é o sistema operacional móvel mais popular do mundo [19]. Mais e mais pessoas em todo o mundo confiam em aplicativos de software móvel (apps) para realizar várias tarefas diárias. Assim, a demanda por qualidade para aplicativos móveis tem crescido junto com sua disseminação. Como consequência, os desenvolvedores móveis devem considerar adequadamente a qualidade de seus aplicativos, adotando técnicas de garantia de qualidade adequadas, como testes. De várias técnicas e ferramentas estão atualmente disponíveis para testar um aplicativo Android antes de ser publicado no mercado [3]. As ferramentas de automação de teste podem facilitar as atividades de teste de software, pois salvam os humanos de tarefas manuais rotineiras, demoradas e propensas a erros [7].

Os aplicativos móveis possuem várias peculiaridades em relação aos aplicativos de software tradicionais que devem ser levados em consideração por meio de técnicas e ferramentas de teste [16]. Em particular, o pequeno tamanho dos dispositivos móveis introduziu a necessidade de ter na tela um único aplicativo focado por vez. No Android um aplicativo é composto por uma ou mais Atividades; cada Activity representa uma única GUI que permite ao usuário interagir com o aplicativo.

O Android Framework define um ciclo de vida peculiar para instâncias de Activity a fim de gerenciá-las de forma transparente para o usuário que pode navegar por um aplicativo e alternar entre aplicativos sem perder seu progresso e dados; ao mesmo tempo, permite não desperdiçar os recursos limitados de um dispositivo móvel, como memória e bateria. O Guia do desenvolvedor Android oficial1 enfatiza a relevância do recurso Ciclo de vida da atividade e alerta os desenvolvedores sobre as ameaças que ele apresenta em várias seções; portanto, fornece recomendações e diretrizes para ajudar os programadores no manuseio correto do ciclo de vida da Activity.

Apesar disso, vários trabalhos na literatura apontaram que aplicativos móveis, incluindo os de força industrial, sofrem de problemas que podem ser atribuídos ao manuseio incorreto do ciclo de vida da atividade [4] [15] [17] [5]. Zein et ai. [22] realizaram um estudo de mapeamento sistemático de técnicas de teste de aplicativos móveis envolvendo 79 artigos e identificaram possíveis áreas que requerem mais pesquisas. Entre eles, eles enfatizaram a necessidade de técnicas de teste específicas visando a conformidade do ciclo de vida da atividade.

Algumas soluções têm sido apresentadas na literatura para resolver este problema. Uma parte deles propôs técnicas de teste que se baseiam em artefatos de teste existentes [1, 8] ou modelos GUI [21] para gerar automaticamente casos de teste capazes de exercitar adequadamente o ciclo de vida da Activity. Outro trabalho [17] detecta, por meio de análise estática, bugs que podem causar um estado corrompido quando um aplicativo é pausado, interrompido ou encerrado. Sua solução também pode gerar casos de teste automaticamente para reproduzir bugs, mas precisa modificar o código do aplicativo para verificar os problemas detectados estaticamente.

Outro grupo de abordagens aproveita a análise dinâmica para encontrar problemas vinculados ao ciclo de vida da atividade [9, 15]. Essas técnicas dinâmicas concentram-se principalmente em encontrar um tipo específico de falha, como travamentos [9, 15] ou vazamentos de recursos [10]. No entanto, nenhum deles abordou falhas de GUI que consistem na manifestação de um unex estado da GUI esperado. Como apontado em [5], as falhas de GUI vinculadas ao ciclo de vida da atividade representam uma categoria generalizada de problemas no Android

ISSTA Companion/ECOOP Companion'18, 16 a 21 de julho de 2018, Amsterdã, Holanda
ACM ISBN 978-1-4503-5939-9/18/07. . . $ 15,00
https://doi.org/10.1145/3236454.3236490

1https://developer.android.com/

aplicativos e há a necessidade de definir técnicas de teste eficazes para detectá-los.

Para superar essas limitações, neste artigo, propomos o ALARic (Activity Lifecycle Android Ripper), uma técnica de teste dinâmico baseada em eventos de caixa preta totalmente automatizada .

O ALARic é capaz de detectar falhas de GUI e travamentos de aplicativos relacionados ao ciclo de vida das atividades de um aplicativo, testando sistematicamente cada estado de GUI de atividade encontrado durante a exploração automática do aplicativo. Para isso, aproveita eventos específicos para dispositivos móveis capazes de exercitar o ciclo de vida da Activity e oráculos de teste especificamente projetados. Nossa solução não requer nenhum conhecimento prévio do aplicativo em teste, modificação do aplicativo ou intervenção manual. A eficácia do ALARic na detecção de problemas vinculados ao ciclo de vida da atividade foi demonstrada em um experimento envolvendo 15 aplicativos Android reais. O experimento também mostrou que o ALARic foi mais eficaz na detecção de falhas vinculadas ao ciclo de vida da atividade do que a ferramenta de teste automatizada do Android de última geração, ou seja, Monkey, a ferramenta mais usada dessa categoria em ambientes industriais.

O artigo melhora a literatura sobre testes automatizados de GUI com as seguintes contribuições:

- uma nova técnica de teste automatizado de GUI para detectar falhas e travamentos de GUI vinculados ao ciclo de vida da atividade do Android. Em particular, o ALARic é a primeira técnica de teste dinâmico capaz de resolver o problema de falhas na GUI;
- um experimento envolvendo aplicativos Android reais mostrando o validade da técnica proposta.

O resto do artigo está estruturado da seguinte forma. A Seção 2 descreve os antecedentes e a Seção 3 fornece uma visão geral da abordagem de teste proposta. A Seção 4 apresenta detalhes de projeto e implementação da ferramenta que desenvolvemos enquanto na Seção 5 descrevemos o experimento que foi realizado. A Seção 6 fornece trabalhos relacionados. Por fim, a Seção 7 traz as conclusões e apresenta trabalhos futuros.

2. ANTECEDENTES

2.1 Ciclo de vida da atividade As atividades são os blocos de construção essenciais dos aplicativos Android; uma Activity pode ser vista como uma única GUI através da qual os usuários podem acessar os recursos oferecidos pelo aplicativo. Uma Activity é implementada como uma subclasse da classe Activity , definida no Android Framework.

As instâncias de atividade exercidas pelo usuário são gerenciadas como uma pilha de atividades pelo sistema operacional Android. Um usuário geralmente navega, sai e volta para um aplicativo, mas apenas a atividade no topo da pilha está ativa no primeiro plano da tela. Para garantir uma transição suave entre as telas, as outras atividades são mantidas na pilha. Isso permite que o usuário navegue para uma atividade exercida anteriormente sem perder seu progresso e informações. Além disso, o sistema pode decidir se livrar de uma atividade em segundo plano para liberar aumentar o espaço de memória.

Para fornecer essa rica experiência ao usuário, as Atividades do Android têm um ciclo de vida adequado, passando por diferentes estados. A Figura 1 mostra o ciclo de vida da atividade conforme ilustrado no Android Developer Guide2 oficial . Os retângulos arredondados representam todos os estados uma Activity pode estar em; as bordas são rotuladas com os métodos de retorno de chamada que são invocados pela plataforma Android quando uma Activity transita entre os estados.

A estrutura do Android fornece sete métodos de retorno de chamada que são invocados automaticamente à medida que uma Activity transita para um novo estado. Eles podem ser substituídos pelo desenvolvedor para permitir que o aplicativo execute um trabalho específico sempre que uma determinada alteração do estado da atividade for acionada.

A Atividade visível no primeiro plano da tela e interagindo com o usuário está no estado Retomado , ou é criada pela primeira vez ou retomada a partir dos estados Pausado ou Parado .

Quando uma atividade perdeu o foco, mas ainda está visível (por exemplo, uma caixa de diálogo modal do sistema tem o foco no topo da atividade), ela está no estado Pausado ; nesse estado, o aplicativo geralmente mantém todo o progresso e informações do usuário. Quando o usuário navega para uma nova Activity, a anterior é colocada no estado Stopped ; ele ainda retém todas as informações do usuário, mas não é mais visível para o usuário. No entanto, quando uma atividade está nos estados Pausado ou Parado , o sistema pode eliminá-la da memória se os recursos do sistema forem necessários para outros aplicativos e , portanto, a atividade transitar para o estado Destruído . Quando ele é exibido novamente para o usuário, ele é reiniciado e seu estado salvo deve ser restaurado.

A Figura 1 também destaca os três ciclos principais do ciclo de vida da atividade. A seguir, de acordo com o Android Developer Guide, chamamos esses loops de Entire Loop, Visible Loop e Foreground Loop, respectivamente, e relatamos sequências de eventos para exercitar cada um deles: (1) O Entire Loop (EL) de uma Activity consiste na retomada Sequência de estados pausada-parada-destruída-criada-iniciada-reiniciada . Este loop pode ser exercido por eventos que causam uma mudança de configuração, por exemplo, uma mudança de orientação da tela, que destrói a instância de Activity e a recria de acordo com a nova configuração3 ; (2) O Loop Visível (VL) corresponde à sequência de estados Retomado-Pausado Parado-Iniciado-Retomado durante o qual a Atividade é ocultada e depois tornada visível novamente. Existem várias sequências de eventos capazes de parar e reiniciar uma Activity, por exemplo, desligar e ligar a tela ou colocar o aplicativo em segundo plano e depois em primeiro plano novamente através dos botões Visão geral ou Início; (3) O Foreground Loop (FL) de uma Activity envolve a sequência de estado Resumed Paused-Resumed. A transição Retomada Pausada pode ser acionada abrindo elementos não de tamanho normal , como caixas de diálogo modais ou atividades semitransparentes que ocupam o primeiro plano enquanto a Atividade ainda está visível em segundo plano. Para acionar a transição Paused-Resumed o usuário deve descartar este elemento.

2.2 Problemas vinculados ao ciclo de vida da atividade Os desenvolvedores de aplicativos Android devem implementar corretamente as atividades, levando em consideração seu ciclo de vida. Isso garante que o aplicativo funcione da maneira que os usuários esperam e não exiba comportamentos aberrantes à medida que transita por diferentes estados do ciclo de vida em tempo de execução. Uma boa implementação dos retornos de chamada do ciclo de vida e o conhecimento dos recursos do Android Framework podem ajudar o programador

2https://developer.android.com/reference/android/app/Activity.html

3https://developer.android.com/guide/components/activities/state-changes.html

Este é o ciclo de vida que realmente queremos?

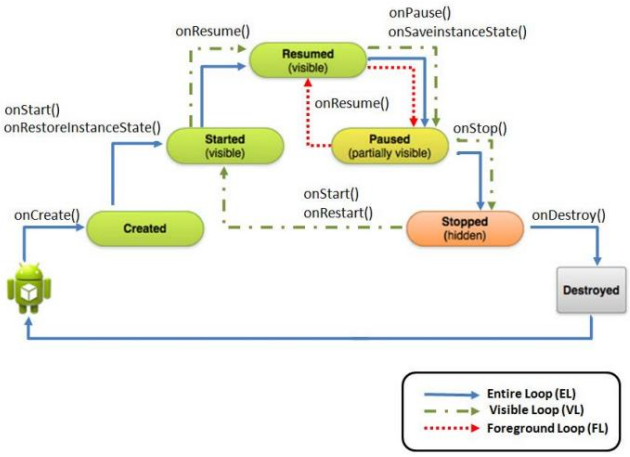
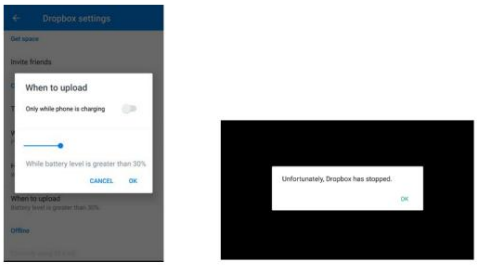


Figura 1: os loops de chave do ciclo de vida da atividade do Android



(a) Antes da aplicação da mudança de orientação

(b) Após a aplicação da mudança de orientação

Figura 2: Falha exposta pelo aplicativo Dropbox

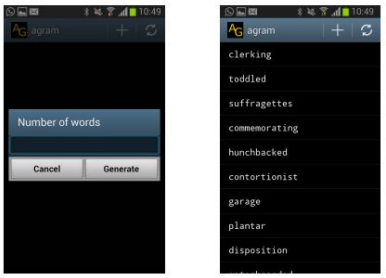
aplicativos que se comportam conforme o esperado e evitam vários problemas, como travamentos e falhas na GUI.

2.2.1 Falhas. Uma falha ocorre quando um aplicativo para de funcionar corretamente e fecha inesperadamente. Quando um aplicativo falha, o Android encerra seu processo e exibe uma caixa de diálogo para informar ao usuário que o aplicativo foi interrompido4 . É um comportamento de aplicativo muito indesejável e, de fato, o mais flagrante. Estamos interessados em travamentos vinculados ao ciclo de vida da atividade, ou seja, travamentos desencadeados por eventos que exercem o ciclo de vida da atividade. Relatamos um exemplo de falha que detectamos no serviço popular de hospedagem de arquivos Dropbor versão 27.1.25 . Se o usuário selecionar o terceiro item das configurações de uploads da câmera, uma caixa de diálogo modal será exibida (consulte a Figura 2(a)). Quando o usuário altera a orientação do dispositivo, o aplicativo para de funcionar repentinamente, conforme mostrado na Figura 2(b).

2.2.2 Falhas da GUI. As falhas da GUI são uma classe relevante de falhas que podem atrapalhar a experiência do usuário e consistem na manifestação de um estado inesperado da GUI [11]. Nós nos concentramos nas falhas da GUI acionadas pelo exercício dos três loops de chave do ciclo de vida da atividade. Dentro

4https://developer.android.com/topic/performance/vitals/crash.html 5https://dropbox.zendesk.com, ID de solicitação de suporte do Dropbox nº 5199918

o aplicativo cliente Android oferecido pelo



(a) Antes da aplicação da mudança de orientação dupla

(b) Após a aplicação da mudança de orientação dupla

Figura 3: Falha na GUI exposta pelo aplicativo Agram

Em particular, pode haver uma falha da GUI vinculada ao ciclo de vida da atividade quando o estado da GUI antes da atividade ser interrompida, pausada ou destruída é diferente do estado da GUI exibido após o usuário retornar à atividade [1, 5, 14, 17, 21]. As falhas da GUI se manifestam de várias maneiras, ou seja, objetos inesperados da GUI podem aparecer em posições erradas, objetos podem ser renderizados com propriedades erradas ou objetos importantes podem desaparecer da GUI [5]. Relatamos um exemplo de falha de GUI que encontramos na versão 2.7.3 do QKSMS, um aplicativo Android que exibe anagramas em inglês e que está disponível gratuitamente na Google Play Store. Se o usuário optar por criar palavras aleatórias, uma caixa de diálogo modal aparecerá solicitando o número de palavras que o usuário deseja gerar (consulte a Figura 3(a)). Quando os usuários alteram a orientação do dispositivo duas vezes, eles naturalmente esperam que o estado da GUI permaneça o mesmo. Em vez disso, o aplicativo exibirá um estado de GUI inesperado quando a atividade for destruída e depois recriada devido à alteração de configuração, ou seja, a caixa de diálogo desaparece e uma lista de palavras aleatórias é renderizada na tela, conforme mostrado em

Figura 3(b).

3 A ABORDAGEM ALARICA

Nesta seção apresentamos a abordagem adotada pela ALARic no teste de aplicativos Android.

ALARic implementa uma técnica de teste online totalmente automatizada, pois explora a aplicação em teste (AUT) e ao mesmo tempo detecta comportamentos aberrantes vinculados ao ciclo de vida da atividade, ou seja, Falhas e falhas de GUI. O ALARic exercita o AUT através de sequências de eventos de entrada, sendo sistemas de software orientados a eventos de aplicativos Android [6].

A estratégia de exploração adotada pelo ALARic envia eventos de entrada aleatórios para o AUT e executa sistematicamente uma sequência de eventos de entrada capaz de exercitar um dos três principais ciclos de vida da atividade cada vez que uma nova GUI é encontrada pela primeira vez durante a exploração do aplicativo. Definimos Lifecycle Event Sequence uma sequência de eventos capaz de acionar um dos principais loops do ciclo de vida da Activity.
Após o exercício do ciclo de vida da atividade, o ALARic avalia se o aplicativo expõe algum problema relacionado ao ciclo de vida da atividade.

Para exercitar os três ciclos principais do ciclo de vida, aproveitamos três sequências de eventos do ciclo de vida, ou seja, a mudança de orientação dupla (DOC), o primeiro plano de fundo (BF) e a atividade semitransparente

Sequências de eventos de intenção (STAI). Escolhemos essas sequências de eventos de ciclo de vida, pois cada uma delas é capaz de exercer um ciclo de vida diferente . Para detectar falhas na GUI, escolhemos as Sequências de Eventos do Ciclo de Vida para as quais o estado da GUI da Activity deve ser retido após suas execuções [21].

A sequência de eventos Double Orientation Change (DOC) exercita duas vezes o loop EL e consiste em uma sequência de dois eventos consecutivos de mudança de orientação. Usamos a sequência de eventos DOC, pois aplicar uma única alteração de orientação pode não ser suficiente para detectar falhas na GUI, pois algumas pequenas diferenças no conteúdo ou visualizações da GUI são realmente aceitáveis entre as orientações de paisagem e retrato, e o estado da GUI do aplicativo pode diferir após um único evento de mudança de orientação . Após uma segunda mudança de orientação consecutiva, o conteúdo e o layout da GUI devem ser os mesmos de antes da primeira mudança de orientação [5].

A sequência Background Foreground (BF) coloca o aplicativo em segundo plano através do toque no botão Home e, em seguida, empurra o aplicativo novamente em primeiro plano. Exercita o loop VL.

No que diz respeito ao loop FL, ele é exercido pela sequência de eventos Semi-Transparent Activity Intent (STAI). Consiste em iniciar uma Activity semitransparente que pausa a Activity atual em primeiro plano e depois retornar a ela tocando no botão Voltar.

A abordagem ALARic é configurável e permite que o testador configure um tipo de Sequência de eventos de ciclo de vida a ser aplicado para exercitar o ciclo de vida da atividade em todos os estados da GUI expostos por uma atividade que são encontrados durante a exploração do aplicativo.

A Figura 4 mostra um exemplo real de como o ALARic funciona. Neste exemplo, usamos o DOC Lifecycle Event Sequence para testar o aplicativo Amaze versão 3.1.2 RC4. Os instantâneos representam os estados da GUI encontrados durante a exploração automática. As bordas vermelhas representam as Sequências de Eventos do Ciclo de Vida, enquanto as bordas pretas são eventos planejados aleatórios. A cada iteração de exploração, o ALARic descreve o estado atual da GUI e verifica se já foi explorado antes durante a exploração. Os estados da GUI encontrados pela primeira vez, ou seja, A, C, E, H, L, são exercidos por um DOC. Considerando que, os estados GUI já encontrados, ou seja, D, F, G, J, K, são exercidos por eventos planejados aleatórios. A ferramenta compara os estados da GUI antes e depois do evento DOC e verifica se eles não são diferentes.

O ALARic encontrou 3 falhas de GUI nesta exploração, ou seja, após a 3ª, 5ª e 9ª iteração. Além disso, o aplicativo travou após o acionamento do 12º evento. Quando ocorre uma falha, o ALARic inicia o aplicativo do zero. A exploração termina após o acionamento de um número predefinido de eventos ou após um determinado tempo de teste.

4 A FERRAMENTA ALARic

A abordagem ALARic foi implementada na Ferramenta ALARic6 . A Fig. 5 mostra a arquitetura da ferramenta que é composta por dois componentes , ou seja, o ALARic Engine e o Test Executor.

O componente ALARic Engine é responsável por implementar a lógica de negócios da abordagem de teste. Ele analisa a GUI atualmente renderizada pelo AUT, planeja a próxima sequência de eventos de entrada a ser disparada e verifica a presença de falhas. Ele não interage diretamente com o AUT, mas delega essa tarefa ao componente Test Executor que é capaz de disparar sequências de eventos de entrada no AUT e para buscar a descrição da GUI atual em termos de seus widgets de composição e seus valores de atributo.

A ferramenta recebe como entrada um arquivo de configuração que é necessário para configurar o processo de teste. Nesse arquivo, o testador especifica a Sequência de Eventos do Ciclo de Vida a ser acionada e a condição de encerramento. Quanto à condição de encerramento, é possível definir um tempo máximo de execução ou o número de sequências de eventos de entrada a serem acionadas. O ALARic Engine busca o AUT explorando seu .apk ou seu código-fonte e o instala no Test Executor.

Durante a exploração automática do aplicativo, o ALARic salva as descrições das GUIs encontradas. Em cada iteração de exploração, ele carrega essas descrições para compará-las com a GUI atualmente renderizada na tela. Dessa forma, ele testa apenas os estados da GUI encontrados pela primeira vez.

A ferramenta produz um arquivo de relatório sobre as falhas detectadas e falhas de GUI. O relatório contém para cada falha da GUI: (1) o nome do aplicativo , (2) o nome da atividade em que a falha foi detectada, (3) a sequência de eventos que levaram à falha e (4) o tipo de sequência de eventos do ciclo de vida executado. Além disso, para as falhas da GUI, também contém a descrição e a captura de tela dos estados da GUI antes e depois da aplicação da Sequência de Eventos do Ciclo de Vida. Quanto aos travamentos, ele contém o tipo de exceção não tratada e seu rastreamento de pilha.

4.1 ALARic Engine

O processo de teste online implementado pela ferramenta ALARic é descrito pelo diagrama de atividades UML mostrado no componente ALARic Engine da Figura 5. Ele estende o algoritmo genérico de teste online apresentado no framework proposto por Amalfitano et al [2]. As etapas pertencentes ao algoritmo original são relatadas em branco, enquanto as introduzidas por nossa abordagem são coloridas em cinza.

O ALARic Engine executa um processo iterativo de exploração automática de GUI, onde etapas sequenciais são executadas até que uma determinada condição de término seja alcançada.

Na etapa Descrever a GUI atual, uma descrição do estado atual da GUI é inferida, de acordo com um critério de abstração da descrição da GUI . A descrição de um estado de GUI inclui os pares (atributo, valor) assumidos por seus componentes em tempo de execução. A descrição do estado atual da GUI é comparada com as encontradas anteriormente para avaliar se ele nunca foi encontrado antes durante a exploração. As etapas Exercise Activity Lifecycle e Evaluate Oracles são executadas quando um novo estado de GUI é encontrado pela primeira vez. Caso contrário, as etapas Planejar Eventos e Executar Eventos são executadas.

Na etapa Exercise Activity Lifecycle, uma Sequência de Eventos de Ciclo de Vida predefinida é acionada. A etapa Avaliar Oráculos permite a verificação de oráculos especificamente criados para detectar a presença de tipos específicos de problemas do ciclo de vida da atividade. A implementação atual do ALARic é capaz de avaliar dois oráculos, ou seja, falhas e travamentos da GUI. Quanto às falhas da GUI, assim como [1, 5, 14, 17, 21], a ferramenta é capaz de reconhecer falhas que ocorrem quando os estados da GUI antes e depois da aplicação de uma Sequência de Eventos do Ciclo de Vida são diferentes. Quanto às falhas, o ALARic verifica se uma exceção não tratada ocorre após a execução de uma Sequência de Eventos do Ciclo de Vida.

A etapa Planejar eventos planeja as sequências de eventos de entrada que serão acionadas na GUI atual de acordo com um agendamento aleatório uniforme

6A ferramenta ALARic está disponível para download neste link https://goo.gl/ypTMVs.

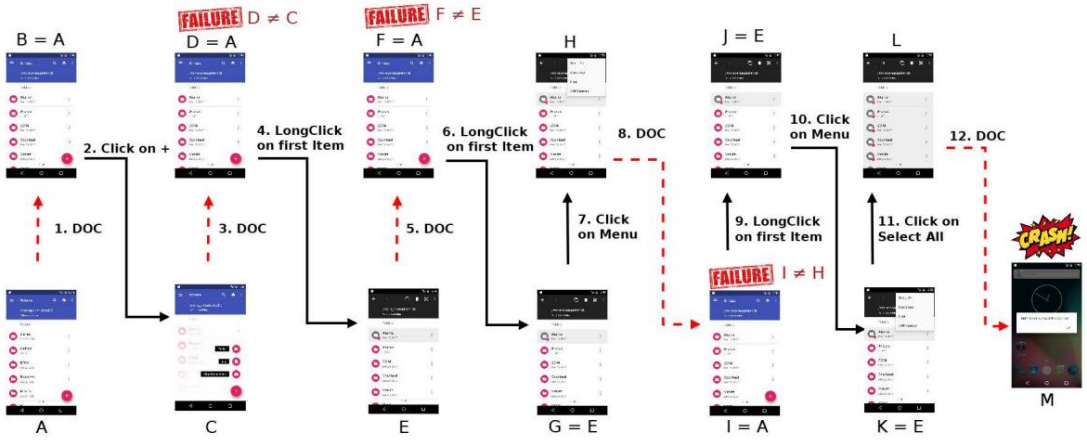


Figura 4: Exemplo de teste ALARic

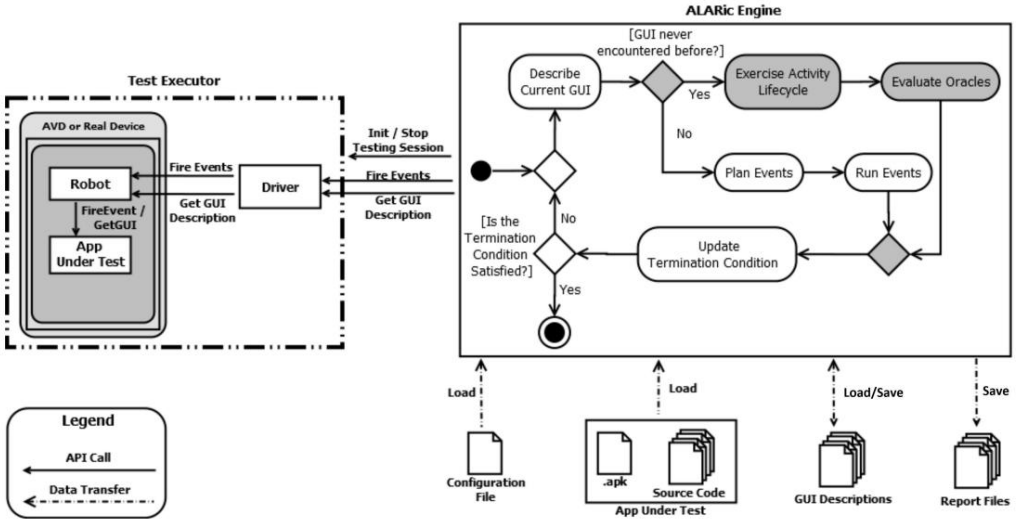


Figura 5: A arquitetura da ferramenta ALARic

estratégia. Na etapa Executar eventos, a sequência de eventos de entrada planejada é realmente executada.

Por fim, na etapa Atualizar Condição de Encerramento, avalia- se se foi atingido um tempo máximo de teste ou um número máximo de eventos de entrada disparados, definidos pelo testador.

O Engine requer as APIs REST fornecidas pelo componente Test Executor para realizar suas atividades. Ele chama as APIs Init Testing Session e Stop Testing Session no início e no final do processo de teste para instalar e desinstalar o aplicativo em teste, respectivamente. A API Fire Events é usada para acionar eventos, enquanto a Get GUI Description é explorada para recuperar a descrição da GUI atual.

4.2 Executor de Teste

O componente Test Executor é responsável por executar as atividades de teste do ALARic Engine no AUT. É capaz de interagir com um dispositivo físico e um dispositivo virtual Android (AVD)7 . É composto por dois componentes, ou seja, Robot e Driver que interagem através da tecnologia Java socket.

O componente Driver é responsável por desacoplar a lógica de negócios implementada no ALARic Engine do dispositivo onde o AUT está instalado. O Robô deve ser executado no mesmo dispositivo onde o AUT está instalado e interage com ele disparando eventos e descrevendo as GUIs renderizadas em tempo de execução. Este componente explora as APIs fornecidas pela biblioteca Robotium8 o Android Debug Bridge (ADB)9 .

5 AVALIAÇÃO EXPERIMENTAL

Nesta seção, relatamos o estudo que realizamos para investigar a capacidade da ferramenta ALARic em detectar problemas vinculados à Activity

7https://developer.android.com/studio/run/emulator.html
8https://github.com/RobotiumTech/robotium 9https://developer.android.com/studio/command-line/adb.html

ciclo da vida. Consideramos como vinculados ao ciclo de vida da Activity os problemas que são expostos por Sequências de Eventos de Ciclo de Vida. O estudo destinado a respondendo as seguintes perguntas de pesquisa:

RQ1 Qual é a eficácia da ferramenta ALARic na detecção de problemas relacionados a o ciclo de vida da atividade em aplicativos Android reais?

RQ2 Como a eficácia da ferramenta ALARic na detecção problemas vinculados ao ciclo de vida da atividade em aplicativos Android reais comparar com o estado da prática?

Algumas ferramentas têm sido propostas na literatura que exploram dinâmicas análise e pode encontrar falhas vinculadas ao ciclo de vida da atividade [9, 15]. No entanto, não conseguimos comparar a ferramenta ALARic com eles, pois eles não estão mais disponíveis ou não são suportados mais e não conseguem segmentar o SO Android e o SDK mais recentes versões. Portanto, consideramos a ferramenta Monkey10 por ser considerado como o estado da prática atual para Android automatizado teste de aplicativos [12, 13], sendo a ferramenta mais utilizada desta categoria em ambientes industriais [23, 24]. É usado principalmente em robustez processos de teste para revelar falhas de aplicativos. Além disso, não poderíamos comparar a eficácia do ALARic em encontrar falhas na GUI com outras técnicas dinâmicas existentes já que nosso trabalho é o primeiro abordar esta questão.

5.1 Objetos

Como objetos da avaliação, selecionamos 15 aplicativos que são distribuídos pela loja oficial de aplicativos do Google11 cujo código-fonte está disponível em o repositório F-Droid12. Desta forma, selecionamos aplicativos que foram representante dos aplicativos típicos disponíveis para usuários do Android. O disponibilidade do código fonte nos permitiu analisar melhor a falhas detectadas. Escolhemos o F-Droid por ser um repositório bem conhecido de aplicativos de software livre e de código aberto (FOSS) para o plataforma Android que tem sido amplamente utilizada em outros estudos sobre Testes Android propostos na literatura [7, 12–15]. Relatórios da Tabela 1 para cada aplicativo selecionado seu nome, a versão que consideramos e seu Tamanho.

Tabela 1: Aplicativos de objeto

Nome do	Tamanho do Apk da Versão (kB)	
aplicativo A1 A Time Tracker	0,21 115	
A2 Port Knocker A3 Quem	1.0.9 2.200	
tem minhas coisas?	1.0.27 104,3	
A4 Agram A5	1.4.1 723	
Alarm Klock A6 Padland	1,9 640	
A7 Syncthing A8 Anecdote	1,3 2.000	
A9 Amaze File Manager	0,9,1 19.300	
3.1.2 RC4 A10 Google	1.1.2 1.800	
Authenticator A11 BeeCount		5.900
	2.21	708
	2.3.9	3.200
Companheiro A12 FOSDEM	1.4.6	1.300
A13 Periódico	0,30	925
Barra de tarefas A14	3.0.2	1.600
A15 SpaRSS	1.11.8	1.400

10https://developer.android.com/studio/test/monkey.html
11https://play.google.com/store/apps
12 https://f-droid.org/

5.2 Métricas

Para avaliar a eficácia do ALARic na detecção de falhas na GUI, consideramos a cardinalidade dos seguintes conjuntos:

- DGFDOC: falhas de GUI distintas acionadas pelo evento DOC seqüência
- DGFBF : falhas de GUI distintas acionadas pelo evento BF se seguir
- DGFST AI : falhas de GUI distintas acionadas pelo evento STAI seqüência
- DGFT ot al : falhas de GUI distintas vinculadas ao ciclo de vida da atividade acionado por sequências de eventos DOC, BF ou STAI

Analogamente, para avaliar a eficácia das ferramentas em encontrar travamentos, consideramos o número de elementos de cada um dos seguintes conjuntos:

- DCDOC: Crashes distintos acionados pelo evento DOC se seguir
- DCBF : Crashes distintos acionados pela sequência de eventos BF
- DCST AI : Crashes distintos acionados pelo evento STAI se seguir
- DCT ot al : Crashes distintos vinculados ao ciclo de vida da atividade acionados por sequências de eventos DOC, BF ou STAI

Como o mesmo problema pode ser exposto várias vezes durante um processo de teste, decidimos contar apenas as ocorrências de questões. Fizemos estas suposições: (1) as falhas da GUI são distintas se eles envolveram estados iniciais não equivalentes ou finais não equivalentes estados [5], e (2) as falhas são distintas se não se referirem ao mesmo tipo de exceção não tratada ou não tem a mesma pilha traço [9, 15].

5.3 Procedimento Experimental

O procedimento experimental que seguimos foi organizado em duas etapas, a saber (1) Teste de Aplicativo e (2) Coleta e Validação de Dados.

5.3.1 Teste de aplicativos. esta etapa foi realizada em duas fases. Dentro Na primeira fase, três processos de teste diferentes foram executados. Em cada processo, todos os objetos foram testados por três execuções do Ferramenta ALARic configurada para planejar e executar apenas um ciclo de vida Tipo de sequência de eventos, ou seja, DOC, BF e STAI. Fizemos três corridas para cada configuração para mitigar o não determinismo dos aplicativos e das técnicas de exploração aleatória [7]. Cada corrida durou uma hora. Um total de nove execuções de teste de uma hora para cada aplicativo foram realizadas.

Na última fase, testamos os aplicativos de objetos usando o Monkey ferramenta, a fim de comparar a eficácia das ferramentas. Macaco é um ferramenta de teste automatizado para aplicativos Android, pertencentes ao Android SDK. Esta ferramenta adota uma abordagem de teste aleatório, que envia um fluxo aleatório de eventos de nível de sistema e interface do usuário para o aplicativo em teste.

Realizamos um processo de teste onde nove Monkeys de uma hora execuções de teste foram executadas para cada aplicativo. Executamos nove macacos funciona como o ALARic para garantir uma comparação justa entre as ferramentas.

Nesta fase, definimos o nível máximo de verbosidade do Monkey ferramenta para produzir uma saída mais precisa e rica contendo informações sobre os eventos propagados e as falhas detectadas.

Todos os processos de teste foram realizados na mesma infraestrutura de teste que consiste em um PC desktop com um processador Intel(R) Processador Core(TM) i7 4790@3.60GHz e 8 GB de RAM, rodando

um Nexus 5 AVD padrão com Android 6 (API 23). O PC host foi equipado com o sistema operacional Ubuntu, versão 16.04. Para garantir que cada execução fosse executada nas mesmas condições, todas as execuções foram executadas em AVDs criados do zero.

5.3.2 Coleta e Validação de Dados. No final dos processos de teste reunimos todos os relatórios produzidos pelas ferramentas consideradas e recrutamos uma equipa composta por dois doutores. estudantes e um Pesquisador de Pós-Doutorado com conhecimento em Depuração de Software e Testes Android.

A equipe foi solicitada a analisar as falhas expostas pelas ferramentas e validá-las. Para tanto, a equipe examinou os relatórios produzidos pelas ferramentas consideradas para identificar as falhas distintas expostas por cada ferramenta. Ao contrário do ALARic, o Monkey não oferece uma descrição detalhada de cada falha exposta em tempo de execução [15]. Para extrair as falhas detectadas pelo Monkey da saída gerada , a equipe inspecionou manualmente esses arquivos para encontrar as instâncias de rastreamento de pilha de exceção e os eventos que levaram a elas. Em seguida, eles exploraram as informações contidas nos relatórios de problemas para reproduzir todas as falhas distintas. Para isso, eles tentaram reproduzir manualmente os problemas relatados em um dispositivo LG G4 H815 real equipado com Android 6.0. Dessa forma, podemos considerar apenas falhas reais causadas por lógica de aplicação incorreta e descartar aquelas causadas por problemas vinculados à infraestrutura de teste e ao dispositivo virtual. No que diz respeito às falhas da GUI, eles também avaliaram se cada falha era realmente a manifestação de um estado incorreto da GUI, em vez de um comportamento pretendido do aplicativo, por exemplo, um cronômetro que continua a contagem regressiva ou um feed de notícias que adiciona novos elementos pode fazer com que a GUI estado seja diferente após a execução de uma Sequência de Eventos de Ciclo de Vida. Para garantir que os problemas estivessem realmente vinculados ao ciclo de vida da atividade, a equipe realizou uma atividade de depuração para verificar se os problemas eram uma manifestação de falhas que são exercidas pela execução do ciclo de vida da atividade. As falhas distintas validadas foram usadas para calcular os valores das métricas.

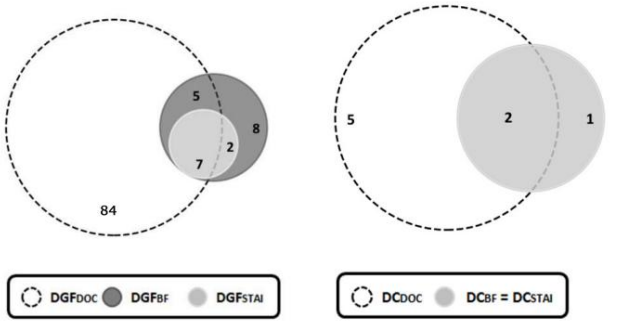
5.4 Resultados e Análise A Tabela

2 relata, para cada aplicativo, o número total de falhas e travamentos de GUI que foram encontrados pelo ALARic e validados pela equipe, agrupados pelo tipo de Sequência de Eventos do Ciclo de Vida que os acionou.

A Tabela 3 mostra para cada aplicativo o número total de falhas vinculadas às Sequências de Eventos do Ciclo de Vida detectadas pelo ALARic e pelo Monkey, respectivamente. Não comparamos os resultados em relação às falhas da GUI, pois o Monkey não é capaz de detectá-las.

No geral, o ALARic encontrou 111 falhas de GUI distintas e 8 falhas. A equipe validou como verdadeiros positivos 106 falhas de GUI e todas as falhas. Todos os aplicativos expuseram pelo menos 2 falhas de GUI e 6 aplicativos exibiram pelo menos uma falha. O DOC acionou o maior número (96) de falhas de GUI e conseguiu expor falhas de GUI em todos os aplicativos considerados. Um total de 22 falhas de GUI vinculadas ao BF foram encontradas em 9 aplicativos. O STAI acionou 9 falhas de GUI em 5 aplicativos. No que diz respeito às falhas, o DOC acionou o maior número de falhas (7) em 5 aplicativos. Um total de 3 falhas relacionadas à sequência BF foram encontradas em 2 aplicativos, enquanto o STAI acionou 3 falhas em 2 aplicativos.

Analisamos as relações entre os conjuntos de questões expostas por cada uma das três Sequências de Eventos do Ciclo de Vida consideradas. Conforme mostrado pelos diagramas de Venn relatados na Fig. 6, em alguns casos o mesmo problema foi exposto por mais de um tipo de ciclo de vida



(a) Falhas de GUI acionadas pelos Eventos de Ciclo de Vida

(b) Falhas desencadeadas pelo Ciclo de Vida Eventos

Figura 6: Problemas detectados pelo ALARic

Sequência de eventos, enquanto outros problemas foram acionados por apenas um tipo de sequência de eventos do ciclo de vida.

Como mostra a Fig. 6(a), 7 das 106 falhas de GUI detectadas pelo ALARic foram encontradas por todas as três Sequências de Eventos de Ciclo de Vida consideradas. Entre as 96 falhas de GUI acionadas pelo DOC, 84 não foram encontradas pelo BF e STAI. 8 falhas de GUI foram acionadas apenas por BF. A Fig. 6(b) ilustra que apenas as 2 falhas expostas por A9 foram acionadas por todas as três Sequências de Eventos de Ciclo de Vida consideradas. 5 de 8 travamentos foram acionados apenas pelo DOC. Em vez disso, o acidente exposto pelo A11 foi desencadeado pelo BF e STAI, mas não pelo DOC.

Em conclusão, esses resultados sugerem que o DOC tem maior probabilidade de expor problemas vinculados ao ciclo de vida da atividade, pois foi o mais eficaz em revelar falhas e travamentos da GUI em nosso experimento. No entanto, o BF não precisa ser negligenciado, pois mostrou a capacidade de descobrir problemas que as outras Sequências de Eventos do Ciclo de Vida perderam. Também o STAI que exerce o FL e tem um impacto limitado no ciclo de vida da Atividade, levou à deteção de problemas.

Com base nos resultados obtidos, pudemos responder a primeira questão de pesquisa RQ1 e concluímos que:

> O ALARic detectou problemas vinculados ao ciclo de vida da atividade em todos os aplicativos analisados. Ele expôs falhas e travamentos da GUI. As sequências de eventos de ciclo de vida que exercitam diversos loops de ciclo de vida chave mostraram diferentes recursos para expor problemas de aplicativos.

Com relação à comparação entre o ALARic e o Monkey, os dados da Tabela 3 mostram que, para 6 de 7 aplicativos, o ALARic conseguiu encontrar mais falhas vinculadas ao ciclo de vida da atividade do que o Monkey. Em A4 ambas as ferramentas expuseram a mesma falha. Além disso, ambas as ferramentas detectaram uma falha adicional no A9 que não estava vinculada ao ciclo de vida da atividade. Para entender melhor esse resultado analisamos detalhadamente os relatórios produzidos pela Monkey. Ele foi capaz de semear eventos que exercitam o ciclo de vida da Activity, por exemplo, mudanças de orientação, botão voltar, mas os aplicou sem uma estratégia adequada, falhando em descobrir vários problemas vinculados ao ciclo de vida da Activity que foram encontrados pelo ALARic. Com base nestes dados pudemos responder ao RQ2 concluindo que:

Tabela 2: Resultados Experimentais

	Falhas na GUI				Falhas			
App	#DGFT ot al	#DGFDOC	#DGFB F	#DGFST AI	#DCT ot al	#DCDOC	#DCB F	#DCST AI
A1	12	9	5	1	0	0	0	0
A2	5	5	0	0	0	0	0	0
A3	5	4	3	0	0	0	0	0
A4	8	8	0	0	1	1	0	0
A5	4	3	2	1	0	0	0	0
A6	8	8	0	0	1	1	0	0
A7	7	7	0	0	1	1	0	0
A8	2	1	1	1	0	0	0	0
A9	17	17	3	3	2	2	2	2
A10	5	4	2	0	0	0	0	0
A11	8	6	4	3	1	0	1	1
A12	3	3	1	0	0	0	0	0
A13	4	3	1	0	0	0	0	0
A14	13	13	0	0	0	0	0	0
A15	5	5	0	0	2	2	0	0
Total	106	96	22	9	8	7	3	3

Tabela 3: Comparação Experimental

Aplicativo	#DCALARic	#DCMonk	ei
A1	0	0	
A2	0	0	
A3	0	0	
A4	1	1	
A5	0	0	
A6	1	0	
A7	1	0	
A8	0	0	
A9	2	0	
A10	0	0	
A11	1	0	
A12	0	0	
A13	0	0	
A14	0	0	
A15	2	1	
Total	8	2	

O ALARic superou a ferramenta de última geração na capacidade de detectar problemas vinculados ao ciclo de vida da atividade. No total, desencadeou mais trava do que o Monkey.

5.5 Lição aprendida

Os resultados experimentais mostraram que as Sequências de Eventos do Ciclo de Vida são capazes de exercitar o ciclo de vida da Activity e expor falhas.

A atividade de depuração que realizamos na etapa de validação de falha mostrou-nos que as falhas que causavam as falhas estavam localizadas principalmente fora do código que substitui os métodos de retorno de chamada do ciclo de vida.

Como exemplo, a falha encontrada em A11 ocorre quando o onSave Método de retorno de chamada InstanceState() do EditProjectActivity é chamado, mas sua causa está localizada dentro da classe LinkEditWidget que define um objeto GUI personalizado. O programador realmente substituiu o método de retorno de chamada onSaveInstanceState() para salvar em tempo de execução o estado do objeto GUI personalizado LinkEditWidget contido na atividade EditProjectActivity . Para isso, o programador serializou corretamente os objetos LinkEditWidget e implementou adequadamente a interface Serializable na classe que define o objeto LinkEditWidget . No entanto, o definido pelo usuário LinkEditWidget contém componentes GUI android.widget.Spinner que não implementam a interface Serializable . Portanto, um java.io.NotSerializableException é lançado em tempo de execução quando o ciclo de vida da atividade EditProjectActivity é exercido . Outro exemplo está relacionado a uma falha que considerou 57 das 106 falhas de GUI detectadas pelo ALARic. Envolveu o Diálogo objetos desaparecendo da GUI após a execução de um ciclo de vida Sequência de Eventos. Essa falha afetou a maioria dos aplicativos considerados já que 12 dos 15 aplicativos o expuseram. Uma caixa de diálogo é uma pequena janela que não preenche a tela e é normalmente usado para eventos modais que exigir que os usuários executem uma ação antes que possam prosseguir. Na maioria casos, a falha que causa essas falhas foi localizada em objetos chamando diretamente o método public show oferecido pelo Diálogo ou as classes AlertDialog Builder para exibir uma caixa de diálogo na tela. Isto irá fazer aparecer correctamente a janela no ecrã mas a janela desaparecerá quando a Atividade for destruída e recriada em seu ciclo da vida. Em vez disso, as diretrizes do Android prescrevem explicitamente que o controle de um objeto GUI de diálogo (decidindo quando mostrar, ocultar, descartar it) deve ser gerenciado pela classe DialogFragment , que garante um tratamento correto das Sequências de Eventos do Ciclo de Vida13 .

Graças a esta análise, aprendemos duas lições que podem ser útil para desenvolvedores Android. A primeira lição é que eles devem usar corretamente os componentes da estrutura do Android, pois eles podem causar inconsistências no comportamento do aplicativo em tempo de execução quando ocorrem sequências de eventos do ciclo de vida. O último é que eles devem procurar falhas que possam afetar o ciclo de vida das Atividades também fora do métodos que substituem os retornos de chamada do ciclo de vida.

5.6 Ameaças à Validade

Esta seção discute as ameaças que podem afetar a validade de os resultados obtidos no estudo [20].

5.6.1 Validade Interna. Sabemos que as falhas que observamos pode não ser causado exclusivamente por Sequências de Eventos de Ciclo de Vida, mas também por fatores alternativos, como a plataforma de execução ou o tempo entre eventos consecutivos. Para mitigar essa ameaça, durante a etapa de validação, cada falha detectada foi manualmente reproduzidos em um dispositivo real para excluir que eles estavam vinculados ao infraestrutura de testes. Um experimento controlado envolvendo diferentes Versões do sistema operacional Android, tipos de dispositivo e intervalos de tempo entre eventos devem ser realizados para investigar melhor este aspecto.

13https://developer.android.com/reference/android/app/DialogFragment.html

5.6.2 Validade externa. Estamos cientes de que a pequena amostra de aplicativos Android considerados pode afetar a generalização de nossos resultados experimentais e pretendemos confirmar nossas descobertas no futuro realizando uma experimentação mais ampla envolvendo um número maior de aplicativos.

6 TRABALHOS RELACIONADOS

O ciclo de vida da atividade foi identificado como uma das principais fontes de problemas para aplicativos Android por diferentes trabalhos na literatura. Portanto, técnicas de teste com o objetivo de expor esses problemas têm sido propostas.

Franke et ai. [8] apresentou uma abordagem baseada em teste de unidade para testar a conformidade do ciclo de vida do aplicativo. Sua abordagem considera as Atividades como unidades a serem testadas. As propriedades dependentes do ciclo de vida devem ser extraídas manualmente da especificação de requisitos funcionais e os métodos de retorno de chamada do ciclo de vida da atividade são usados para testar tais propriedades explorando asserções. Ao contrário de nossa técnica de caixa preta totalmente automatizada , essa abordagem depende muito do esforço manual para extrair requisitos e definir casos de teste de unidade baseados em asserções e requer a disponibilidade do código-fonte do aplicativo.

O trabalho de Zaeem et al. [21] baseia-se na intuição de que diferentes aplicativos e plataformas móveis compartilham um conjunto de recursos chamados de Recursos de Interação do Usuário e que há um senso comum de expectativa de como um aplicativo deve responder a esses recursos. Entre esses recursos, eles consideraram também o acionamento dos principais ciclos de vida. Eles propõem uma abordagem automatizada de geração de suíte de teste orientada por modelo e a implementam no QUANTUM, uma estrutura que gera automaticamente uma suíte de teste para testar os recursos de interação do usuário de um determinado aplicativo, aproveitando os oráculos de teste agnósticos do aplicativo com base nos estados da GUI expostos pelo aplicativo . Diferentemente de nossa abordagem, QUANTUM precisa de um conhecimento prévio do aplicativo em teste, pois requer um modelo de GUI de aplicativo gerado pelo usuário como entrada.

Adamsen et ai. [1] propuseram uma ferramenta chamada THOR que amplifica sistematicamente os casos de teste injetando sequências de eventos neutras que não devem afetar a funcionalidade do aplicativo em teste e a saída dos casos de teste originais. Eles se concentram em sequências de eventos que geralmente são negligenciadas em abordagens de teste tradicionais, incluindo aquelas que exercitam os principais ciclos de vida. O THOR aproveita os casos de teste existentes. Em vez disso, nossa abordagem não requer artefatos de teste existentes.

Shan et ai. [17] focou em uma classe de falhas específica devido ao correto manuseio dos dados que devem ser preservados quando os principais loops de atividade são exercidos. Eles nomearam erros de KR as falhas causadas por essas falhas. Esses autores propuseram uma técnica de análise estática automatizada para encontrar erros de KR. Eles também projetaram uma ferramenta que gera uma sequência de eventos de entrada que levam ao estado do aplicativo onde o erro KR se manifesta. Ao contrário do nosso trabalho, esta solução precisa de modificação do aplicativo para verificar as falhas vinculadas ao ciclo de vida da atividade, rastreando os campos do aplicativo e despejando os estados da GUI nos métodos de retorno de chamada do ciclo de vida da atividade.

G. Hu et ai. [9] introduziu o AppDoctor, um sistema de teste capaz de realizar uma análise dinâmica rápida do aplicativo em teste que visa revelar falhas do aplicativo. Nossa abordagem de teste é capaz de detectar também falhas de GUI. A análise de aplicativo proposta, chamada de execução aproximada, é mais rápida que a execução real, pois exercita um aplicativo invocando diretamente manipuladores de eventos. Nossa técnica, em vez disso, aciona eventos reais porque eles representam melhores interações reais do usuário. Desde a sua aproximação pode introduzir vários falsos positivos, o AppDoc tor automaticamente tenta verificar os bugs detectados reproduzindo -os usando eventos reais. Assim como nós, eles apontaram a relevância de exercitar o ciclo de vida da Activity em testes móveis. Portanto, eles introduziram aproximações de eventos de ciclo de vida entre os eventos suportados pelo AppDoctor.

Moran et al. [15] projetou o Crashscope, uma abordagem de teste totalmente automatizada para descobrir, relatar e reproduzir falhas de aplicativos Android. Eles também propõem uma abordagem de teste de caixa preta totalmente automatizada, mas se concentram apenas em um tipo de falha específico, ou seja, travamentos de aplicativos, enquanto nossa abordagem é capaz de encontrar também falhas de GUI. Eles identificam o evento de ciclo de vida de mudança de orientação dupla como uma das principais fontes de falhas e, portanto, sua exploração automatizada de aplicativos executa uma rotação dupla cada vez que encontram uma atividade rotativa . Visto que nossa abordagem é capaz de realizar três tipos diferentes de eventos de ciclo de vida capazes de cobrir os três principais ciclos de ciclo de vida da atividade. Além disso, nossa estratégia de exploração executa um evento de ciclo de vida toda vez que encontra um estado de GUI nunca encontrado antes durante a exploração.

Jun et ai. [10] propuseram o LeakDAF, uma abordagem de teste totalmente automatizada para detectar vazamentos de memória. Assim como nosso trabalho, eles propõem uma técnica de teste visando a conformidade do ciclo de vida de componentes de aplicativos Android que explora uma técnica de exploração automatizada de aplicativos e não precisa de modificação de aplicativos ou interação manual. Eles testam os aplicativos com dois eventos de ciclo de vida que exercitam apenas o ciclo de ciclo de vida inteiro. Considerando que, nossa abordagem estuda o efeito de eventos que exercem também os ciclos Visible Lifecycle e Foreground Lifecycle . Eles visam detectar um tipo específico de vazamento de memória, ou seja, o vazamento de componentes do aplicativo Activity e Fragment Android. Em vez disso, pretendemos propor uma técnica de teste capaz de detectar diferentes tipos de problemas relacionados ao ciclo de vida das atividades de

7 CONCLUSÕES E TRABALHOS FUTUROS

Neste artigo, apresentamos o ALARic, uma técnica de teste automatizado do Android que combina as abordagens tradicionais de teste baseadas na exploração dinâmica de aplicativos com uma estratégia que dispara três eventos específicos para dispositivos móveis capazes de expor problemas vinculados a recursos peculiares da plataforma Android. Nós nos concentramos no gerenciamento do ciclo de vida da atividade do Android e projetamos uma técnica que exercita sistematicamente o ciclo de vida das atividades do aplicativo, para detectar falhas e travamentos da GUI.

Nossa técnica foi implementada em uma ferramenta e validada em um estudo envolvendo 15 aplicativos do mundo real que mostraram a capacidade da ferramenta de detectar automaticamente problemas vinculados ao ciclo de vida da Activity. O estudo também mostrou que o ALARic foi mais eficaz na detecção de falhas do que ferramentas aleatórias padrão, como o Monkey, e nos permitiu aprender algumas lições úteis para testadores e desenvolvedores de aplicativos Android.

Como trabalho futuro, planejamos estender a ferramenta ALARic adicionando outras Sequências de Eventos de Ciclo de Vida além das três já implementadas. Pretendemos propor e implementar um conjunto de oráculos capazes de detectar outros problemas ligados ao ciclo de vida da Activity, como vazamentos de memória e problemas de threading. Para provar melhor a eficácia do ALARic, planejamos realizar uma experimentação mais ampla envolvendo um conjunto maior de aplicativos Android e considerando diferentes configurações da ferramenta. Por fim, planejamos estender nossa abordagem para testar o ciclo de vida de outros componentes do aplicativo, como serviços, fragmentos e provedores de conteúdo.

REFERÊNCIAS

[1] Christoffer Quist Adamsen, Gianluca Mezzetti e Anders Møller. 2015. Execução Sistemática de Suítes de Testes Android em Condições Adversas. In Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015). ACM, Baltimore, MD, EUA, 83-93. http://doi.acm.org/10.1145/2771783.2771786 [2] Domenico Amalfitano, Nicola Amatucci, Atif M. Memon, Porfirio Tramontana e Anna Rita Fasolino. 2017. Uma estrutura geral para comparar técnicas de teste automático de aplicativos móveis Android. Journal of Systems and Software 125 (2017), 322–343.

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana e Bryan Robbins. 2013. Testando aplicativos móveis Android: desafios, estratégias e abordagens. Avanços em Computadores 89 (2013), 1–52.

[4] D. Amalfitano, AR Fasolino, P. Tramontana, BD Ta e AM Memon. 2015. MobiGUITAR: Testes Automatizados Baseados em Modelos de Aplicativos Móveis. Software IEEE 32, 5 (setembro de 2015), 53–59.

[5] Domenico Amalfitano, Vincenzo Riccio, Ana CR Paiva, and Anna Rita Fasolino. 2018. Por que a mudança de orientação atrapalha meu aplicativo Android? De falhas de GUI a falhas de código. Softw. Teste., Verif. Confiável. 28, 1 (2018).

[6] F. Belli, M. Beyazit e A. Memon. 2012. O teste é uma atividade centrada em eventos. Em Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on. 198-206.

[7] Shauvik Roy Choudhary, Alessandra Gorla e Alessandro Orso. 2015. Geração de entrada de teste automatizada para Android: já chegamos? (E). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15). IEEE Computer Society, Washington, DC, EUA, 429–440.

[8] D. Franke, S. Kowalewski, C. Weise e N. Prakobkosol. 2012. Testando a Conformidade das Propriedades Dependentes do Ciclo de Vida de Aplicativos Móveis. Em 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. 241-250.

[9] Gang Hu, Xinhao Yuan, Yang Tang e Junfeng Yang. 2014. Detectando bugs de aplicativos móveis com eficiência e eficácia com o AppDoctor. In Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14). ACM, Nova York, NY, EUA, Artigo 18, 15 páginas.

[10] M. Jun, L. Sheng, Y. Shengtao, T. Xianping e L. Jian. 2017. LeakDAF: uma ferramenta automatizada para detectar atividades e fragmentos vazados de aplicativos Android . Em 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), Vol. 1. 23–32.

[11] V. Lelli, A. Blouin e B. Baudry. 2015. Classificação e qualificação de defeitos de GUI. Em 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). 1-10.

[12] Riad Mahmood, Nariman Mirzaei e Sam Malek. 2014. EvoDroid: Teste Evolutivo Segmentado de Aplicativos Android. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014). ACM, Nova York, NY, EUA, 599-609.

[13] Ke Mao, Mark Harman e Yue Jia. 2016. Sapienz: Testes Automatizados Multiobjetivos para Aplicativos Android. In Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016). ACM, Nova York, NY, EUA, 94-105.

[14] Abel Méndez-Porras, Giovanni Méndez-Marín, Alberto Tablada-Rojas, Mario Ni eto Hidalgo, Juan Manuel García-Chamizo, Marcelo Jenkins e Alexandra Martínez. 2017. Um analisador de bug distribuído baseado em recursos de interação do usuário para aplicativos móveis. Journal of Ambient Intelligence and Humanized Computing 8, 4 (01 de agosto de 2017), 579–591.

[15] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome e D. Poshy vanyk. 2016. Automaticamente descobrindo, relatando e reproduzindo falhas de aplicativos Android. Em 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST). 33-44.

[16] Henry Muccini, Antonio di Francesco e Patrizio Esposito. 2012. Teste de software de aplicativos móveis: Desafios e direções de pesquisas futuras. Em Automação de Teste de Software (AST), 2012 7º Workshop Internacional sobre. IEEE, Zurique, Suíça, 29-35.

[17] Zhiyong Shan, Tanzirul Azim e Iulian Neamtiu. 2016. Encontrando Erros de Retomar e Reinicie em Aplicativos Android. SIGPLAN Não. 51, 10 (outubro de 2016), 864-880.

[18] Estatista. 2016. Número de usuários de smartphones em todo o mundo a partir de 2014 até 2019 (em milhões). https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/ [19] Statista. 2017. Participação de mercado global detida pelos principais sistemas operacionais de smartphones em vendas para usuários finais do 1º trimestre de 2009 ao 1º trimestre de 2017. https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/ [20] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell e And ders Wessln. 2012. Experimentação em Engenharia de Software. Springer Publishing Company, Incorporated.

[21] Razieh Nokhbeh Zaeem, Mukul R. Prasad e Sarfraz Khurshid. 2014. Geração Automatizada de Oracles para Teste de Recursos de Interação com o Usuário de Aplicativos Móveis. In Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation (ICST '14). IEEE Computer Society, Washington, DC, EUA, 183–192.

[22] Samer Zein, Norsaremah Salleh e John Grundy. 2016. Um estudo de mapeamento sistemático de técnicas de teste de aplicativos móveis. J. Syst. Softw. 117, C (julho de 2016), 334-356.

[23] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang e Tao Xie. 2016. Geração de entrada de teste automatizada para Android: estamos realmente lá ainda em um caso industrial?. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016). ACM, Nova York, NY, EUA, 987-992.

[24] Haibing Zheng, Dengfeng Li, Beihai Liang, Xia Zeng, Wujie Zheng, Yuetang Deng, Wing Lam, Wei Yang e Tao Xie. 2017. Geração automatizada de entrada de teste para Android: para chegar lá em um caso industrial. In Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '17). IEEE Press, Piscataway, NJ, EUA, 253–262.