UNIVERSIDADE FEDERAL DO AMAZONAS - UFAM

INSTITUTO DE COMPUTAÇÃO - ICOMP

PROGRAMA PÓS-GRADUAÇÃO EM INFORMÁTICA - PPGI

Previsão de Vazamento de Recursos em Aplicações Android usando Aprendizado de Máquina

Josias Gomes Lima

Manaus - AM

Setembro de 2024

Josias Gomes Lima

Previsão de Vazamento de Recursos em Aplicações Android usando Aprendizado de Máquina

Tese de doutorado submetida à avaliação, como parte dos requisitos necessários à obtenção do título de Doutor em Informática no Programa de Pós-Graduação em Informática, Instituto de Computação.

Orientador

Prof. Dr. Rafael Giusti

Coorientador

Prof. Dr. Arilo Claudio Dias Neto

Universidade Federal do Amazonas - UFAM

Instituto de Computação - IComp

Manaus - AM

Setembro de 2024

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

Lima, Josias Gomes
L732p Previsão de Vazamento de Recursos em Aplicações Android usando Aprendizado de Máquina / Josias Gomes Lima . 2024
133 f.: il. color; 31 cm.

Orientador: Rafael Giusti
Coorientador: Arilo Claudio Dias Neto
Tese (Doutorado em Informática) - Universidade Federal do Amazonas.

1. vazamentos de recursos. 2. aplicações móveis. 3. aprendizado de máquina. 4. teste. 5. análise estática. I. Giusti, Rafael. II. Universidade Federal do Amazonas III. Título

Ministério da Educação
Universidade Federal do Amazonas
Coordenação do Programa de Pós-Graduação em Informática

FOLHA DE APROVAÇÃO

"PREVISÃO DE VAZAMENTO DE RECURSOS EM APLICAÇÕES ANDROID USANDO APRENDIZADO DE MÁQUINA"

JOSIAS GOMES LIMA

Tese de Doutorado defendida e aprovada pela banca examinadora constituída pelos Professores:

Prof. Dr. Rafael Giusti - **Presidente**

Prof. Dr. Raimundo da Silva Barreto - **Membro Interno**

Prof. Dr. Eduardo James Pereira Souto - **Membro Interno**

Prof. Dr. Marco Antônio Pinheiro de Cristo - **Membro Externo**

Prof. Dr. Moisés Gomes de Carvalho - **Membro Externo**

Manaus, 11 de setembro de 2024.

Documento assinado eletronicamente por **Rafael Giusti**, **Professor do**

Magistério Superior, em 17/09/2024, às 16:59, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Raimundo da Silva Barreto**, **Professor do Magistério Superior**, em 17/09/2024, às 17:46, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Eduardo James Pereira Souto**, **Professor do Magistério Superior**, em 18/09/2024, às 15:05, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Marco Antônio Pinheiro de Cristo**, **Professor do Magistério Superior**, em 23/09/2024, às 15:46, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Moisés Gomes de Carvalho**, **Professor do Magistério Superior**, em 01/10/2024, às 13:39, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://sei.ufam.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **2224260** e o código CRC **CA5008D8**.

Avenida General Rodrigo Octávio, 6200 - Bairro Coroado I Campus Universitário Senador Arthur Virgílio Filho, Setor Norte - Telefone: (92) 3305-1181 / Ramal 1193
CEP 69080-900, Manaus/AM, coordenadorppgi@icomp.ufam.edu.br

Referência: Processo nº 23105.038913/2024-10 SEI nº 2224260

A Deus, a minha família e amigos.

AGRADECIMENTOS

A Deus, pela dádiva da vida e por me cercar de pessoas que foram essenciais para que eu pudesse alcançar mais esta conquista.

À minha família, em especial aos meus pais, Antonia e Felix, pelo amor e apoio incondicionais, e aos meus irmãos, David, Allan, Mayra e Nayra. Agradeço também à minha cunhada, Bianca, e aos meus irmãos de coração, Jodi e Moisés, por sua constante presença e carinho.

Aos amigos da graduação, Verônica, Arcanjo e Francisco, cujo incentivo foi fundamental para minha entrada no mestrado e doutorado.

Aos meus orientadores, professores Rafael e Arilo, pela confiança, paciência e pelas valiosas orientações ao longo dessa jornada acadêmica.

Aos queridos amigos do grupo de pesquisa ExperTS, Karina, Silvia, Awdren, Laiza, Oswald, Larissa, Kariny, Renata, Ivanilse, Jonathas e PH Munhoz, por todos os momentos de troca e aprendizado.

Aos participantes do estudo experimental, que gentilmente dedicaram seu tempo e contribuíram para o desenvolvimento deste trabalho, meu sincero agradecimento.

Ao corpo docente e administrativo do Instituto de Computação, pelo suporte e auxílio durante todo o período do doutorado.

Aos amigos e familiares que pacientemente ouviram inúmeras vezes que eu estava "terminando o doutorado".

À CAPES, pelo apoio financeiro que tornou possível a realização desta pesquisa.

Por fim, a todos que, de alguma forma, contribuíram para a conclusão deste trabalho, meu muito obrigado.

Previsão de Vazamento de Recursos em Aplicações Android usando Aprendizado de Máquina

Autor: Josias Gomes Lima

Orientador: Prof. Dr. Rafael Giusti

Coorientador: Prof. Dr. Arilo Claudio Dias Neto

Resumo

Quando as aplicações móveis adquirem recursos do dispositivo (como câmera, reprodutor de mídia e sensores) sem liberá-los da maneira adequada e em tempo hábil, ocorre uma falha chamada **vazamento de recursos**. Esse tipo de falha pode causar problemas sérios, como degradação de desempenho do dispositivo ou falha do sistema. Este trabalho propõe a abordagem *LeakPred* para auxiliar desenvolvedores na identificação de componentes que tenham vazamentos de recursos. Um conjunto de seis métricas relacionadas ao tempo de vida dos recursos ou da aplicação foi selecionado para a caracterização dos componentes. Seis técnicas de aprendizado de máquina foram analisadas para identificar componentes com vazamentos a partir dessas métricas. Os resultados sugerem que a abordagem LeakPred, associada com técnicas de classificação, é capaz de identificar vazamento de recursos, sendo que dois modelos, k-Vizinhos Mais Próximos e rede neural profunda, obtiveram, respectivamente, acurácias de 87,84% e 87,75%. A abordagem LeakPred foi comparada com 5 ferramentas do estado da arte, a saber, *Android Lint*, *FindBugs*, *Infer*, *Checker Framework* e *EcoAndroid*, superando todas em taxa de identificação de componentes com vazamentos de recursos.

Palavras-chave: predição de defeitos, aplicações Android, análise estática, vazamento de recursos.

Resource Leak Prediction in Android Applications Using Machine Learning

Author: Josias Gomes Lima

Advisor: Prof. Dr. Rafael Giusti

Co-advisor: Prof. Dr. Arilo Claudio Dias Neto

Abstract

When mobile applications acquire device resources (such as camera, media player, and sensors) without releasing them properly and in a timely manner, a failure called **resource leak** occurs. This type of failure can cause serious problems, such as device performance degradation or system failure. This work proposes the *LeakPred* approach to assist developers in identifying components that have resource leaks. A set of six metrics related to the lifetime of resources or the application was selected to characterize the components. Six machine learning techniques were analyzed to identify leaky components from these metrics. The results suggest that the LeakPred approach, associated with classification techniques, is capable of identifying resource leaks, with two models, k-Nearest Neighbors and deep neural network, obtaining, respectively, accuracies of 87.84% and 87.75%. The LeakPred approach was compared with 5 state-of-the-art tools, namely, *Android Lint*, *FindBugs*, *Infer*, *Checker Framework* and *EcoAndroid*, surpassing all of them in the rate of identification of components with resource leaks.

Keywords: defect prediction, Android applications, static analysis, resource leaks.

LISTA DE ILUSTRAÇÕES

Figura 1.1 – Método de pesquisa. 22
Figura 2.1 – Ilustração simplificada do ciclo de vida da Atividade. 27
Figura 3.1 – Publicações por ano. 44
Figura 3.2 – Publicações por tipo de local. 45
Figura 3.3 – Tipos de contribuições. 47
Figura 3.4 – Tipos de pesquisa. 49
Figura 4.1 – Quantidade de marcações como sendo de nenhuma relevância pelos participantes. 79
Figura 5.1 – Visão geral da abordagem *LeakPred*. 83
Figura 6.1 – Acurácia de teste para os classificadores. 96
Figura 6.2 – A área sob a curva de característica de operação do receptor (*ROC AUC*) para os classificadores. 97
Figura 7.1 – Processo de identificação de vazamento de recursos 101
Figura 7.2 – Seleção das Aplicações Móveis. 103
Figura 7.3 – Processo de execução do estudo. 105
Figura 7.4 – Componentes com vazamentos, taxa de detecção e de falsos positivos na aplicação OI Notepad. 108
Figura 7.5 – Componentes com vazamentos, taxa de detecção e de falsos positivos na aplicação OI Safe. 110
Figura 7.6 – Componentes, taxa de detecção e de falsos positivos com vazamentos na aplicação AnyMemo. 111

Figura 7.7 – Componentes com vazamentos, taxa de detecção e de falsos positivos na aplicação Avare. 113
Figura 7.8 – Componentes com vazamentos, taxa de detecção e de falsos positivos na aplicação Seafile. 115

LISTA DE TABELAS

Tabela 2.1 – Matriz de Confusão. 37
Tabela 3.1 – A *string* na estrutura PICO para o mapeamento realizado. 41
Tabela 3.2 – Formulário de Extração de Dados. 43
Tabela 3.3 – Número de artigos selecionados. 44
Tabela 3.4 – Principais locais de publicação. 45
Tabela 3.5 – Classes de recurso. 51
Tabela 3.6 – Comparação com os trabalhos relacionados sobre vazamentos de recursos. 72
Tabela 4.1 – Objetivo da pesquisa de opinião segundo o paradigma GQM. 74
Tabela 4.2 – Participantes da pesquisa de opinião. 77
Tabela 4.3 – Relevância das categorias de causas de vazamento. 78
Tabela 4.4 – Lista final das categorias de causas de vazamento. 80
Tabela 5.1 – Lista de métricas extraídas. 85
Tabela 5.2 – Lista de classes de recurso. 86
Tabela 5.3 – Exemplo de um conjunto de componentes com vazamento. 87
Tabela 6.1 – Melhores hiperparâmetros por execução 92
Tabela 6.2 – Matriz de confusão dos classificadores. 96
Tabela 6.3 – Métricas para os classificadores. 97
Tabela 7.1 – Palavras-chave para logs dos commits e diferenças de código. 102
Tabela 7.2 – As 32 aplicações selecionadas. 104

Tabela 7.3 – As classes de vazamento de recursos que poderiam ser identificadas por cada ferramenta nas aplicações OI Notepad, OI Safe, AnyMemo, Avare e Seafile. 106
Tabela 7.4 – Resumo do resultado das aplicações. 107
Tabela 7.5 – Componentes com vazamentos de recursos na aplicação notepad. . . 107
Tabela 7.6 – Componentes com vazamentos na aplicação safe. 109
Tabela 7.7 – Componentes com vazamentos de recursos na aplicação anymemo. . 111
Tabela 7.8 – Componentes com vazamentos de recursos na aplicação avare. . . . 112
Tabela 7.9 – Componentes com vazamentos de recursos na aplicação seafile. . . . 114

SUMÁRIO

1 INTRODUÇÃO 17
1.1 Contextualização e Motivação 17
1.2 Descrição do Problema 18
1.3 Hipótese 21
1.4 Objetivos 21
1.4.1 Objetivo Geral 21
1.4.2 Objetivos Específicos 21
1.5 Método de Pesquisa 22
1.5.1 Fase de Concepção 22
1.5.2 Fase de Avaliação 23
1.6 Estrutura do Documento 23

2 FUNDAMENTOS 25
2.1 Aplicações Móveis 25
2.2 Vazamento de Recursos nas Aplicações Móveis 28
2.3 Aprendizado de Máquina 30
2.3.1 Máquina de Vetores de Suporte 31
2.3.2 Naive Bayes 32
2.3.3 Redes Neurais Profundas 32
2.3.4 Regressão Logística 33
2.3.5 Floresta Aleatória 33
2.3.6 K-Vizinhos mais Próximos 33
2.4 Predição de Defeitos 34
2.4.1 Definição da Base de Dados 35
2.4.2 Definição das Métricas 35

2.4.3 Construção do Modelo . . . 35
2.4.4 Avaliação de Desempenho . . . 36
2.5 Considerações Finais . . . 38

3 MAPEAMENTO SISTEMÁTICO . . . 39
3.1 Objetivo e Questões de Pesquisa . . . 39
3.2 Identificação dos Estudos . . . 40
3.2.1 Esquema de Classificação . . . 41
3.3 Estratégia de Extração de Dados . . . 43
3.4 Execução do Estudo do Mapeamento Sistemático . . . 43
3.5 Resultados do Mapeamento Sistemático . . . 44
3.5.1 RQ 1.1 - Tipos de contribuição . . . 46
3.5.2 RQ 1.2 - Tipos de pesquisa . . . 48
3.5.3 RQ 1.3 - Classes de recurso . . . 49
3.5.4 RQ 1.4 – Categoria de causas de vazamento . . . 52
3.6 Conclusões do Mapeamento Sistemático . . . 67
3.7 Trabalhos Relacionados . . . 68
3.7.1 Detecção de Vazamentos de Recursos . . . 68
3.8 Considerações Finais . . . 73

4 PESQUISA DE OPINIÃO . . . 74
4.1 Objetivo . . . 74
4.2 Questão de Pesquisa . . . 75
4.3 Instrumentação . . . 75
4.4 Procedimento para Análise da Relevância . . . 75
4.5 Execução do Estudo . . . 76
4.6 Análise dos Resultados . . . 78
4.7 Ameaças à Validade . . . 81
4.8 Conclusões do Estudo . . . 81
4.9 Considerações Finais . . . 82

5 ABORDAGEM PROPOSTA . . . 83

5.1 Visão Geral da Proposta 83
5.1.1 Preparação dos Dados 84
5.1.2 Execução do Treinamento 85
5.1.3 Classificação dos Componentes 87
5.2 Considerações Finais 88

6 PROVA DE CONCEITO 89
6.1 Objetivo do Estudo 89
6.1.1 Base de Dados 89
6.1.2 Otimização de Hiperparâmetros 90
6.1.3 Execução do Estudo 92
6.1.4 Análise dos Resultados 92
6.1.5 Ameaças à Validade 98
6.1.6 Conclusões do Estudo 98
6.2 Considerações Finais 99

7 ESTUDO DE VIABILIDADE 100
7.1 Objetivo do Estudo 100
7.1.1 Metodologia para a Identificação dos Componentes com Vazamento de Recursos 101
7.1.2 Seleção das Aplicações Móveis 102
7.1.3 Seleção das Ferramentas 103
7.1.4 Execução do Estudo 104
7.2 Resultados 106
7.2.1 Resultados Aplicação OI Notepad 108
7.2.2 Resultados Aplicação OI Safe 109
7.2.3 Resultados Aplicação AnyMemo 110
7.2.4 Resultados Aplicação Avare 112
7.2.5 Resultados Aplicação Seafile 113
7.3 Ameaças à Validade 116
7.4 Conclusões do Estudo 117

7.5 Considerações Finais 117

8 CONCLUSÕES E TRABALHOS FUTUROS 118
8.1 Considerações Finais 118
8.2 Contribuições 120
8.3 Limitações e Decisões do Projeto de Implementação 121
8.4 Trabalhos futuros 122

Referências 124

ANEXO A ANEXOS 133

1

INTRODUÇÃO

Neste capítulo serão apresentados o contexto, a motivação e o problema que será tratado nesta proposta de tese. Também serão apresentados a hipótese, os objetivos, o método de pesquisa adotado e a organização deste documento.

1.1 Contextualização e Motivação

Os dispositivos móveis se tornaram uma parte importante do dia-a-dia de muitas pessoas ao redor do mundo. A quantidade de dispositivos utilizados ultrapassou 7 bilhões em 2023 e esse número tende a crescer nos próximos anos (STATISTA, 2024d). O uso destes dispositivos é influenciado pelos diversos recursos, em geral limitados, que eles oferecem, tais como recursos de mídia, câmera, sensores, acesso à Internet, dentre outros, e ainda mais intensificado pela diversidade de aplicações que podem ser utilizadas neles, as chamadas aplicações móveis. No ano de 2023, os usuários baixaram cerca de 257 bilhões de aplicações móveis em seus dispositivos, ante 104,7 bilhões no ano de 2016 (STATISTA, 2024c). O gasto médio do usuário em aplicações móveis por dispositivo no terceiro trimestre de 2023 foi calculado em 5,05 dólares, enquanto, no mesmo trimestre de 2019, o dispêndio foi de 4,11 dólares (STATISTA, 2024a).

Garantir a qualidade das aplicações móveis tem sido um desafio para os desenvolvedores, pois além dos defeitos tradicionais de uma aplicação, há os defeitos relacionados à utilização dos recursos limitados dos dispositivos. Cada modelo de

dispositivo possui um subconjunto de diferentes recursos e o correto gerenciamento de cada recurso por parte da aplicação móvel não é trivial, podendo variar de recurso para recurso.

1.2 Descrição do Problema

Os dispositivos móveis possuem vários recursos, como microfone, GPS (Sistema de Posicionamento Global, em inglês, *Global Positioning System*), memória, câmera, NFC (Comunicação por Campo de Proximidade, em inglês, *Near Field Communication*), sensores e bluetooth. Quando uma aplicação móvel utiliza algum desses recursos e não o libera após a sua utilização, ocasiona uma falha chamada **vazamento de recurso** (em inglês, *resource leak*). Essa falha pode ocorrer devido a uma ação tomada por um componente que contém um defeito[1]. No contexto deste trabalho, um componente é definido como um método de uma classe em um programa orientado a objetos. Alguns dos problemas que podem ocorrer quando há vazamento de recursos são os seguintes.

1. **Enorme consumo da bateria**: de acordo com (NASEER; NADEEM; ZAMAN, 2021) e (BANERJEE et al., 2014) pode ser categorizada em defeitos de energia (em inglês, *energy-bugs*) e pontos quentes de energia (em inglês, *energy-hotspots*). Uma aplicação é energeticamente ineficiente devido a um defeito de energia quando impede que o dispositivo fique ocioso mesmo após concluída a sua execução e não há atividade do usuário. Já uma aplicação energeticamente ineficiente devido a ponto quente de energia é aquela que consome grande quantidade de energia durante sua execução;

2. **Degradação da usabilidade e responsividade**: pode acontecer quando um recurso fica parcial ou totalmente indisponível ao usuário (BHATT; FURIA, 2022);

3. **Enorme consumo de memória**: a aplicação aloca recursos na memória do dispositivo e não faz a liberação, o que pode causar estouro de memória e consequente falha na utilização do dispositivo (AMALFITANO et al., 2020);

[1] O desenvolvedor comete um erro, que pode gerar um ou mais defeitos na aplicação, os quais, quando executados, podem causar uma falha.

4. **Degradação do desempenho**: por meio do esgotamento gradual dos recursos computacionais do dispositivo móvel em tempo de execução, ocasionando lentidão durante a utilização do dispositivo e da aplicação (LIU et al., 2019);

5. **Falha da aplicação**: pode ocorrer quando a aplicação se encerra inesperadamente devido à utilização de um recurso mal gerenciado (LIU et al., 2021);

6. **Problema em outra aplicação**: pode ocorrer quando um recurso que só pode ser usado em uma aplicação por vez não é liberado e outra aplicação tenta acessá-lo (RIGANELLI; MICUCCI; MARIANI, 2019a).

Código 1.1 – Exemplo do problema de pesquisa com erro.

```
1  public int getInt(String key) throws SQLException {
2      Cursor cur = getDB().getDatabase().rawQuery("SELECT value FROM deckVars WHERE
           key = '" + key + "'", null);
3      if (cur.moveToFirst()) {
4          return cur.getInt(0);
5      } else {
6          throw new SQLException("DeckVars.getInt: could not retrieve value for " +
               key);
7      }
8  }
```

No Código 1.1 é mostrado um trecho de código da aplicação *AnkiDroid*[2]. Na linha 2 é instanciado um objeto do tipo *Cursor*. Esse objeto é utilizado nas linhas 3 e 4, mas não é liberado pelo componente (método getInt). Isso faz com que o recurso fique alocado na memória e o Coletor de Lixo (em inglês, *Garbage Collection*) não possa recuperar essa área de memória inutilizada. Após várias chamadas ao componente *getInt*, pode ocorrer um estouro de memória (*java.lang.OutOfMemoryError*). No Código 1.2 é mostrada a correção do código implementada pelo autor da aplicação[3], em que a instrução de liberação do recurso é implementada na linha 12.

[2] <https://github.com/ankidroid/Anki-Android/blob/3e9ddc7eca6cd035e1cf6f82cb4e5d1e3b7220cf/src/com/ichi2/anki/Deck.java>

[3] <https://github.com/ankidroid/Anki-Android/blob/3a579e409118bcfa5027f71d7843730c9ab99eb7/src/com/ichi2/anki/Deck.java>

Código 1.2 – Exemplo do problema de pesquisa corrigido.

```
1   public int getInt(String key) throws SQLException {
2       Cursor cur = null;
3       try {
4           cur = getDB().getDatabase().rawQuery("SELECT value FROM deckVars
                WHERE key = '" + key + "'", null);
5           if (cur.moveToFirst()) {
6               return cur.getInt(0);
7           } else {
8               throw new SQLException("DeckVars.getInt: could not retrieve value
                    for " + key);
9           }
10      } finally {
11          if (cur != null) {
12              cur.close();
13          }
14      }
15  }
```

É responsabilidade do desenvolvedor da aplicação a identificação, o mais cedo possível, de componentes que possuem vazamentos de recursos. Existem diversos métodos na literatura técnica para identificação desses componentes, como analisadores estáticos (JIANG et al., 2017), técnicas de verificação (VEKRIS et al., 2012) e técnicas de teste (QIAN; ZHOU, 2016). No entanto, cada técnica identifica vazamentos em um conjunto pequeno de classes de recursos, o que faz com que o desenvolvedor necessite aprender a configurar e utilizar várias ferramentas. Além disso, nenhuma ferramenta existente é capaz de identificar todos os vazamentos de uma aplicação, como verificado em (PEREIRA et al., 2022). Os autores adicionaram suporte a detecção de vazamentos de recurso ao aplicativo *EcoAndroid* e realizaram uma análise de 107 aplicações móveis disponíveis em (LIU et al., 2019). Neste estudo, foram descobertos, com precisão de 72,5%, 191 vazamentos de recursos não identificados anteriormente (74 ao considerar vazamentos exclusivos), o que indica que ainda existe uma lacuna a ser explorada visando a maior eficácia da detecção desses vazamentos de recursos. Nesse contexto, esta pesquisa visa o desenvolvimento de uma solução para a identificação de vazamento em várias classes de recursos por meio da análise do código fonte de aplicações Android. Para isso, serão utilizadas técnicas de predição de defeito[4] e de

[4] Predição de defeito é o processo de identificar, com base em dados históricos e métricas de software,

aprendizado de máquina. O objetivo é indicar ao desenvolvedor componentes que possuem vazamentos de recursos para que ele ou ela possa realizar a correção do problema durante o desenvolvimento da aplicação.

1.3 Hipótese

Uma abordagem para a detecção de componentes com vazamentos de recursos, baseada em predição de defeitos, aprendizado de máquina e métricas de código, será mais eficaz na identificação de componentes com vazamentos em aplicações móveis Android em comparação com ferramentas existentes

1.4 Objetivos

1.4.1 Objetivo Geral

Identificar automaticamente, por meio de aprendizado de máquina e com base em métricas de software, vazamentos de recursos em aplicações móveis Android.

1.4.2 Objetivos Específicos

Para atingir o objetivo geral desta pesquisa, pretende-se alcançar os seguintes resultados intermediários:

- Identificar e definir métricas relacionadas ao problema de vazamento de recursos em aplicações Android;
- Definir uma abordagem para a detecção de vazamentos de recursos em aplicações móveis envolvendo aprendizado de máquina e métricas de código.

quais partes do código têm maior probabilidade de conter defeitos futuros (RAWAT; DUBEY, 2012).

1.5 Método de Pesquisa

Na Figura 1.1 é mostrado o método de pesquisa usado, o qual foi proposto em (MAFRA; BARCELOS; TRAVASSOS, 2006). Ele é composto por duas fases: **concepção**, cujo foco é a definição e construção da abordagem; e **avaliação**, focada em analisar a viabilidade da abordagem.

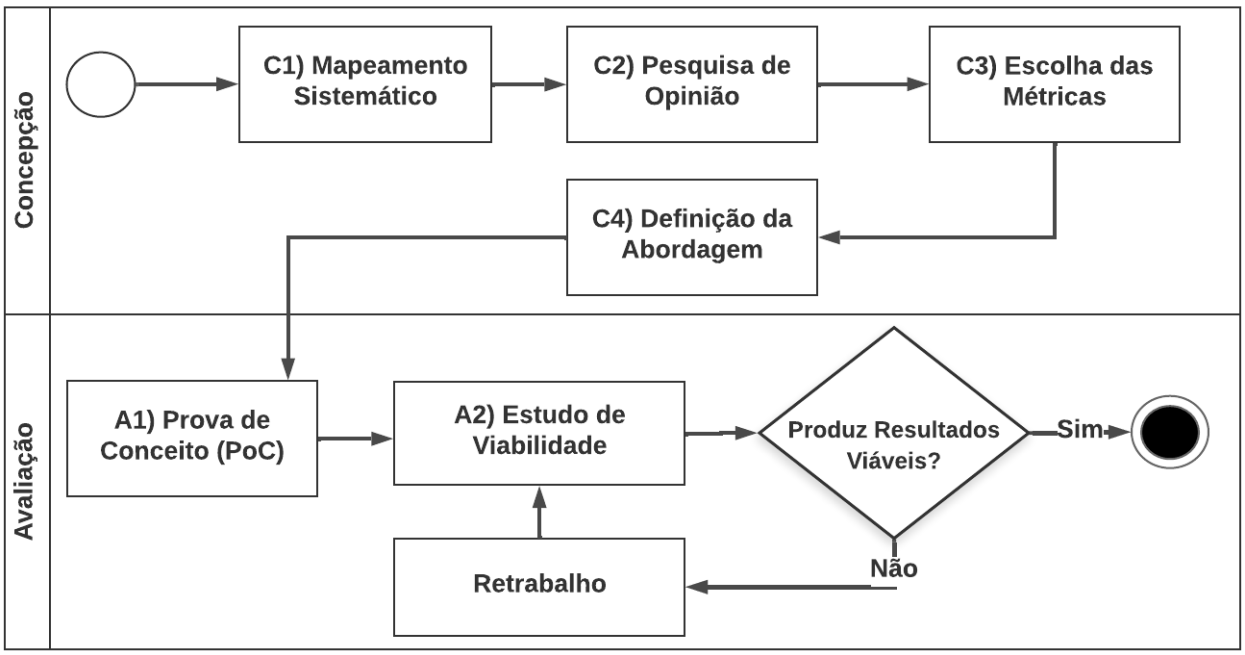


Figura 1.1 – Método de pesquisa.

1.5.1 Fase de Concepção

- **C1) Mapeamento Sistemático:** O mapeamento sistemático da literatura técnica foi realizado com o objetivo de identificar, analisar e sintetizar publicações científicas que tratam de identificação de vazamento de recursos em aplicações móveis. Os resultados do mapeamento são apresentados no Capítulo 3;

- **C2) Pesquisa de Opinião:** A pesquisa de opinião com especialistas foi realizada com o objetivo de caracterizar a relevância das categorias de causas de vazamentos identificadas no mapeamento sistemático. Os resultados da pesquisa de opinião podem ser vistos no Capítulo 4;

- **C3) Escolha das Métricas:** Nessa etapa foi realizada a identificação e a definição das métricas de código, em nível de componente, que serão utilizadas na abordagem. Os resultados dessa etapa são mostrados na Subseção 5.1.1;

- **C4) Definição da Abordagem:** Nessa etapa foi feita a definição da abordagem, como o artefato de entrada, as ferramentas a serem utilizadas e os processos utilizados para a identificação dos vazamentos. Os resultados dessa etapa podem ser vistos na Seção 5.1.

1.5.2 Fase de Avaliação

- **A1) Prova de Conceito (PoC):** Nessa etapa foi realizado um estudo experimental para verificar se a abordagem proposta produz resultados viáveis. Os resultados da prova de conceito são apresentados no Capítulo 6;

- **A2) Estudo de Viabilidade:** No estudo de viabilidade, realizou-se um estudo experimental com 15 aplicações móveis de código aberto, as quais foram selecionadas aleatoriamente da lista de aplicações disponibilizadas em (LIU et al., 2019), a fim de avaliar a eficácia (identificação de componentes com mais vazamentos de recursos com menos falso-positivo) da proposta em relação ao estado da arte. Os resultados desse estudo são mostrados no Capítulo 7.

1.6 Estrutura do Documento

Este primeiro capítulo apresentou o contexto no qual este trabalho está inserido, a motivação para realizá-lo, o objetivo a ser atingido e o método a ser seguido. Além deste capítulo, a organização do texto está estruturada em mais cinco capítulos, descritos a seguir:

O Capítulo 2 aborda o referencial teórico, o qual embasa o presente trabalho, com o seguinte conteúdo: aplicações móveis, vazamento de recursos em aplicações móveis e predição de defeitos. O Capítulo 3 apresenta o corpo de conhecimento sobre

identificação de vazamentos de recursos em aplicações móveis, que foi obtido tanto por meio do mapeamento sistemático da literatura técnica. Como também são apresentados os trabalhos relacionados. O Capítulo 4 aborda a pesquisa de opinião com especialistas sobre a relevância das categorias de causa de vazamento identificadas no mapeamento sistemático. O Capítulo 5 apresenta a abordagem proposta e os processos que a compõem. O Capítulo 6 refere-se a um estudo de prova de conceito da abordagem. O Capítulo 7 apresenta o planejamento, a execução e os resultados do estudo de viabilidade da abordagem proposta. Por fim, o Capítulo 8 mostra as considerações finais e os trabalhos futuros.

2

FUNDAMENTOS

Neste capítulo serão apresentados os conceitos relacionados ao desenvolvimento desta pesquisa, tais como aplicações móveis, vazamento de recursos em aplicações móveis e predição de defeitos.

2.1 Aplicações Móveis

As aplicações móveis são um tipo de software projetado para executar em um dispositivo móvel, tais como celular e *tablet*. Essas aplicações podem vir pré-instaladas no dispositivo móvel pelo fabricante ou serem baixadas das lojas de aplicações, como *Apple Store*[1], *Google Play*[2] e *Amazon Store*[3].

Existem três tipos principais de aplicações móveis (SERRANO; HERNANTES; GALLARDO, 2013):

- **Aplicação Nativa**: desenvolvida especificamente para um sistema operacional de um dispositivo móvel, tendo a facilidade de utilização otimizada dos recursos do dispositivo;
- **Aplicação Web**: acessada por meio de um navegador, em sites responsivos que adaptam sua interface de usuário ao dispositivo móvel; e

[1] <https://www.apple.com/br/app-store/>
[2] <https://play.google.com/store>
[3] <https://bityli.com/nULyL>

- **Aplicação Híbrida**: tendo características das duas categorias anteriores, é uma aplicação nativa que consome e disponibiliza páginas webs para o usuário.

Esta pesquisa está focada nas aplicações nativas da plataforma móvel Android, considerada a líder mundial no primeiro trimestre de 2024, com uma participação de mercado de 70,7% (STATISTA, 2024b). Essa plataforma permite a criação de aplicações em módulos. Cada módulo é um ponto de entrada por meio do qual o sistema ou usuário pode acessar a aplicação. Alguns módulos dependem de outros. Existem quatro tipos de módulos: Atividade, Serviço, Receptor de Transmissão e Provedor de Conteúdo. A seguir será mostrada uma descrição resumida desses módulos, conforme (GOOGLE, 2024a).

O módulo Atividade (*Activity*) permite a interação com o usuário por meio de uma interface (em inglês, *User Interface - UI*) e o seu funcionamento consiste em estágios de um ciclo de vida, conforme mostra a Figura 2.1. Para navegar entre os estágios, é fornecido um conjunto básico de sete funções *callback*: *onCreate*(), *onStart*(), *onResume*(), *onPause*(), *onRestart*(), *onStop*() e *onDestroy*(). Cabe destacar o componente *onCreate()*, que é chamado quando o sistema Android "inicia" a atividade principal da aplicação pela primeira vez para o usuário, e o componente *onDestroy()*, que é chamado quando o usuário deixa de interagir com a atividade e decide usar outra aplicação (GOOGLE, 2024b).

O módulo Serviço (*Service*) é semelhante ao módulo Atividade, exceto que são executados em segundo plano sem uma *UI* (GOOGLE, 2024b). O módulo permite que seja executada a lógica da aplicação sem congelar a interface do usuário, sendo útil quando deve baixar um arquivo da Web ou tocar música, por exemplo (HAGOS; HAGOS; ANGLIN, 2018).

Receptor de Transmissão (*Broadcast Receiver*) é um módulo da plataforma Android responsável por escutar as mensagens de transmissão de outras aplicações ou do próprio sistema; por exemplo, ao exibir uma mensagem de aviso quando a bateria atingir menos de 10% de energia (HAGOS; HAGOS; ANGLIN, 2018).

Por último, o Provedor de Conteúdo (*Content Provider*) fornece dados de uma aplicação para outras aplicações mediante solicitação. O principal motivo para utilização

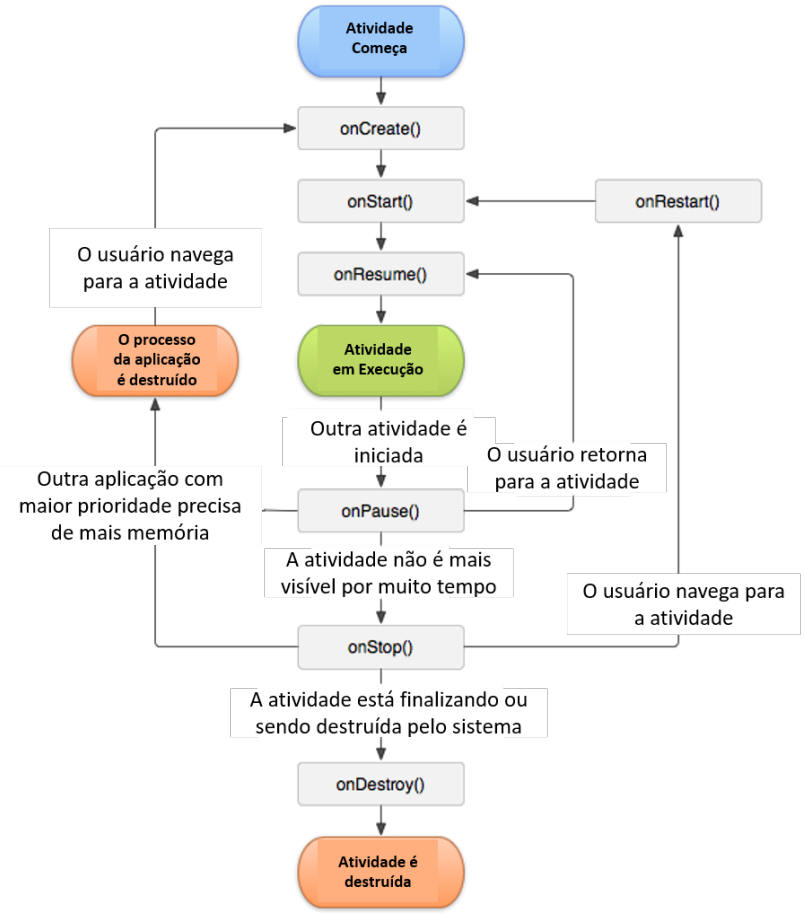


Figura 2.1 – Ilustração simplificada do ciclo de vida da Atividade.
Fonte: (GOOGLE, 2024b)

desse componente é permitir que as aplicações acessem os dados de outras aplicações sem que seja necessário implementar funções de *SQL*, ou seja, os detalhes de acesso a base de dados são completamente ocultados no lado da aplicação cliente (HAGOS; HAGOS; ANGLIN, 2018, pp. 175). Um exemplo do uso de *Content Provider* é a aplicação "Contatos" do Android.

Três dos quatro tipos de módulos – *Activity*, *Service* e *Broadcast Receiver* – são ativados por uma mensagem assíncrona chamada *Intent*. *Intents* vinculam módulos individuais uns aos outros no ambiente de execução servindo como mensageiros que solicitam uma ação de outros módulos, seja o componente pertencente a aplicação ou não (GOOGLE, 2024a). A maioria desses módulos são recursos exclusivos[4] ou recursos

[4] Os recursos exclusivos são aqueles que só podem ser usados por uma aplicação por vez. A câmera é um exemplo de recurso exclusivo, pois se uma aplicação estiver usando outra aplicação não pode usar.

que consomem mais memória/energia do que o geral (GUO et al., 2013). Os recursos devem ser liberados explicitamente pelos desenvolvedores. No entanto, a falta de operação de liberação desses recursos pode causar problemas sérios, como degradação do desempenho ou travamento do sistema (GUO et al., 2013; BHATT; FURIA, 2020).

2.2 Vazamento de Recursos nas Aplicações Móveis

O vazamento de recursos é ocasionado quando uma aplicação não libera recursos que adquiriu durante a sua execução (BHATT; FURIA, 2020). O gerenciamento de recursos de forma adequada não é uma tarefa trivial para desenvolvedores (LIU et al., 2019).

Os recursos dos dispositivos Android e a complexidade de suas aplicações continuam a crescer rapidamente. Esse crescimento apresenta desafios significativos para a correção e desempenho da aplicação (YAN; YANG; ROUNTEV, 2013). Para um melhor entendimento, foram identificadas 14 classes de recursos, com base no mapeamento sistemático executado. A seguir são mostrados alguns exemplos de recursos e os classes.

- **Concorrência**: são as classes de recursos que permitem a execução de tarefas de forma intercalada em um mesmo intervalo de tempo. Alguns exemplos de classes desse tipo de recurso são: *java.lang.Thread*, *java.util.concurrent.Semaphore* e *android.os.Binder*.

- **Conectividade**: são as classes de recursos relacionados a conexão com outros dispositivos, por exemplo, bluetooth (*android.bluetooth.BluetoothAdapter*), USB (*android.hardware.usb*) e Wifi (*android.net.wifi.WifiManager*).

- **Base de Dados**: são as classes de recursos que manipulam os dados da aplicação, por exemplo, *android.database.DatabaseUtils* e *android.database.sqlite.SQLiteDatabase.Cursor*.

- **Arquivos**: são as classes de recursos relacionadas a escrita e leitura de arquivos, por exemplo, *java.io.InputStream*, *java.util.Scanner* e *java.io.FileOutputStream*.

- **Localização**: são as classes de recursos que manipulam a localização do usuário, por exemplo, *android.location.LocationProvider*, *android.location.LocationListener* e *android.location.LocationManager*.
- **Memória**: são as classes de recursos relacionadas ao consumo de memória, por exemplo, *android.app.Activity*, *android.app.Fragment* e *android.content.Context*.
- **Multimídia**: são as classes de recursos que manipulam as mídias do dispositivo, por exemplo, *android.media.MediaPlayer*, *android.media.AudioManager* e *android.hardware.Camera*.
- **Rede**: são as classes de recursos relacionadas a conexão com a internet, por exemplo, *org.apache.http.impl.cliente.AndroidHttpClient*, *java.net.Socket* e *java.net. Network*.
- **Tela**: são as classes de recursos que se relacionam ao o que é exibido na tela do usuário, por exemplo, *android.view.MotionEvent*, *android.text.TextUtils* e *android.appwidget*.
- **Segurança**: são as classes de recursos que tratam da segurança da aplicação, por exemplo, as classes do pacote *android.security*.
- **Sensor**: são as classes de recursos relacionadas ao sensores do dispositivo, por exemplo, *android.hardware.SensorManager*, *android.hardware.Sensor* e *android.os. Vibrator*.
- **Serviço**: são as classes de recursos que auxiliam a aplicação rodar como um serviço, por exemplo, *android.app.Service*.
- **WakeLock**: são as classes de recursos que auxiliam a aplicação ficar em execução mesmo com o celular bloqueado, por exemplo, a classe *android.os.PowerManager. WakeLock*.
- **Energia**: são os recursos relacionados ao consumo de energia.

Infelizmente, os defeitos de gerenciamento de recursos são comuns em programas Android (GUO et al., 2013). Cabe destacar que, além dos defeitos tradicionais,

devem ser considerados os defeitos que visam o consumo excessivo dos recursos limitados disponíveis nos dispositivos móveis (YAN; YANG; ROUNTEV, 2013), por exemplo, reprodutores de mídia, memória, câmera e sensores. O ideal seria que todos os programadores entendessem todos os contratos relevante das *API* (Interface de Programação de Aplicação, em inglês, *Application Programmig Interface*), mas mesmo programadores experientes falham em liberar todos os recursos ao longo de todas as sequências de invocações possíveis de manipuladores de eventos (GUO et al., 2013).

Sendo assim, a identificação desses vazamentos e sua posterior correção possui grande importância para redução dos erros que geram o vazamento de recursos (LIMA; GIUSTI; DIAS-NETO, 2020). Algumas técnicas de Aprendizado de Máquina (AM) estão sendo utilizadas para predição de defeitos, por exemplo, no trabalho de (QIAN; ZHOU, 2016) foi utilizado AM para priorizar casos de teste com maior propensão a revelar vazamentos de recursos, ao passo que, nesta pesquisa, propõe-se identificar vazamento de recursos em componentes das aplicações Android.

2.3 Aprendizado de Máquina

Aprendizado de máquina (AM) pode ser definido como um conjunto de métodos computacionais para otimizar um critério de desempenho utilizando exemplos ou experiência passada (MOHRI; ROSTAMIZADEH; TALWALKAR, 2018). Esses exemplos podem estar na forma de conjuntos de dados com rótulos (por exemplo, com defeito ou sem defeito), caracterizando o chamado aprendizado supervisionado, ou podem ser dados não rotulados, situação típica de aprendizado não supervisionado. Em todos os casos, a experiência passada é representada por um conjunto de dados de treinamento e o objetivo é produzir um modelo capaz de representar conhecimento contido nesses dados, de modo que a qualidade e a quantidade das informações são cruciais para o sucesso das predições feitas pelo modelo (MOHRI; ROSTAMIZADEH; TALWALKAR, 2018).

Existem várias tarefas de AM, tais como classificação, regressão, agrupamento e outras. A de maior relevância para este trabalho é a classificação, na qual os exemplos

são associados a conceitos, denominados classes, e o objetivo é identificar, com base nos dados, a qual classe um exemplo pertence. Por exemplo, a classe pode indicar a presença ou ausência de vazamento em um componente. Nesse caso, o objetivo do classificador é identificar, com base em dados que descrevem o componente, se ele possui ou não um vazamento.

Para a modelagem formal do problema de classificação de duas classes, pode-se considerar um espaço de entrada X e um espaço de saída Y, onde para esta pesquisa tem-se $Y = \{1, 0\}$ ("com vazamento" e "sem vazamento", respectivamente). Supõe-se que existe alguma distribuição de probabilidade desconhecida sobre o espaço $Z = X \times Y$, ou seja, $p(z) = p(\vec{x}, y)$. Assim, o conjunto de treinamento do classificador é composto de uma amostra contendo *n* ensaios dessa distribuição de probabilidade e pode ser notado como $S = \{(\vec{x}_1, y_1), ..., (\vec{x}_n, y_n)\} = \{(\vec{z}_1, ..., \vec{z}_n)\}$. Cada $\vec{x}_i \in X$ é um vetor de entrada dos dados de treinamento, composto por *características* do exemplo $(\vec{x}_i, y_i)$, e y_i é a saída corretamente rotulada desse exemplo (RUSSELL; NORVIG, 2010).

Nesse formalismo, o problema de inferência consiste em encontrar uma função $\zeta : X \to Y$, denominada **função-conceito**, tal que $\zeta(\vec{x}_i) = y_i$. O objetivo ideal seria **induzir** a função-conceito ζ a partir do conjunto de treinamento S. Contudo, esse objetivo geralmente é inalcançável, uma vez que as amostras são normalmente insuficientes para atingir perfeitamente a função-conceito. Deste modo, o objetivo fica sendo encontrar uma boa aproximação $\hat{h}$ tal que $\hat{h} \approx \zeta$. Assim, o mapeamento $\hat{h} : X \to Y$ é uma hipótese ou um modelo sobre ζ (RUSSELL; NORVIG, 2010).

Um algoritmo que classifica um exemplo é denominado modelo de classificação ou, simplesmente, classificador. A seguir são brevemente explicados os principais algoritmos de classificação comumente utilizados na predição de defeito em software (EFENDIOGLU; SEN; KOROGLU, 2019; LI et al., 2019; MANJULA; FLORENCE, 2019).

2.3.1 Máquina de Vetores de Suporte

A Máquina de Vetores de Suporte (em inglês, *Support Vector Machine - SVM*) é um conjunto de métodos de aprendizado usados para classificação, regressão e detecção de

outliers (CORTES; VAPNIK, 1995).

O SVM é um separador linear que utiliza uma função de kernel para mapear os dados para espaços de separação não linear. As funções tradicionais do *kernel* do *SVM* são as seguintes: *linear*, *polinomial*, *RBF* e *sigmoide*. O hiperparâmetro *C*, comum a todas essas funções kernels, compensa a classificação incorreta de exemplos de treinamento pela simplicidade do separador linear mapeado no espaço não linear (SCIKIT-LEARN, 2024b; NOBLE, 2006).

2.3.2 Naive Bayes

Os métodos *Naive Bayes* são um conjunto de algoritmos de aprendizado de máquina com base na aplicação do teorema de Bayes com a suposição "ingênua" de independência condicional entre cada par de características dado o valor da variável classe (SCIKIT-LEARN, 2024c). Ou seja, se determinada raiz é rotulada como "Cenoura" e ela também pode ser descrita como "laranja" e "comprida", o algoritmo não vai levar em consideração a correlação entre esses fatores. Apesar de suas suposições aparentemente simples, os classificadores Bayes funcionaram muito bem em muitas situações do mundo real (SCIKIT-LEARN, 2024c). Maiores detalhes podem ser vistos em (FRANK et al., 2000).

2.3.3 Redes Neurais Profundas

As redes neurais profundas (em inglês, *Deep Neural Networks - DNN*) foram inicialmente inspiradas pelo cérebro e permite que os computadores resolvam tarefas cognitivas nas quais os humanos se destacam (CICHY; KAISER, 2019). As *DNN* são modelos computacionais que consistem em muitas unidades de processamento simples (semelhantes aos neurônios) que funcionam em paralelo e são organizadas em camadas interconectadas (CICHY; KAISER, 2019; LECUN; BENGIO; HINTON, 2015). Maiores detalhes podem ser vistos em (CICHY; KAISER, 2019).

2.3.4 Regressão Logística

A Regressão Logística (em inglês, *Logistic Regression*) é um modelo estatístico que utiliza uma função logística para modelar uma variável dependente binária (SCIKIT-LEARN, 2024a). Uma função logística ou curva logística é uma curva em forma de S comum (curva sigmoide), conforme a Equação 2.1 (SCIKIT-LEARN, 2024a):

$$f(x) = \frac{L}{1 + e^{-k(x-x_0)}} \tag{2.1}$$

Na qual x_0 é o valor do ponto médio do sigmoide para x; L é o valor máximo da curva; e é a base dos logaritmos naturais; e k é a taxa de crescimento logístico ou inclinação da curva. Maiores detalhes podem ser vistos em (LAVALLEY, 2008).

2.3.5 Floresta Aleatória

A Floresta Aleatória (em inglês, *Random Forest*) consiste em um grande número de árvores de decisão[5] individuais que operam como um conjunto, onde cada árvore individual na floresta aleatória exibe uma predição de classe e a classe com mais votos se torna a predição do modelo (KAUR; PANNU; MALHI, 2019). Maiores detalhes podem ser vistos em (BIAU; SCORNET, 2016).

2.3.6 K-Vizinhos mais Próximos

O classificador k-vizinhos mais próximos (em inglês, *K-Nearest Neighbor - KNN*) é um classificador baseado no princípio estatístico dos vizinhos mais próximos. Ele funciona da seguinte forma: dada uma instância de teste Yq, o algoritmo encontra os k (por exemplo, 6) vizinhos mais próximos de Yq no conjunto de treinamento. Em seguida, as classes dos vizinhos são utilizadas para estimar a classe da instância de Yq, por exemplo, a classe Yq pode ser estimada como a moda das classes dos vizinhos encontrados no

[5] Uma árvore de decisão é um modelo preditivo que utiliza uma estrutura em forma de árvore para tomar decisões com base em características dos dados de entrada.

conjunto de treinamento (FUKUNAGA; NARENDRA, 1975). Maiores detalhes podem ser vistos em (KRAMER; KRAMER, 2013).

2.4 Predição de Defeitos

O termo defeito é muitas vezes utilizado de forma genérica. No entanto, existe uma distinção entre erro, defeito e falha. Uma descrição típica para exemplificar as diferenças é a seguinte: O desenvolvedor comete um erro, que pode ocasionar um ou mais defeitos na aplicação, os quais, ao serem executados, podem produzir uma falha (KALINOWSKI, 2008).

Os defeitos de software geralmente incorrem em custos em termos de qualidade e tempo. Além disso, identificar e corrigir defeitos é um dos processos de software mais demorados e caros, sendo praticamente impossível identificar e eliminar todos os defeitos. Mas é possível reduzir a magnitude dos defeitos e seus efeitos adversos nos projetos (RAWAT; DUBEY, 2012). Para isso, podemos utilizar a técnica de predição de defeitos, a qual destina-se a identificar os componentes do código fonte propensos a defeitos, permitindo aos desenvolvedores concentrar esforços em áreas específicas das aplicações. De acordo com (RICKY; PURNOMO; YULIANTO, 2016) e (CATAL; DIRI, 2009), os principais benefícios na utilização dessa técnica são:

- O processo de teste pode ser refinado, com a alocação de mais recursos nos componentes propensos a defeitos, aumentando assim a qualidade da aplicação;
- É possível especificar componentes que requerem refatoração durante a fase de manutenção;
- A predição de defeitos de software fornece estabilidade e contribui com a qualidade da aplicação móvel;
- A predição de defeitos de software pode reduzir o tempo e o esforço gastos no processo de revisão de código.

Para a criação de um modelo de predição de defeitos, alguns passos são seguidos: (1) definir a base de dados; (2) definir as métricas; (3) construir o modelo; e (4) avaliar o desempenho (KAMEI; SHIHAB, 2016). A seguir cada uma das quatro etapas mencionadas serão detalhadas.

2.4.1 Definição da Base de Dados

De acordo com (KAMEI; SHIHAB, 2016), a tarefa de predição de defeitos pode fazer uso de vários tipos de dados de diferentes repositórios, como (1) repositórios de código fonte, que armazenam e registram o código fonte e o histórico de desenvolvimento de um projeto, (2) repositórios de defeitos, que rastreiam os relatórios de defeitos ou solicitações de recursos arquivados para um projeto e seu progresso de resolução e (3) repositórios de lista de e-mails, que rastreiam a comunicação e as discussões entre as equipes de desenvolvimento. Outros repositórios também podem ser aproveitados para predição de defeitos.

2.4.2 Definição das Métricas

As métricas, quando usadas na pesquisa de predição de defeitos de software, são consideradas variáveis independentes, o que significa que são usadas para realizar a predição (KAMEI; SHIHAB, 2016). Algumas métricas para a predição de defeitos são complexidade, palavras-chave, mudanças e dependências estruturais. Ao combinar rótulos e recursos de instâncias, podemos produzir um conjunto de treinamento a ser usado por uma técnica de aprendizado de máquina para construir um modelo de predição (RAWAT; DUBEY, 2012).

2.4.3 Construção do Modelo

Neste trabalho optou-se por explorar os métodos de aprendizado de máquina porque é uma forma de gerar um modelo preditivo sem intervenção humana a partir dos dados.

Vale ressaltar que o modelo elaborado é um modelo de classificação e que classificação tem como objetivo fazer mapeamento de atributos de entrada para um atributo categórico (classe). Os seguintes classificadores foram escolhidos para serem testados no problema de pesquisa desta tese: *SVM*, *Naive Bayes*, *DNN*, *Regressão Logística*, *Floresta Aleatória* e *KNN*. Esses classificadores foram escolhidos por serem muito utilizados para predição de defeitos (EFENDIOGLU; SEN; KOROGLU, 2019; LI et al., 2019; MANJULA; FLORENCE, 2019; YUCALAR et al., 2019).

2.4.4 Avaliação de Desempenho

A avaliação do desempenho é uma etapa fundamental na resolução de problemas por aprendizado de máquina. Normalmente essa avaliação é feita de maneira empírica, na qual os dados são divididos em um ou mais conjuntos de treinamento e um ou mais conjuntos de teste. O objetivo é simular o modelo quando submetido a dados nunca vistos antes. Assim, o modelo é construído com os exemplos de treinamento e só tem acesso aos exemplos de teste no momento da avaliação de desempenho.

Uma estratégia simples para testar algoritmos de aprendizado de máquina é o *holdout*, na qual uma fração (normalmente 20% a 30%) dos dados originais é "deixada de fora" do treinamento. Para a avaliação dos testes algumas medidas de desempenho são comumente usadas, com o intuito de avaliar a eficácia dos modelos de predição de defeitos, incluindo acurácia (Equação 2.2), precisão (Equação 2.3), revocação (Equação 2.4), área sob a curva ROC (*Area Under the Receiver Operating Characteristic Curve* ou *ROC AUC*) (CLARK; WEBSTER-CLARK, 2008) e *F-measure* (Equação 2.5) (EFENDIOGLU; SEN; KOROGLU, 2019; LI et al., 2019; MANJULA; FLORENCE, 2019). Essas métricas são calculadas com base no número de "verdadeiros" e "falsos" "positivos" ou "negativos" verificados no conjunto de teste. No contexto deste trabalho, um "positivo" é um componente que possui um defeito, enquanto "negativo" é um componente que não possui defeito. Um "verdadeiro" é quando o modelo identifica corretamente se o componente possui defeito ou não; enquanto um "falso" é quando o modelo não consegue identificar corretamente. Normalmente esses elementos são dispostos em uma

tabela, com estrutura semelhante ao mostrado na Tabela 2.1.

Tabela 2.1 – Matriz de Confusão.

		Predição	
		Sem Defeito	Com Defeito
Real	Sem Defeito	Verdadeiro Negativo (VN)	Falso Positivo (FP)
	Com Defeito	Falso Negativo (FN)	Verdadeiro Positivo (VP)

A tabela é preenchida numericamente, com cada elemento indicando o número de casos ocorridos no conjunto de teste. Com isso é possível calcular as métricas de desempenho supracitadas, usando as equações a seguir:

$$acuracia = \frac{VP + VN}{VP + VN + FP + FN} \tag{2.2}$$

$$precisao = \frac{VP}{VP + FP} \tag{2.3}$$

$$revocacao = \frac{VP}{VP + FN} \tag{2.4}$$

$$f\_measure = \frac{2 \times (revocacao \times precisao)}{(revocacao + precisao)} \tag{2.5}$$

Diferentes medidas refletem diferentes aspectos do desempenho do classificador. Algumas medidas se aplicam a classes específicas, como precisão e revocação. Em problemas nos quais existem apenas duas classes (classificação binária), essa classe "de interesse" é normalmente chamada a classe positiva. Neste trabalho, a classe positiva é "com defeito".

A acurácia é a capacidade geral de um classificador classificar qualquer exemplo, pois é uma medida que não discrimina nem favorece nenhuma classe. Trata-se de uma medida de desempenho geral que é frequentemente usada para comparar diferentes classificadores, mas que pode ser insuficiente em problemas desbalanceados, nos quais uma classe possua poucos exemplos diante de outras. A precisão mede a probabilidade de uma classe prevista como defeito de estar realmente com defeito, enquanto a revocação denota a proporção de componentes defeituosos corretamente previstos (no contexto deste trabalho). Em outras palavras, quanto maior a precisão, maior a confiança

de que um componente realmente está defeituoso e, quanto maior a revocação, maior a confiança de que o classificador não está "deixando passar" componentes defeituosos. Já ROC AUC é a capacidade de um classificador de distinguir entre as classes sob diferentes condições de "certeza". Ela é obtida com classificadores que produzem um escore de classificação (por exemplo, quando a saída do modelo é a probabilidade de um componente possuir defeito). Variando-se o limiar de certeza necessário para que um exemplo $\vec{x_i}$ seja classificado como positivo, pode-se elaborar uma curva que descreve o modelo quando a sua saída é menos ou mais conservadora. A área sob essa curva é um resumo do desempenho geral do classificador. Por fim, a *F-measure* é a média harmônica de precisão e revocação e é indicada quando se deseja aproximar um equilíbrio entre essas duas medidas.

2.5 Considerações Finais

Neste capítulo foram considerados os principais conceitos sobre o ciclo de vida das aplicações móveis e suas características. Além disso, foi descrita a problemática do vazamento de recursos presentes em eventos de execução das aplicações.

Logo, são apresentados sobre predição de defeitos, as fases de desenvolvimento de um modelo e os principais classificadores de aprendizado de máquina, que são avaliados visando atender a hipótese desta Tese. No próximo capítulo será apresentado um mapeamento sistemático sobre vazamentos de recursos em aplicações móveis.

3

MAPEAMENTO SISTEMÁTICO

Na literatura científica não foi encontrada uma análise sobre vazamentos de recursos em aplicações móveis. Por isso, optou-se por realizar esse Mapeamento Sistemático (MS), o qual foi realizado conforme as diretrizes especificadas em (PETERSEN; VAKKALANKA; KUZNIARZ, 2015) e (KITCHENHAM; CHARTERS, 2007), que possui os seguintes passos principais:

- Definição de objetivo e questões de pesquisa;
- Identificação dos estudos;
- Definição da estratégia de extração de dados;
- Execução do mapeamento sistemático.

3.1 Objetivo e Questões de Pesquisa

O objetivo deste estudo foi identificar, analisar e sintetizar publicações científicas que tratam de identificação de vazamento de recursos em aplicações móveis. Com base no objetivo de pesquisa, foi formulada a questão de pesquisa principal (RQ, do inglês *Research Question*) (RQ 1). Para extrair informações detalhadas, a questão foi dividida em várias subquestões, conforme descrito abaixo.

RQ 1 - mapeamento sistemático: Qual é o espaço de pesquisa da literatura científica em identificação de vazamento de recursos em aplicações móveis? Para responder

a essa pergunta, foram propostas as seguintes subquestões de pesquisa:

- **RQ 1.1 - tipos de contribuição:** Quantos artigos apresentam métodos/técnicas, ferramentas, modelos, métricas ou processos para a identificação de vazamento de recursos? O documento de orientação de MS descrito em (PETERSEN et al., 2008) propõe os tipos de contribuições acima referidos. A resposta a esta RQ nos permite avaliar se a comunidade, como um todo, teve mais foco no desenvolvimento de novas técnicas de identificação ou mais foco no desenvolvimento de novas ferramentas de identificação;

- **RQ 1.2 - tipos de pesquisa:** Quais tipos de pesquisa são utilizados nos trabalhos nessa área? O documento de orientação de MS descrito em (PETERSEN et al., 2008) propõe os seguintes tipos de pesquisa: proposta de solução, pesquisa de validação, pesquisa de avaliação e relatos de experiência. A lógica por trás desta RQ é conhecer a maturidade do campo no uso de abordagens empíricas;

- **RQ 1.3 - classes de recurso:** Quais são as classes de recursos identificados como tendo vazamentos? Alguns trabalhos detectam vazamento em apenas um recurso, como o *WakeLock*. Outros detectam em vários recursos como câmera, sensor e localização;

- **RQ 1.4 – categoria de causas de vazamento:** Quais são as categorias de causas de vazamento? Essa RQ ajudará a entender quais são as causas de vazamento de recurso.

3.2 Identificação dos Estudos

A *string* de busca foi modelada a partir da estrutura analítica PICO (do inglês *Population*, *Intervention*, *Comparison* e *Outcomes*), conforme as recomendações descritas em (PETERSEN; VAKKALANKA; KUZNIARZ, 2015). Para aprimorar a *string*, foram utilizados 8 artigos de controle (XU; WEN; QIN, 2018; BANERJEE et al., 2017; JIANG et al., 2017; ZHANG; WU; ROUNTEV, 2016; WU et al., 2016b; GUO et al., 2013; KIM; CHA, 2013; LIU; XU, 2013), tendo sido identificados os sinônimos das palavras-chave. Também

foram realizados ciclos de execução da *string* de busca para refinamentos na biblioteca digital *Scopus*[1]. A referida *string* é mostrada na Tabela 3.1.

Tabela 3.1 – A *string* na estrutura PICO para o mapeamento realizado.

População	(app OR apps OR android OR "mobile application" OR "mobile applications")
Intervenção	(leak OR leaks OR "no sleep" OR "no-sleep" OR (acqui* AND releas*)) AND (resource* OR context* OR memory OR energy OR bug* OR wakelock OR wakelocks OR wifi OR "wi-fi" OR sensor OR sensors)
Comparação	Não se aplica
Resultados	Sem restrições

A busca de estudos utilizando a *string* definida foi realizada nas bibliotecas digitais *Scopus*, *IEEE Xplore*[2], *Science Direct*[3], *Engineering Village*[4], *Web of Science*[5] e *ACM DL*[6]. Essas bibliotecas digitais foram selecionadas com base na experiência relatada em (PETERSEN; VAKKALANKA; KUZNIARZ, 2015). Os estudos retornados pelas bibliotecas digitais serão avaliados e/ou analisados somente se atenderem a todos os critérios de inclusão, sendo que os critérios são:

- Estudos relacionados a identificação de vazamento de recursos em aplicações móveis;
- Artigos disponíveis na web ou por meio de contato com os autores;
- Artigos escritos na língua inglesa;
- Artigos não duplicados.

3.2.1 Esquema de Classificação

Um esquema de classificação é derivado de uma análise cuidadosa dos estudos primários para ajudar na categorização dos artigos. A classificação para RQ 1.1 foi baseada

[1] https://www.scopus.com/search/form.uri?display=basic
[2] https://ieeexplore.ieee.org/Xplore/home.jsp
[3] https://www.sciencedirect.com/
[4] https://www.engineeringvillage.com/search/quick.url
[5] http://www.webofknowledge.com
[6] https://dl.acm.org

em (SHAHROKNI; FELDT, 2013), onde um *framework* é um método detalhado que tem um propósito amplo e se concentra em múltiplas questões ou áreas de pesquisa, um *método* geralmente tem uma abordagem com um objetivo específico e uma questão ou propósito de pesquisa restrito, um *modelo* fornece uma classificação ou modelo abstrato de um tópico e problema, em vez de uma maneira específica e tangível de resolver um problema específico, a *ferramenta* é quando é fornecido uma ferramenta, *avaliação* inclui artigos que avaliam um conceito já publicado e não introduzem novos conceitos ou soluções, a *métrica* fornece diretrizes sobre como medir aspectos da aplicação. Também foi adicionada a categoria *Base de dados*, que se refere a uma coleção organizada de informações composta por registros relacionados.

A RQ 1.2 (tipos de pesquisa) foi organizada de acordo com (PETERSEN et al., 2008), onde artigos que possuem apenas exemplos e uma boa linha de argumentação são categorizados como *Proposta de Solução*; trabalhos cujas técnicas ainda não foram implementadas na prática, ou seja, apenas com experimentos realizados em laboratório são classificados como *Pesquisa de Validação*; se a técnica for implementada na prática e possuir avaliação técnica mostrando como o estudo foi conduzido, quais os benefícios e desvantagens (incluindo a identificação de problemas na indústria), é classificada como *Pesquisa de Avaliação*; artigos que relatam apenas aplicações ou experiências na prática, baseadas na experiência pessoal do autor, são classificados como *Relatos de Experiência*.

A RQ 1.3 (classes de recursos) foi estruturada em 14 classes de recursos (*concorrência*, *conectividade*, *base de dados*, *arquivos*, *localização*, *memória*, *multimídia*, *rede*, *tela*, *segurança*, *sensor*, *serviço*, *wakelock* e *energia*) baseado em (LIU et al., 2019; GUO et al., 2013; YAN; YANG; ROUNTEV, 2013), descritos na seção 2.2.

Para a construção do esquema de classificação RQ 1.4, foi criada uma versão inicial a partir dos estudos de controle, posteriormente evoluída durante a extração dos dados, por meio dos atributos extraídos e etapas iterativas de refinamento. Adicionar novas categorias ou mesclar as existentes. Portanto, foram definidas as seguintes 10 categorias: *Vazamento completo*, *Vazamento em caminhos normais*, *Vazamento em caminhos excepcionais*, *Vazamento em caminhos irregulares*, *Vazamento causado por condição de corrida*, *Vazamento causado por recurso não utilizado*, *Vazamento causado pela complexidade do ciclo de*

vida da aplicação, *Vazamento causado por alocação e desalocação imprópria*, *Vazamento causado por liberação inadequada de recursos em loop* e *Vazamento causado pelo Android SDK*.

3.3 Estratégia de Extração de Dados

Para cada artigo selecionado, serão extraídos os dados necessários para responder às questões de pesquisa propostas para este estudo. Os dados que serão extraídos estão descritos na Tabela 3.2.

Tabela 3.2 – Formulário de Extração de Dados.

RQ	**Item**	**Descrição**
RQ 1		
RQ 1.1	Tipos de contribuição	Framework, método, modelo, ferramenta, avaliação, métrica ou base de dados
RQ 1.2	Tipos de pesquisa	Proposta de solução, pesquisa de validação, pesquisa de avaliação e relatos de experiência
RQ 1.3	Classes de recurso	Quais recursos o artigo pode identificar com vazamento
RQ 1.4	Causa do vazamento	Quais as causas dos vazamentos de recursos

3.4 Execução do Estudo do Mapeamento Sistemático

Foi executada a busca nas bibliotecas digitais selecionadas utilizando a *string* definida e houve um retorno de 1.760 artigos, conforme mostrado na Tabela 3.3. Após isso, realizou-se a remoção dos artigos duplicados e aplicou-se o *filtro 1* (seleção baseada no título, resumo e palavras-chave). Depois, conduziu-se o *filtro 2* (análise completa do artigo). Nos artigos selecionados no *filtro 2*, deu-se a extração das informações descritas na Tabela 3.2. Com os dados extraídos, sucedeu-se à sumarização e análise, cujos resultados serão mostrados nas próximas sessões.

Tabela 3.3 – Número de artigos selecionados.

Biblioteca Digital	**Artigos Recuperados**	**1º Filtro**	**2º Filtro**
Scopus	427	25	14
IEEE Xplore	337	9	1
Science Direct	34	1	0
ACM DL	217	13	3
Web of Science	258	26	13
Engineering Village	487	48	26
Total	**1760**	**122**	**57**

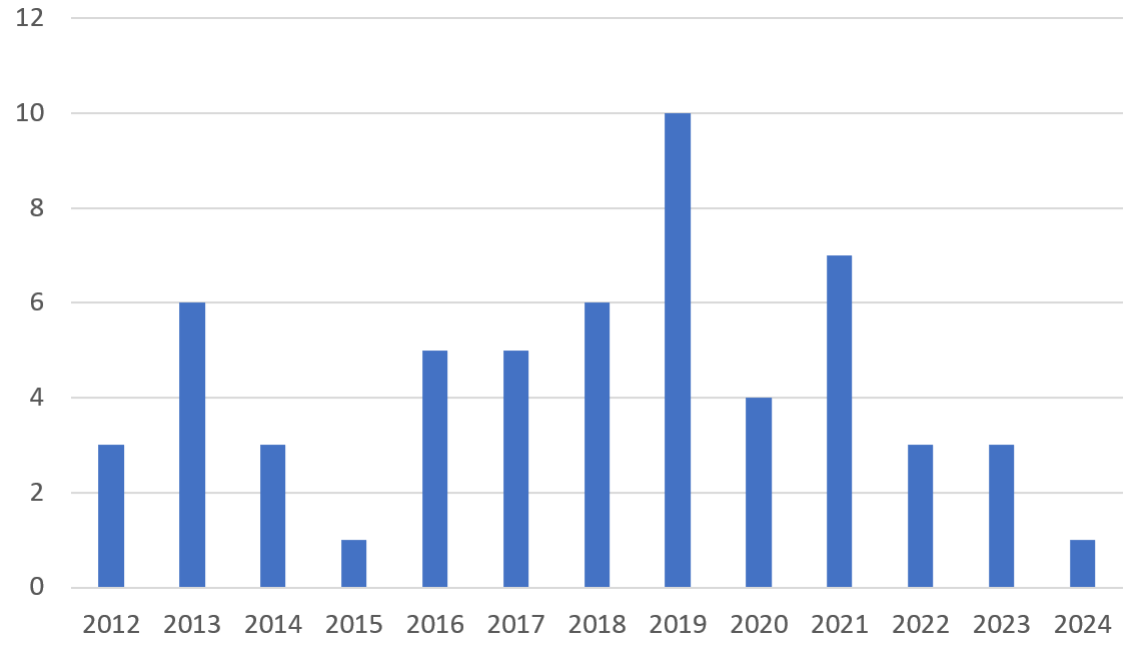


Figura 3.1 – Publicações por ano.

3.5 Resultados do Mapeamento Sistemático

Os 57 artigos selecionados foram publicados entre 2012 e 2024. Na Figura 3.1, percebe-se que em 2013 (segundo ano com publicações) houve um aumento no número de artigos, seguido de uma declínio nos anos de 2014 e 2015. Nota-se também que a mesma quantidade de artigos publicados em 2013 foi alcançada em 2018 e ampliada em 2019. Vale lembrar que a busca dos artigos nas bibliotecas digitais ocorreu em julho de 2024, portanto, é possível que alguns artigos de 2024 não tenham sido incluídos.

Na Figura 3.2 é mostrada a quantidade de artigos por tipo de local da publicação. Pode-se observar que a maioria dos autores (59,6%) publicou seus artigos em

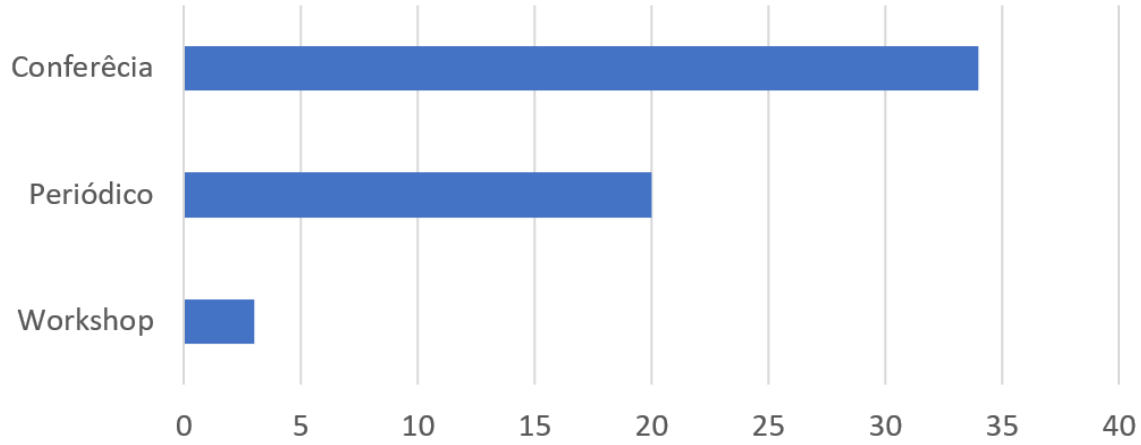


Figura 3.2 – Publicações por tipo de local.

conferências, 35,1% em periódicos (*journal*) e, por último, 5,3% em *workshop*.

Tabela 3.4 – Principais locais de publicação.

Local	**Acrônimo**	**Qtd**
International Conference on Software Engineering	ICSE	3
Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)	LNCS	3
International Conference on Automated Software Engineering	ASE	3
International Computer Software and Applications Conference	COMPSAC	2
ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering	ESEC/FSE	2
IEEE Access	IEEE	2
Journal of Computer Science and Technology	JCST	2
International Conference on Mobile Systems, Applications, and Services	MobiSys	2
Software - Practice and Experience	SPE	2
IEEE Transactions on Software Engineering	TSE	2
International Symposium on Software Testing and Analysis	ISSTA	2

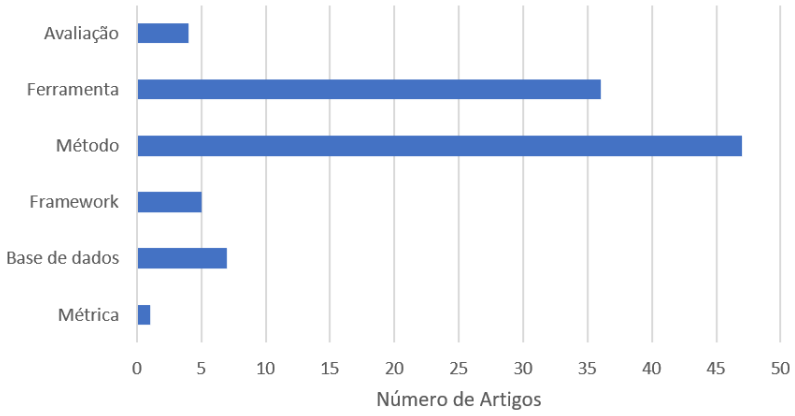
Para classificar os locais de publicação, foi usado como métrica o número de artigos publicados em cada local. A classificação dos principais locais com pelo menos dois artigos é mostrado na Tabela 3.4. Existem nove locais nessa lista: cinco periódicos e quatro conferências, dentre os quais estão alguns dos principais meios de publicação na área de engenharia de software. Por exemplo, as duas conferências com mais artigos (três artigos, 5,3%) são o ICSE e o ASE, que são um dos dois principais eventos da área.

A seguir serão apresentados os resultados obtidos no estudo de mapeamento

sistemático (RQ 1).

3.5.1 RQ 1.1 - Tipos de contribuição

A Figura 3.3 mostra a distribuição dos trabalhos por tipo de contribuição para todos os 57 trabalhos incluídos neste estudo. Com base em suas contribuições, alguns trabalhos foram classificados em mais de um tipo. Por exemplo, (SONG; ZHANG; HUANG, 2019) fizeram duas contribuições: (1) desenvolveram uma técnica de análise estática que encontra defeitos de ineficiência no uso de serviços e (2) implementaram essa técnica como uma ferramenta chamada *ServDroid*. Ainda na Figura 3.3, é indicado que a proposição de métodos foi o item que mais atraiu publicações, com 47 artigos (cerca de 82,5%) focados nesse aspecto. Em seguida, cerca de 63,2% dos artigos selecionados (36 de 57) propuseram novas ferramentas.

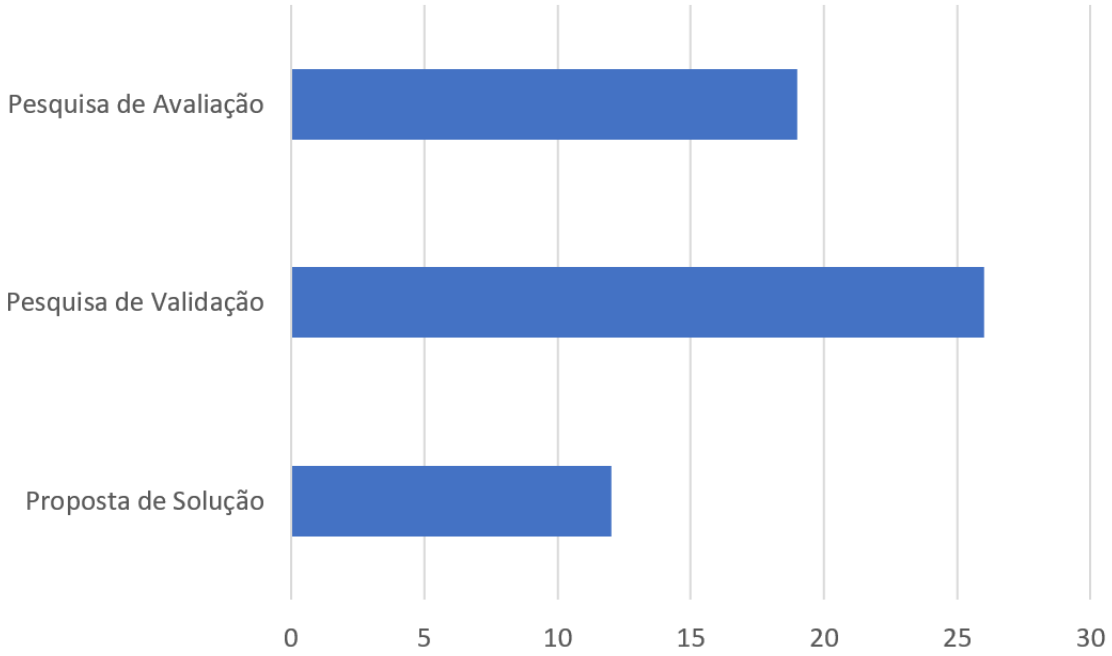


Avaliação: Liu et al. (2019), Song, Wedyan e Jararweh (2021), Palomba et al. (2019), Nguyen, Vu e Nguyen (2019)
Ferramenta: Pathak et al. (2012), Jiang et al. (2017), Vilk e Berger (2018), Guo et al. (2013), Ma et al. (2018), Ferrari et al. (2015), Zein, Salleh e Grundy (2017), Zhang et al. (2012), Wu et al. (2016a), Banerjee et al. (2017), Jun et al. (2017), Wu et al. (2016b), Vekris et al. (2012), Song, Zhang e Huang (2019), Hoshieah et al. (2019), Toffalini, Sun e Ochoa (2019), Xu, Wen e Qin (2018), Wu, Yang e Rountev (2016), Chang (2014), Yan, Yang e Rountev (2013), Kim e Cha (2013), Liu e Xu (2013), Chen, Li e Zhou (2017), Ghanem e Zein (2020), Pereira et al. (2022), Lu et al. (2022), Liu et al. (2021), Amalfitano et al. (2020), Wu et al. (2020), Bhatt e Furia (2022), Khan et al. (2021b), Riganelli, Micucci e Mariani (2019a), Nguyen, Vu e Nguyen (2020)
Método: Hall, Nataraj e Kim (2018), Qian e Zhou (2016), Zein, Salleh e Grundy (2017), Wu et al. (2016b), Vekris et al. (2012), Hoshieah et al. (2019), Xu, Wen e Qin (2018), Wu, Yang e Rountev (2016), Shahriar, North e Mawangi (2014), Chang (2014), Araujo et al. (2013), Ghanem e Zein (2020), Naseer, Nadeem e Zaman (2021), Amalfitano et al. (2020), Khan et al. (2021b), Riganelli, Micucci e Mariani (2019a), Nguyen, Vu e Nguyen (2020), Pathak et al. (2012), Jiang et al. (2017), Vilk e Berger (2018), Ma et al. (2018), Ferrari et al. (2015), Zhang et al. (2012), Jun et al. (2017), Song, Zhang e Huang (2019), Toffalini, Sun e Ochoa (2019), Zhang, Wu e Rountev (2016), Alam et al. (2014), Yan, Yang e Rountev (2013), Pereira et al. (2022), Lu et al. (2022), Liu et al. (2021), Khan et al. (2021a), Sakhare, Kim e Hamdi (2019), Wu et al. (2020), Bhatt e Furia (2022), Xia et al. (2013), Guo et al. (2013), Wu et al. (2016a), Santhanakrishnan, Cargile e Olmsted (2016), Kim e Cha (2013), Ahn (2019), Le (2021)
Framework: Banerjee et al. (2017), Liu e Xu (2013), Chen, Li e Zhou (2017), Saju et al. (2021), Sehgal et al. (2020)
Base de dados: Liu et al. (2019), Riganelli, Micucci e Mariani (2019b), Saju et al. (2021), Sehgal et al. (2020), Palomba et al. (2019), Nguyen, Vu e Nguyen (2019), Nguyen, Vu e Nguyen (2020)
Métrica: Vilk e Berger (2018)

Figura 3.3 – Tipos de contribuições (veja as referências acima mencionadas para obter mais informações).

3.5.2 RQ 1.2 - Tipos de pesquisa

A Figura 3.4 mostra a distribuição dos artigos selecionados por tipo de pesquisa. Pode-se observar que os trabalhos sobre identificação de vazamento de recursos são, em sua maioria, relacionados a pesquisas de validação com 26 artigos (45,6%), indicando o nível relativamente alto de maturidade dessa comunidade. Em seguida estão os trabalhos que se concentram em pesquisas de avaliação com 19 artigos (33,3%), o que sugere que a comunidade tem uma atenção especial às abordagens empíricas, tais como as referidas espécies de pesquisa. Já as pesquisas que descrevem propostas de soluções totalizam 12 trabalhos (21,1%).



Pesquisa de Avaliação: Pathak et al. (2012), Liu et al. (2019), Wu, Wang e Rountev (2018), Vilk e Berger (2018), Guo et al. (2013), Ma et al. (2018), Zein, Salleh e Grundy (2017), Zhang et al. (2012), Banerjee et al. (2017), Jun et al. (2017), Wu et al. (2016b), Vekris et al. (2012), Song, Zhang e Huang (2019), Hoshieah et al. (2019), Toffalini, Sun e Ochoa (2019), Xu, Wen e Qin (2018), Kim e Cha (2013), Liu e Xu (2013)
Pesquisa de Validação: Qian e Zhou (2016), Jiang et al. (2017), Ferrari et al. (2015), Wu et al. (2016a), Zhang, Wu e Rountev (2016), Wu, Yang e Rountev (2016), Alam et al. (2014), Shahriar, North e Mawangi (2014), Chang (2014), Araujo et al. (2013), Yan, Yang e Rountev (2013), Ahn (2019), Chen, Li e Zhou (2017), Ghanem e Zein (2020), Pereira et al. (2022), Lu et al. (2022), Naseer, Nadeem e Zaman (2021), Liu et al. (2021), Khan et al. (2021a), Amalfitano et al. (2020), Wu et al. (2020), Bhatt e Furia (2022), Khan et al. (2021b), Riganelli, Micucci e Mariani (2019a), Nguyen, Vu e Nguyen (2020)
Proposta de Solução: Hall, Nataraj e Kim (2018), Xia et al. (2013), Riganelli, Micucci e Mariani (2019b), Santhanakrishnan, Cargile e Olmsted (2016), Le (2021), Song, Wedyan e Jararweh (2021), Saju et al. (2021), Sakhare, Kim e Hamdi (2019), Sehgal et al. (2020), Palomba et al. (2019), Nguyen, Vu e Nguyen (2019)

Figura 3.4 – Tipos de pesquisa (veja as referências acima mencionadas para obter mais informações).

3.5.3 RQ 1.3 - Classes de recurso

Durante a análise dos 57 trabalhos, foram identificadas 14 classes de recursos. Esses recursos estão listados nas colunas da Tabela 3.5, onde são mostradas as classes de recursos identificados para cada artigo. O artigo que trabalhou com a maior quantidade

de classes de recursos foi (LIU et al., 2019) com 9 (64,3%) recursos, seguido do trabalho de (GUO et al., 2013) com 8 (57,1%) classes de recursos. A classe de recurso mais trabalhada nos artigos foi *Energia*, com 29 (50,9%) trabalhos. Em seguida, tem-se o recurso *WakeLock* com 21 (36,8%) artigos.

Tabela 3.5 – Classes de recurso.

Artigo	Concorrência	Conectividade	Base de Dados	Arquivos	Localização	Memória	Multimídia	Rede	Tela	Segurança	Sensor	Serviço	WakeLock	Energia
Liu et al. (2019)	x	x	x	x	x		x	x	x				x	x
Guo et al. (2013)		x			x	x	x	x	x		x		x	x
Riganelli, Micucci e Mariani (2019b)		x	x		x	x	x		x				x	x
Hoshieah et al. (2019)		x	x	x	x		x	x			x			
Xu, Wen e Qin (2018)		x		x	x		x	x					x	x
Banerjee et al. (2017)		x			x		x				x		x	x
Wu et al. (2016b)		x			x		x				x		x	x
Riganelli, Micucci e Mariani (2019a)	x	x			x		x				x		x	
Jiang et al. (2017)		x			x	x	x				x			x
Lu et al. (2022)			x	x			x	x			x			
Bhatt e Furia (2022)		x			x		x				x		x	
Pathak et al. (2012)					x		x				x		x	x
Zein, Salleh e Grundy (2017)					x		x	x			x			
Pereira et al. (2022)			x				x						x	x
Naseer, Nadeem e Zaman (2021)		x			x								x	x
Nguyen, Vu e Nguyen (2019)			x	x			x		x					
Nguyen, Vu e Nguyen (2020)			x	x			x	x						
Ferrari et al. (2015)				x					x	x				x
Ghanem e Zein (2020)		x			x		x							
Le (2021)	x												x	x
Sakhare, Kim e Hamdi (2019)	x												x	x
Hall, Nataraj e Kim (2018)	x												x	x
Zhang et al. (2012)		x												x
Wu et al. (2016a)													x	x
Vekris et al. (2012)													x	x
Song, Zhang e Huang (2019)												x		x
Zhang, Wu e Rountev (2016)	x					x								
Wu, Yang e Rountev (2016)					x									x
Alam et al. (2014)													x	x
Yan, Yang e Rountev (2013)	x					x								
Kim e Cha (2013)													x	x
Liu e Xu (2013)			x	x										
Chen, Li e Zhou (2017)													x	x
Liu et al. (2021)	x					x								
Khan et al. (2021a)													x	x
Khan et al. (2021b)													x	x
Xia et al. (2013)						x								x
Qian e Zhou (2016)						x								
Vilk e Berger (2018)						x								
Ma et al. (2018)												x		
Jun et al. (2017)						x								
Toffalini, Sun e Ochoa (2019)						x								
Santhanakrishnan, Cargile e Olmsted (2016)						x								
Shahriar, North e Mawangi (2014)						x								
Chang (2014)						x								
Araujo et al. (2013)						x								
Ahn (2019)						x								
Song, Wedyan e Jararweh (2021)														x
Saju et al. (2021)														x
Sehgal et al. (2020)														x
Amalfitano et al. (2020)						x								
Wu et al. (2020)											x			
Palomba et al. (2019)														x
Wang et al. (2024)	x	x	x	x			x	x						
Cui et al. (2023)				x		x		x						
Nanavati et al. (2024)						x								
Shahoor et al. (2023)						x		x						

3.5.4 RQ 1.4 – Categoria de causas de vazamento

As causas de vazamento extraídas de cada artigo foram agrupadas com o objetivo de identificação de categorias, as quais foram definidas em 10, sendo elas: *Vazamento completo, Vazamento em caminhos normais, Vazamento em caminhos excepcionais, Vazamento em caminhos irregulares, Vazamento causado por condição de corrida, Vazamento causado por recurso não utilizado, Vazamento causado pela complexidade do ciclo de vida da aplicação, Vazamento causado por alocação e desalocação imprópria, Vazamento causado por liberação inadequada de recursos em loop e Vazamento causado pelo Android SDK*. Cada uma das categorias de vazamento será explicada abaixo.

Código 3.1 – Exemplo de vazamento completo (com defeito).

```
1  static OtpType getType(String email) {
2    Cursor cursor = DATABASE.query(TABLE_NAME, null, EMAIL_COLUMN + "= ?",
3        new String[] {email}, null, null, null);
4    if (cursor != null && cursor.getCount() > 0) {
5      cursor.moveToFirst();
6      Integer value = cursor.getInt(cursor.getColumnIndex(TYPE_COLUMN));
7      return OtpType.getEnum(value);
8    }
9    return null;
10 }
```

Vazamento completo (*Complete leak*): Os desenvolvedores deixam completamente de liberar os recursos adquiridos após o seu uso. Por exemplo, no Código 3.1 é mostrado um trecho de código da aplicação *Google Authenticator*, onde na linha 2 é inicializado um objeto do tipo *Cursor*. Nas linhas 5 e 6, este é utilizado e em seguida não é liberado. O código com a correta liberação do recurso, implementada por um dos desenvolvedores da aplicação, é mostrado no Código 3.2. Neste caso, o código de utilização do recurso foi colocado no bloco *try* e a chamada do método (linha 10), que faz a liberação do recurso, foi incluída na cláusula *finally*, a fim de que o mesmo seja liberado tanto quando na ocasião em que o código executar sem falha quanto na que ocorrer uma falha.

Código 3.2 – Exemplo de vazamento completo (correto).

```
1  static OtpType getType(String email) {
2    Cursor cursor = getAccount(email);
3    try {
4      if (!cursorIsEmpty(cursor)) {
5        cursor.moveToFirst();
6        Integer value = cursor.getInt(cursor.getColumnIndex(TYPE_COLUMN));
7        return OtpType.getEnum(value);
8      }
9    } finally {
10     tryCloseCursor(cursor);
11   }
12   return null;
13 }
```

Código 3.3 – Exemplo de vazamento em caminhos normais (com defeito).

```
1   private void realRun() {
2       synchronized (createLock) {
3           if (mTask != this) {
4               Log.w(THIS_FILE, "  unexpected task: " + mNetworkType + (mConnected ?
                    " CONNECTED" : "DISCONNECTED"));
5               return;
6           }
7           mTask = null;
8           Log.d(THIS_FILE, " deliver change for " + mNetworkType + (mConnected ? "
                CONNECTED" : "DISCONNECTED"));
9           // onConnectivityChanged(mNetworkType, mConnected);
10          dataConnectionChanged(mNetworkType, true);
11          sipWakeLock.release(this);
12      }
13  }
```

Vazamento em caminhos normais (*Leak on normal paths*): Os recursos são liberados somente em alguns caminhos do código. Por exemplo, o Código 3.3 mostra o trecho de código extaído da aplicação *CSipSimple*[7], no qual o recurso no objeto *sipWakeLock* é liberado na linha 11. No entanto, caso a condição da linha 3 seja verdadeira, o recurso não será liberado, pois há uma instrução *return* que encerra a execução da função corrente. No Código 3.4 é mostrada a correção do código implementada pelo autor da aplicação[8], em que a instrução de liberação do recurso é implementada dentro

[7] <https://github.com/r3gis3r/CSipSimple/blob/88d62bc9510809ab6d9750b0a0b761ef08c85948/src/com/csipsimple/service/SipService.java>

[8] <https://github.com/r3gis3r/CSipSimple/blob/686620ba55f584c16388f4c1052a336ee6896bcb/src/com/csipsimple/service/SipService.java>

do *if* na linha 5, ao passo que a outra instrução de liberação permanece na linha 12.

Código 3.4 – Exemplo de vazamento em caminhos normais (correto).

```
1  private void realRun() {
2      synchronized (createLock) {
3          if (mTask != this) {
4              Log.w(THIS_FILE, "  unexpected task: " + mNetworkType + (mConnected ?
                   " CONNECTED" : "DISCONNECTED"));
5              sipWakeLock.release(this);
6              return;
7          }
8          mTask = null;
9          Log.d(THIS_FILE, " deliver change for " + mNetworkType + (mConnected ? "
               CONNECTED" : "DISCONNECTED"));
10         // onConnectivityChanged(mNetworkType, mConnected);
11         dataConnectionChanged(mNetworkType, true);
12         sipWakeLock.release(this);
13     }
14 }
```

Código 3.5 – Exemplo de vazamento em caminhos excepcionais (com defeito).

```
1  @Override
2  protected Boolean doInBackground(Void... params) {
3      try {
4          mImportResults = StorageImporter.importSettings(mContext, mInputStream,
5                  mEncryptionKey, mIncludeGlobals, mAccountUuids, mOverwrite);
6      } catch (StorageImportExportException e) {
7          Log.w(K9.LOG_TAG, "Exception during export", e);
8          return false;
9      }
10     return true;
11 }
```

Vazamento em caminhos excepcionais (*Leak on exceptional paths*): Os recursos do sistema não são liberados caso ocorram exceções. Por exemplo, no trecho de código extraído da aplicação *k-9 Mail*[9], mostrado no Código 3.5, não há a liberação do *InputStream* em caso de falha. A versão correta da aplicação está implementada no código[10] mostrado no Código 3.6, onde a utilização do recurso foi colocada no bloco *try* e a liberação do recurso foi incluída na cláusula *finally*, assegurando assim a liberação do recurso, ocorrendo uma falha ou não.

[9] <https://github.com/k9mail/k-9/blob/dfa97cd878dceb6821a5e98c722d5bf3c5c5a02d/src/com/fsck/k9/activity/Accounts.java>

[10] <https://github.com/k9mail/k-9/blob/644571cfe512a21e84764a4adcb34cf40da9a335/src/com/fsck/k9/activity/Accounts.java>

Código 3.6 – Exemplo de vazamento em caminhos excepcionais (correto).

```
1   @Override
2   protected Boolean doInBackground(Void... params) {
3       try {
4           InputStream is = mContext.getContentResolver().openInputStream(mUri);
5           try {
6               mImportResults = StorageImporter.importSettings(mContext, is,
7                       mEncryptionKey, mIncludeGlobals, mAccountUuids, mOverwrite);
8           } finally {
9               try {
10                  is.close();
11              } catch (IOException e) { /* Ignore */ }
12          }
13      } catch (StorageImportExportException e) {
14          Log.w(K9.LOG_TAG, "Exception during import", e);
15          return false;
16      } catch (FileNotFoundException e) {
17          Log.w(K9.LOG_TAG, "Couldn't open import file", e);
18          return false;
19      }
20      return true;
21  }
```

Código 3.7 – Exemplo de vazamento em caminhos irregulares (com defeito).

```
1   builder.setNegativeButton(R.string.cancel_action,
2       new DialogInterface.OnClickListener() {
3           @Override
4           public void onClick(DialogInterface dialog, int which) {
5               dialog.dismiss();
6               try {
7                   mInputStream.close();
8               } catch (Exception e) { /* Ignore */ }
9           }
10      });
11  builder.show();
```

Vazamento em caminhos irregulares (*Leak on irregular paths*): Os recursos são liberados em local impróprio, ou seja, a aplicação tem uma operação de liberação executada somente quando ocorre um evento específico, como *onClick*, *onKeyDown* e assim por diante. Se o usuário não acionar nenhum desses eventos, os recursos relacionados não poderão ser liberados. Por exemplo, no Código 3.7 é mostrado um trecho de código da aplicação *k-9 mail*[11], onde se observa que a liberação do recurso de

[11] <https://github.com/k9mail/k-9/blob/dfa97cd878dceb6821a5e98c722d5bf3c5c5a02d/src/com/

um objeto do tipo *InputStream* (linha 7) acontece no evento *onClick* do botão "cancelar" da *dialog*. Caso o usuário saia da aplicação e não clique nesse botão, o recurso não será liberado. O código com a correção do defeito[12], feito por um dos desenvolvedores da aplicação, é mostrado no Código 3.8, onde removeu-se a liberação do recurso do evento *onClick* para um local adequado. Essa situação pode ocorrer com diferentes classes de recursos.

Código 3.8 – Exemplo de vazamento em caminhos irregulares (correto).

```
1  builder.setNegativeButton(R.string.cancel_action,
2          new DialogInterface.OnClickListener() {
3              @Override
4              public void onClick(DialogInterface dialog, int which) {
5                  dialog.dismiss();
6              }
7          });
8  builder.show();
```

fsck/k9/activity/Accounts.java>

12 <https://github.com/k9mail/k-9/blob/644571cfe512a21e84764a4adcb34cf40da9a335/src/com/fsck/k9/activity/Accounts.java>

Código 3.9 – Exemplo de vazamento causado por condição de corrida.

```
1  ...
2  public void stop() {
3      synchronized (createLock) {
4          if (mTask != null) {
5              Log.d(THIS_FILE, "Delete already pushed task in stack");
6              mTask.cancel();
7              sipWakeLock.release(mTask);
8          }
9          if(mTimer != null) {
10             mTimer.purge();
11             mTimer.cancel();
12         }
13         mTimer = null;
14     }
15 }
16 ...
17 protected void onChanged(String type, boolean connected) {
18     boolean fireChanges = false;
19     synchronized (createLock) {
20         ...
21         if (connected) {
22             Log.d(THIS_FILE, "Push a task to connected timer");
23             if (mTask != null) {
24                 Log.d(THIS_FILE, "We already have a current task in stack");
25                 mTask.cancel();
26                 sipWakeLock.release(mTask);
27             }
28             ...
29             sipWakeLock.acquire(mTask);
30         } else {
31             if ((mTask != null) && mTask.mNetworkType.equals(type)) {
32                 mTask.cancel();
33                 sipWakeLock.release(mTask);
34             }
35             fireChanges = true;
36         }
37     }
38     if(fireChanges) {
39         Log.d(THIS_FILE, "Fire changes right now cause it's a deconnect info");
40         dataConnectionChanged(type, false);
41     }
42 }
```

Vazamento causado por condição de corrida (*Leak caused by race condition*): Ocorre quando o gerenciamento de um recurso pode ser realizado (ou seja, adquirido e

liberado) por diferentes *threads* na aplicação. No caso comum, uma *thread* ativa o recurso e, algum tempo depois, outra *thread* desativa o recurso, resultando no comportamento normal da utilização do componente. No entanto, a ordem na qual os encadeamentos tentam acessar o recurso compartilhado pode não ser o esperado devido a diversos fatores. Por exemplo, no trecho de código extraído da aplicação *CSipSimple*[13], mostrado no Código 3.9, é possível observar que a aquisição, utilização e liberação de alguns recursos são realizados por diferentes *threads* (linhas 3 e 19). Para o melhor gerenciamento, foi utilizada a palavra-chave *syncronized*, que bloqueia a utilização de um bloco de código em um dado momento, ou seja, se uma *thread* está executando seu bloco de código restrito, as demais *trheads* não poderão executar seus blocos de código até essa *thread* terminar.

Código 3.10 – Exemplo de vazamento causado por recurso não utilizado (com defeito).

```
1  private void checkOutgoing() throws MessagingException {
2      if (!(account.getRemoteStore() instanceof WebDavStore)) {
3          publishProgress(R.string.account_setup_check_settings_check_outgoing_msg)
               ;
4      }
5      Transport transport = Transport.getInstance(K9.app, account);
6      transport.close();
7      transport.open();
8      transport.close();
9  }
```

Vazamento causado por recurso não utilizado (*Leak caused by unused resource*): O recurso é adquirido, mas não é utilizado. A título de exemplo, tem-se o trecho de código extraído da aplicação *k-9 mail*[14], mostrado no Código 3.10. Nota-se que na linha 6 é liberado o recurso, contudo, o desenvolvedor o adquire novamente na linha 7 e depois o libera novamente na linha 8, sem ter sido utilizado nas últimas duas operações. O código com a correção sugerida pelo autor da tese é mostrado no Código 3.11, onde foram removidas a aquisição e liberação que não eram necessárias.

[13] <https://github.com/r3gis3r/CSipSimple/blob/88d62bc9510809ab6d9750b0a0b761ef08c85948/src/com/csipsimple/service/SipService.java#L646>

[14] <https://github.com/k9mail/k-9/blob/acd18291f2ded8f176beb074415f8c42ca829966/k9mail/src/main/java/com/fsck/k9/activity/setup/AccountSetupCheckSettings.java>

Código 3.11 – Exemplo de vazamento causado por recurso não utilizado (correto).

```
1  private void checkOutgoing() throws MessagingException {
2      if (!(account.getRemoteStore() instanceof WebDavStore)) {
3          publishProgress(R.string.account_setup_check_settings_check_outgoing_msg)
               ;
4      }
5      Transport transport = Transport.getInstance(K9.app, account);
6      transport.close();
7  }
```

Código 3.12 – Exemplo de vazamento causado pela complexidade do ciclo de vida da aplicação (com defeito).

```
1  @Override
2  protected void onPause() {
3      super.onPause();
4      if (mChatController != null) {
5          mChatController.onPause();
6      }
7
8      stopWatchingExternalStorage();
9
10 }
```

Vazamento causado pela complexidade do ciclo de vida da aplicação (*Leak caused by complex application lifecycle*): Má administração de recursos durante o ciclo de vida de um componente, seja por descuido ou desconhecimento de como funciona o ciclo de vida do componente Android. Um exemplo desse problema pode ser visto no trecho de código da aplicação *Surespot*[15] mostrado no Código 3.12, no qual não foi implementada a instrução para pausar a captação do áudio no método *onPause* do ciclo de vida. A correção[16] desse defeito feita por um dos autores da aplicação é mostrada no Código 3.13. Na linha 8 há uma instrução que chama o método que pausa a captação do áudio. Uma forma de reproduzir o vazamento é alternar a exibição dessa aplicação e de outra, enquanto estiver em diferentes estados do componente (e.g. navegue até a tela inicial e retorne à sua aplicação).

15 <https://f-droid.org/forums/topic/surespot-complete-whatsapp-alternative-application/>
16 <https://f-droid.org/forums/topic/surespot-complete-whatsapp-alternative-application/>

Código 3.13 – Exemplo de vazamento causado pela complexidade do ciclo de vida da aplicação (correto).

```
1   @Override
2   protected void onPause() {
3       super.onPause();
4       if (mChatController != null) {
5           mChatController.onPause();
6       }
7
8       VoiceController.pause();
9
10      stopWatchingExternalStorage();
11
12  }
```

Código 3.14 – Exemplo de vazamento causado por alocação e desalocação imprópria (com defeito).

```
1   private void stopMovement() {
2       listView.onTouchEvent(MotionEvent.obtain(SystemClock.uptimeMillis(),
3               SystemClock.uptimeMillis(), MotionEvent.ACTION_CANCEL, 0, 0, 0));
4   }
```

Vazamento causado por alocação e desalocação imprópria (*Leak caused by improper allocation and deallocation*): A alocação e desalocação inadequadas de espaço na memória nativa, o mau uso de *API* (*Application Programming Interface*), falta ou perda de referências a objetos de recursos podem causar o aumento inesperado da memória usada. Por exemplo, no método *stopMovement*[17] (Código 3.14) da aplicação *Xabber* é alocado e utilizado o recurso *MotionEvent*, mas este não é liberado, o que faz com que a *View* fique ativa e não seja liberada. O código com a correção[18] implementada por um dos desenvolvedores da aplicação é mostrado no Código 3.15, em que a alocação do recurso é atribuída a uma variável (linhas 2 e 3), depois o mesmo é utilizado (linha 4) e então na linha 5 é chamado o método *recycle* (permitindo que o objeto seja destruído).

[17] <https://github.com/redsolution/xabber-android/blob/926f4ca93ecaebf2c614dd65a8aab39de8757218/src/com/xabber/android/ui/ContactListFragment.java>

[18] <https://github.com/redsolution/xabber-android/blob/4cf5f2db9a67189b8990899651f1618bd1b0fd1e/src/com/xabber/android/ui/ContactListFragment.java>

Código 3.15 – Exemplo de vazamento causado por alocação e desalocação imprópria (correto).

```
1    private void stopMovement() {
2        MotionEvent event = MotionEvent.obtain(SystemClock.uptimeMillis(),
3                SystemClock.uptimeMillis(), MotionEvent.ACTION_CANCEL, 0, 0, 0);
4        listView.onTouchEvent(event);
5        event.recycle();
6    }
```

Código 3.16 – Exemplo de vazamento causado por liberação inadequada de recursos em loop (com defeito).

```
1  private boolean resumeDownload() {
2      BufferedInputStream in = null;
3      FileOutputStream fos = null;
4      BufferedOutputStream bout = null;
5
6      try {
7          for (; downloadIndex < fileNames.length; downloadIndex++) {
8              ...
9              in = new BufferedInputStream(connection.getInputStream());
10             fos = (downloaded == 0) ? new FileOutputStream(file
11                     .getAbsolutePath()) : new FileOutputStream(file
12                     .getAbsolutePath(), true);
13
14             bout = new BufferedOutputStream(fos, DOWNLOAD_BUFFER_SIZE);
15             byte[] data = new byte[DOWNLOAD_BUFFER_SIZE];
16             int x = 0;
17
18             while (isRunning && (x = in.read(data, 0, DOWNLOAD_BUFFER_SIZE)) >=
                   0) {
19                 bout.write(data, 0, x);
20                 downloaded += x;
21                 double percent = 100.0 * ((1.0 * downloaded) / (1.0 * total));
22                 updateProgress((int) percent, fileNames.length, downloadIndex);
23             }
24             ...
25         }
26     } catch (FileNotFoundException e) {
27         Log.e("quran_srv", "File not found: IO Exception", e);
28     } catch (IOException e) {
29         Log.e("quran_srv", "Download paused: IO Exception", e);
30         return false;
31     } catch (Exception e) {
32         Log.e("quran_srv", "Download paused: Exception", e);
33         return false;
34     }
35
36     return true;
37 }
```

Vazamento causado por liberação inadequada de recursos em loop (*Leak caused by improper resource liberation in loop*): Um recurso é adquirido repetidamente dentro de um loop, mas não é liberado por uma quantidade suficiente de vezes antes de sair da

aplicação. Por exemplo, o método *resumeDownload*[19] (Código 3.16) da aplicação *Quran for Android* possui a alocação de três recursos (*BufferedInputStream*, *FileOutputStream* e *BufferedOutputStream*) dentro de um laço de repetição e esses recursos não são liberados dentro do laço. A correção[20] feita por um dos desenvolvedores é mostrada no Código 3.17, onde nas linhas 24 a 26 são realizadas as operações de liberação dos recursos.

[19] <https://github.com/quran/quran_android/blob/121cd5803b610eeb79bd42c78ece53df7043cc16/src/com/quran/labs/androidquran/service/QuranDataService.java>

[20] <https://github.com/quran/quran_android/blob/4e4a60249a27cbf834d59db4611eb9f1843240c0/src/com/quran/labs/androidquran/service/QuranDataService.java>

Código 3.17 – Exemplo de vazamento causado por liberação inadequada de recursos em loop (correto).

```
1   private boolean resumeDownload() {
2       BufferedInputStream in = null;
3       FileOutputStream fos = null;
4       BufferedOutputStream bout = null;
5
6       try {
7           for (; downloadIndex < fileNames.length; downloadIndex++) {
8               ...
9               in = new BufferedInputStream(connection.getInputStream(),
                    DOWNLOAD_BUFFER_SIZE);
10              fos = (downloaded == 0) ? new FileOutputStream(file
11                      .getAbsolutePath()) : new FileOutputStream(file
12                      .getAbsolutePath(), true);
13
14              bout = new BufferedOutputStream(fos, DOWNLOAD_BUFFER_SIZE);
15              byte[] data = new byte[DOWNLOAD_BUFFER_SIZE];
16              int x = 0;
17
18              while (isRunning && (x = in.read(data, 0, DOWNLOAD_BUFFER_SIZE)) >=
                    0) {
19                  bout.write(data, 0, x);
20                  downloaded += x;
21                  double percent = 100.0 * ((1.0 * downloaded) / (1.0 * total));
22                  updateProgress((int) percent, fileNames.length, downloadIndex);
23              }
24              bout.flush();
25              bout.close();
26              fos.close();
27              ...
28          }
29      } catch (FileNotFoundException e) {
30          Log.e("quran_srv", "File not found: IO Exception", e);
31      } catch (IOException e) {
32          Log.e("quran_srv", "Download paused: IO Exception", e);
33          return false;
34      } catch (Exception e) {
35          Log.e("quran_srv", "Download paused: Exception", e);
36          return false;
37      }
38
39      return true;
40  }
```

Código 3.18 – Exemplo de vazamento causado pelo Android SDK (com defeito).

```
1  ...
2  public class MainActivity extends Activity {
3    private TextView immView;
4    @Override protected void onCreate(Bundle savedInstanceState) {
5      super.onCreate(savedInstanceState);
6      ...
7      Button button = new Button(this);
8      button.setText("Remove EditText");
9      button.setOnClickListener(new View.OnClickListener() {
10       @Override public void onClick(View v) {
11         layout.removeView(leaking);
12         layout.removeView(v);
13       }
14     });
15     ...
16     updateImmView();
17   }
18   private void updateImmView() {
19     logServedView();
20     immView.postDelayed(new Runnable() {
21       @Override public void run() {
22         updateImmView();
23       }
24     }, 100);
25   }
26   private void logServedView() {
27     try {
28       Field sInstanceField = InputMethodManager.class.getDeclaredField("sInstance
             ");
29       sInstanceField.setAccessible(true);
30       ...
31       View servedView = (View) mServedViewField.get(imm);immView.setText("
             InputMethodManager.mServedView: "
32           + servedView.getClass().getName()
33           + "\nAttached:"
34           + servedView.isAttachedToWindow());
35     } catch (NoSuchFieldException e) {
36       throw new RuntimeException(e);
37     } catch (IllegalAccessException e) {
38       throw new RuntimeException(e);
39     }
40   }
41 }
```

Vazamento causado pelo Android SDK (*Leak caused by Android SDK*): Os vazamentos podem ser causados pelo Android SDK e o desenvolvedor dispõe de poucos meios para corrigi-los. Alguns padrões de vazamentos conhecidos estão descritos no *AndroidExcludedRefs.java*[21] fornecido por *LeakCanary*[22]. Um exemplo é o vazamento da última *View* focada que acontece na *API* 22 do Android. Os passos para reproduzir essa falha são: (1) configurar uma hierarquia de *View* com apenas uma focalizável; (2) solicitar foco na *View* focalizável; (3) remover essa *View* (ou um de seus ancestrais) da hierarquia de *View*. Um possível código[23] para reproduzir essa falha é mostrado no Código 3.18, no qual o *TextView* (linhas 19 a 24) exibe a referência em *InputMethodManager.mServedView*, atualizada a cada 100 milissegundos. Ao clicar no botão declarado nas linhas 7 a 14 o *EditText* é removido. Após isso, nota-se que ele ainda é referenciado por *InputMethodManager.mServedView*, embora esteja desanexado. Um desenvolvedor descobriu um meio de evitar esse vazamento, o qual foi implementado na forma de um método chamado *fixInputMethod* (mostrado no Código 3.19) que deve ser chamado no método *onDestroy* do ciclo de vida.

[21] <https://github.com/hehonghui/leakcanary-for-eclipse/blob/master/Leakcanary-lib/src/com/squareup/leakcanary/AndroidExcludedRefs.java>
[22] <https://square.github.io/leakcanary/>
[23] <https://issuetracker.google.com/issues/37043700#comment1>

Código 3.19 – Exemplo de vazamento causado pelo Android SDK (correto).

```
1  public static void fixInputMethod(Context context) {
2      if (context == null) return;
3      InputMethodManager inputMethodManager = null;
4      try {
5          inputMethodManager = (InputMethodManager) context.getApplicationContext()
6              .getSystemService(Context.INPUT_METHOD_SERVICE);
7      } catch (Throwable th) {
8          th.printStackTrace();
9      }
10     if (inputMethodManager == null) return;
11     String[] strArr = new String[]{"mCurRootView", "mServedView", "
           mNextServedView"};
12     for (int i = 0; i < 3; i++) {
13         try {
14             Field declaredField = inputMethodManager.getClass().getDeclaredField(
                   strArr[i]);
15             if (declaredField == null) continue;
16             if (!declaredField.isAccessible()) {
17                 declaredField.setAccessible(true);
18             }
19             Object obj = declaredField.get(inputMethodManager);
20             if (obj == null || !(obj instanceof View)) continue;
21             View view = (View) obj;
22             if (view.getContext() == context) {
23                 declaredField.set(inputMethodManager, null);
24             } else {
25                 return;
26             }
27         } catch (Throwable th) {
28             th.printStackTrace();
29         }
30     }
31 }
```

3.6 Conclusões do Mapeamento Sistemático

Esta seção apresentou o estudo do mapeamento sistemático de publicações científicas sobre identificação de vazamentos de recursos em aplicações móveis. No estudo, foram incluídos um total de 57 artigos publicados entre 2012 e 2024. Os resultados mostraram 14 classes de recursos nos quais os trabalhos atuaram, sendo que as classes de recursos

Energia (50,9%) e *WakeLock* (36,8%) foram os dois mais explorados nos artigos.

A partir da análise dos artigos foi possível identificar 10 categorias de causas de vazamento, as quais podem auxiliar os desenvolvedores a evitar esses problemas em suas aplicações. Além disso, apresenta-se uma análise bibliométrica da área para obter uma compreensão da tendência de publicações por ano, citações, pesquisadores ativos e veículos de publicações da área. Na próxima seção será apresentado um estudo que irá analisar as categorias de causas de vazamentos identificados no mapeamento sistemático.

3.7 Trabalhos Relacionados

Nesta seção serão apresentados os trabalhos relacionados encontrados durante a revisão da literatura científica.

3.7.1 Detecção de Vazamentos de Recursos

Zhang et al. (2012) desenvolveram o *ADEL* (Detector Automático de Vazamentos de Energia) que consiste em aprimoramentos de rastreamento de contaminação (em inglês, *taint-tracking*) para a plataforma Android. Ele detecta e isola vazamentos de energia resultantes de comunicação de rede desnecessária, rastreando o uso direto e indireto dos dados recebidos para determinar se eles afetam o usuário.

Guo et al. (2013) propuseram um método automático para detectar vazamentos de recursos com base em um gráfico de chamada de componente modificado. Esse método lida com os atributos da programação móvel orientada a eventos, analisando os retornos de chamada definidos na estrutura do Android. Por fim, usaram a pesquisa em profundidade no gráfico de chamadas de função e analisaram a hierarquia de classes para identificar recursos não liberados.

Qian e Zhou (2016) desenvolveram uma nova abordagem para priorizar os casos de teste de acordo com sua probabilidade de causar vazamentos de memória em um determinado conjunto de testes. Em primeiro lugar, é construído um modelo de previsão

para determinar se cada teste pode potencialmente levar a vazamentos de memória com base no aprendizado de máquina em atributos de código selecionados. Em seguida, cada caso de teste de entrada é executado parcialmente para obter seus atributos de código e prever sua probabilidade de causar vazamentos. A abordagem sugere que os casos de teste com maior probabilidade sejam executados primeiro (de forma completa), a fim de revelar falhas de vazamento de memória o mais rápido possível.

Zhang, Wu e Rountev (2016) apresentam uma abordagem para geração sistemática de teste automatizado para expor defeitos de vazamento de recursos em aplicações Android. Primeiro definiram a noção de uma sequência neutra de eventos da UI. Intuitivamente, pode-se esperar que tal sequência tenha efeitos "neutros" no consumo de recursos, de modo que execuções repetidas dessa sequência não devem exibir crescimento de recursos. Usando um modelo de fluxo de controle, de uma aplicação Android, demonstraram como obter a geração automatizada de tais sequências. Em seguida, definiram algoritmos de geração de teste para duas categorias importantes de sequências neutras (que não devem aumentar o consumo de recursos, se aumentar é porque tem vazamento de recursos), com base em padrões de vazamento comuns específicos para Android.

Jiang et al. (2017) desenvolveram uma técnica de análise estática chamada *SAAD*, que detecta vazamento de recursos em aplicações Android por análise sensível ao contexto. Essa técnica combina análise de chamada de componente, análise interprocedimento e análise intraprocedimento, sendo considerado o contexto de chamada ao analisar o destino de uma chamada de componente. Para melhorar a eficiência, os autores se concentraram na análise de caminhos eficazes que estão envolvidos nas operações de utilização/liberação de recursos.

Ma et al. (2018) mostraram o *LESDroid* para detecção de vazamentos de serviços exportados, pois os serviços podem vazar de modo que não sejam mais usados, mas não possam ser reciclados pelo Coletor de Lixo. O *LESDroid* gera automaticamente instâncias de serviço e cargas de trabalho (iniciar/parar ou vincular/desvincular de serviços exportados) da aplicação em teste e aplica um oráculo designado ao instantâneo de *heap*[24] para detecção de vazamento de serviço.

[24] Um *heap* é uma estrutura de dados especializada, baseada em árvore

Xu, Wen e Qin (2018) propuseram uma análise estática chamada análise de estado-marcação (em inglês, *state-taint*) para detectar vazamentos de recursos, especificando o uso apropriado de recursos em termos de protocolos de recursos. Em seguida, propuseram uma análise semelhante a marcação (em inglês, *taint-like*) que emprega protocolos de recursos para orientar a detecção de vazamentos de recursos. Como uma extensão e uma aplicação, enriqueceram os protocolos com os comportamentos inadequados que podem causar vazamentos de energia e usaram os protocolos refinados para orientar a análise para detecção de vazamento de energia. Implementaram a análise como uma ferramenta de protótipo chamada *Statedroid*.

Hoshieah et al. (2019) desenvolveram uma ferramenta chamada *SAALC* que é capaz de analisar as atividades (em inglês, *activities*) do Android e extrair informações valiosas sobre o uso de componentes de retorno de chamada do ciclo de vida. Os resultados mostraram quais componentes de retorno de chamada são implementados e a natureza do código que eles contêm. De igual modo, mostraram a implementação incorreta dos componentes de retorno de chamada e aquisição e liberação incorreta de recursos do sistema em muitas aplicações Android.

Toffalini, Sun e Ochoa (2019) apresentaram um novo método de análise estática que encontra vazamentos de contexto em aplicações nativas Android escritas em Java. Primeiro, os APK são convertidos em *bytecode* Java. Segundo, o *bytecode* Java é analisado com a ferramenta *Julia* para identificar locais onde contextos podem se tornar acessíveis a partir de campos estáticos ou *threads*. Por último, os potenciais vazamentos são analisados sistematicamente para definir sua gravidade. Por exemplo, se a execução de um componente vazar um contexto, o problema será mais perigoso se o componente contiver *loops* ou executar bloqueio de E/S, estendendo assim a duração do vazamento.

Bhatt e Furia (2020) propuseram *PlumbDroid*, que é uma técnica para detectar e corrigir automaticamente vazamentos de recursos em aplicações Android. O *PlumbDroid* usa análise estática para encontrar rastros de execução que podem vazar um recurso. As informações criadas para detecção também sustentam a construção automática de uma correção (consistindo em operações de liberação realizadas em locais apropriados) que remove o vazamento e não afeta o uso do recurso pela aplicação. Kellogg et al. (2021)

estenderam o *Checker Framework* com o Verificador de Vazamento de Recursos (RLC) e realizaram um estudo comparando com duas ferramentas (Compilador Eclipse para Java (ECJ) e *Grapple*). O ECJ é muito rápido (quase instantâneo), mas tem baixa precisão (25% para avisos de alta confiança; muito menor se todos os avisos forem incluídos). O *Grapple* é mais preciso (50% de precisão), mas uma ordem de grandeza mais lento que o RLC. O RLC teve 100% de revocação e 26% de precisão. A quantidade de falsos positivos do RLC é próxima à das ferramentas e foi possível descobrir 49 vazamentos de recursos em código Java muito usado e muito testado.

Pereira et al. (2022) fizeram uma extensão do *EcoAndroid* (um plugin para a IDE Android Studio), na qual adicionaram o suporte a detecção de vazamentos relacionados a quatro recursos do Android: *Cursor*, *SQLiteDatabase*, *Wakelock* e *Camera*. Foi realizado um estudo comparando com oito ferramentas, no qual a ferramenta teve a pior taxa de detecção de vazamentos, mas a segunda melhor taxa de falsos positivos. Também foi realizado um segundo estudo, somente com o *EcoAndroid*, para analisar 107 aplicações do *DroidLeaks* onde foram descobertos 191 vazamentos de recursos não identificados anteriormente (74 ao considerar vazamentos exclusivos) e obteve uma precisão de 72,5%.

Facebook (2024) mantém a ferramenta *Infer*, a qual usa análise estática. Ela pode analisar código nas linguagens Java, C, C++ e Objective-C. Após a análise ela irá produzir uma lista de possíveis defeitos (inclusive vazamentos de recursos). Algumas das empresas que usam essa ferramenta são: *Amazon Web Services*, *Spotify*, *Uber*, *WhatsApp*, *Microsoft*, *Mozilla* e *Instagram* (FACEBOOK, 2024).

Pugh, Loskutov e Lea (2024) apresentam a ferramenta de análise estática *FindBugs*. Ela consegue identificar mais de 200 padrões de erros como vazamentos de recursos, mau uso das bibliotecas Java e *deadlocks*. Uma diferença entre essa ferramenta e a abordagem proposta, é que a abordagem pode identificar uma maior quantidade de classes de vazamento que a ferramenta. O projeto é de código aberto e foi baixado mais de 230.000 vezes e é usado por muitas grandes empresas e instituições financeiras.

Android (2024) disponibiliza a ferramenta *Android Lint*, essa ferramenta ajuda a encontrar códigos com estrutura ineficiente que podem afetar a confiabilidade e

eficiência das aplicações Android e dificultar a manutenção do código. Os possíveis erros e melhorias são agrupados conforme os seguintes critérios: precisão, segurança, desempenho, usabilidade, acessibilidade e internacionalização.

Tabela 3.6 – Comparação com os trabalhos relacionados sobre vazamentos de recursos.

Artigo	Objetivo	Análise	Detalhes
(ZHANG et al., 2012)	Detectar vazamentos de recursos	Dinâmica	Análise dinâmica de rastreamento de contaminação para vazamentos de energia
(GUO et al., 2013)	Detectar vazamentos de recursos	Estática	Programação móvel orientada a eventos e chamada de componente modificado
(QIAN; ZHOU, 2016)	Priorizar casos de teste para a revelação de vazamentos de recursos	Estática e Dinâmica	Aprendizado de máquina
(ZHANG; WU; ROUNTEV, 2016)	Gerar testes de UI para detectar vazamentos de recursos	Estática e Dinâmica	Fluxo de controle, ciclo neutro do *Window Transition Graph* (WTG) e teste de software
(JIANG et al., 2017)	Detectar vazamentos de recursos	Estática	Análise sensível ao contexto (análise de chamada de componente, análise interprocedimento e análise intraprocedimento)
(MA et al., 2018)	Detectar vazamentos de recursos	Estática	Aplicou-se uma análise estática conservadora insensível ao fluxo para vazamento de serviços
(XU; WEN; QIN, 2018)	Detectar vazamentos de recursos	Estática	Análise de estado-contaminação, protocolos de recursos e gráfico de fluxo de controle (em inglês, CFG)
(HOSHIEAH et al., 2019)	Detectar vazamentos de recursos	Estática	Componentes de retorno de chamada do ciclo de vida
(TOFFALINI; SUN; OCHOA, 2019)	Detectar vazamentos de recursos	Estática	Identificar locais onde contextos podem se tornar acessíveis a partir de campos estáticos ou *threads*
(BHATT; FURIA, 2020)	Detectar e corrigir automaticamente vazamentos de recursos	Estática	Gráfico de fluxo de recursos
(KELLOGG et al., 2021)	Detectar vazamentos de recursos	Estática	Análise intraprocedural de fluxo de dados
(PEREIRA et al., 2022)	Detectar vazamentos de recursos	Estática	Análise interprocedural sensível ao contexto e ao fluxo
(FACEBOOK, 2024)	Detectar vazamentos de recursos	Estática	Análise intraprocedural e caminhos de fluxo de controle excepcionais
(PUGH; LOSKUTOV; LEA, 2024)	Detectar vazamentos de recursos	Estática	Análise intraprocedural de fluxo de dados
(ANDROID, 2024)	Detectar vazamentos de recursos	Estática	Utiliza uma das tarefas Gradle (lint) que visita cada parte do código e a examina com base em um conjunto definido de regras
Este trabalho	Detectar vazamentos de recursos	Estática	Aprendizado de máquina e métricas de software

Na Tabela 3.6 é mostrado um resumo dos trabalhos relacionados sobre vazamentos de recursos. Um dos trabalhos relacionados utiliza análise estática e dinâmica e aprendizado de máquina para priorizar casos de teste com maior probabilidade de revelar vazamentos de recursos. Neste trabalho, pretende-se utilizar métricas extraídas utilizando análise estática e aprendizado de máquina com o objetivo de detectar vazamentos de recursos, considerando uma maior variedade de classes de recursos do que os trabalhos relacionados.

Assim sendo, podemos destacar as cinco ferramentas do estado da arte que serão usadas no estudo de viabilidade da proposta: *Android Lint* (ANDROID, 2024), *FindBugs* (PUGH; LOSKUTOV; LEA, 2024), *Infer* (FACEBOOK, 2024), *Checker Framework*

(KELLOGG et al., 2021), e *EcoAndroid* (PEREIRA et al., 2022).

3.8 Considerações Finais

Este capítulo apresentou o estudo do mapeamento sistemático de publicações científicas sobre identificação de vazamentos de recursos em aplicações móveis. No estudo, foram incluídos um total de 57 artigos publicados entre 2012 e 2024. A partir da análise dos artigos foi possível identificar 10 categorias de causas de vazamentos, que podem auxiliar os desenvolvedores a evitar esses problemas em suas aplicações.

Os resultados também mostraram 14 tipos de recursos em que os artigos atuaram, sendo que os tipos de recursos Energy (50,9%) e WakeLock (36,8%) foram os dois mais explorados nos artigos. Também foram apresentados os trabalhos relacionados identificados na literatura científica e uma análise comparativa entre eles. No próximo capítulo, será apresentado um estudo que analisará as categorias de causas de vazamentos identificadas no mapeamento sistemático.

4

PESQUISA DE OPINIÃO

No mapeamento sistemático anteriormente apresentado foram extraídas 10 categorias de causas de vazamentos em aplicações móveis. Com o intuito de validar essas categorias, foi planejada uma pesquisa de opinião cujos detalhes são apresentados nesta seção.

4.1 Objetivo

O objetivo deste estudo está esquematizado a partir do paradigma GQM (em inglês, *Goal, Question, and Metric*) (BASILI; CALDIERA; ROMBACH, 1994), o qual é apresentado na Tabela 4.1. A execução desta pesquisa de opinião teve como principal objetivo caracterizar a relevância das categorias de causas de vazamentos identificados no mapeamento sistemático.

Tabela 4.1 – Objetivo da pesquisa de opinião segundo o paradigma GQM.

Analisar	Categorias de causas de vazamentos
Com o propósito de	Caracterizar
Com relação	À relevância
Do ponto de vista de	Profissionais de desenvolvimento
No contexto de	Aplicações móveis

4.2 Questão de Pesquisa

Com base no objetivo foi formulada a questão de pesquisa abaixo:

Qual a relevância de cada uma das categorias para representar causas de vazamento de recursos para aplicações móveis?

Métricas: Valor da escala de *Likert* atribuído pelo participante e o peso do participante.

4.3 Instrumentação

Foram propostas questões aos participantes com o objetivo de identificar o seu perfil no que se refere à sua experiência com vazamentos de recursos em aplicações móveis, além do Termo de Consentimento Livre e Esclarecido - TCLE. Além disso, foi preparado um questionário para avaliar a relevância das categorias de causas de vazamentos em aplicações móveis conforme a questão de pesquisa citada na subseção anterior.

Será utilizada a escala ordinal de *Likert*, oferecendo as opções: (0) Nenhuma relevância, (1) Muito baixa relevância, (2) Baixa relevância, (3) Média relevância, (4) Alta relevância e (5) Muito alta relevância. Além disso, questiona-se por intermédio de uma pergunta aberta: "Qual a sua opinião sobre as categorias apresentadas? Há alguma outra categoria de causas de vazamentos?". O questionário referente à avaliação está disponível em: <https://drive.google.com/drive/folders/12zC6TUQhtm-KU1DGsnfuYvLyzAETfrln>.

4.4 Procedimento para Análise da Relevância

Para definir o nível de relevância de cada categoria utilizou-se como base os anos de experiência do participante e as plataformas móveis em que ele trabalha ou trabalhou. De acordo com (DIAS-NETO; TRAVASSOS, 2008) é necessário diferenciar as respostas dos participantes associando um peso a cada um deles, considerando por exemplo, anos de experiência. O peso de cada participante para este estudo é calculado conforme a Equação 4.1.

$$Peso(i) = QPT(i) + \frac{AnosDeExperiencia(i)}{Mediana(AnosDeExperienciaDeTodosParticipantes)} \quad (4.1)$$

Na qual *Peso(i)* é o peso atribuído ao participante *i*; e *QPT(i)* é quantidade de plataformas trabalhadas do participante *i*.

Por fim, o nível de relevância está sendo calculado como um valor entre 0% e 100% por meio da normalização do valor obtido por cada categoria, conforme mostrado na Equação 4.2.

$$NivelRelevancia(c) = \frac{\sum_{i=1}^{N}(EscalaLikert(i,c) \times Peso(i))}{\sum_{i=1}^{N}(Peso(i) \times 5)} \quad (4.2)$$

Na qual *NivelRelevancia(c)* é o nível de relevância para a categoria *c*; *EscalaLikert(i, c)* é o valor de relevância, de acordo com a escala de *Likert* (entre 0 e 5), definida pelo participante *i* para a categoria *c*; *Peso(i)* é o peso atribuído para o participante *i*; *N* é o total de participantes que responderam a pesquisa; e a constante *5* é o valor máximo para a escala do nível de relevância.

Após esta etapa, os valores serão ordenados na forma decrescente. As categorias mais relevantes serão aquelas cuja variável *NivelRelevancia(c)* têm os maiores valores.

4.5 Execução do Estudo

Para participar desse estudo, foram convidados desenvolvedores com experiência em vazamentos de recursos em aplicações móveis. A base de desenvolvedores foi obtida a partir de uma busca na rede social *LinkedIn*[1], como também em comunidades de desenvolvimento no *Facebook*[2]. Para participar do estudo, os profissionais tiveram que manifestar interesse, concordando com o Termo de Consentimento Livre e Esclarecido (TCLE) e, após isso, responderam o questionário proposto.

O questionário ficou ativo por um período de 72 dias. O formulário, enviado para 147 desenvolvedores, obteve respostas de 22 deles, alcançando 81% de nível de

[1] http://linkedin.com
[2] https://www.facebook.com

confiança sobre a quantidade de participantes, de acordo com a fórmula de (HAMBURG, 1980) e uma taxa de resposta de 15%.

Tabela 4.2 – Participantes da pesquisa de opinião.

ID	Plataformas	Anos de Experiência	País	Peso
P01	Android	4	Paquistão	2
P02	Android, iOS	5	Índia	3,25
P03	Android	6	Índia	2,5
P04	Android, iOS	5	Brasil	3,25
P05	Android, iOS	2	Brasil	2,5
P06	Windows	7	Brasil	2,75
P07	Android, iOS	6	Brasil	3,5
P08	Android	5	Paquistão	2,25
P09	Android	2	Brasil	1,5
P10	Android, iOS	2	Brasil	2,5
P11	Android	3	Brasil	1,75
P12	Android	3	Brasil	1,75
P13	Android	3	Brasil	1,75
P14	Android, iOS	8	Brasil	4
P15	Android	4	Brasil	2
P16	Android, Windows	10	Brasil	4,5
P17	Android, iOS	5	Brasil	3,25
P18	Android, iOS	1	Brasil	2,25
P19	Android, iOS	4	Brasil	3
P20	Android	1	Brasil	1,25
P21	Android	5	Brasil	2,25
P22	Android, iOS	2	Brasil	2,5

Em respeito à privacidade, os dados pessoais dos participantes não serão apresentados. A Tabela 4.2 apresenta as informações de caracterização dos desenvolvedores, incluindo o peso já calculado para cada um. Pode-se observar que eles trabalham ou trabalharam em pelo menos uma das seguintes plataformas: *Android*, *iOS* ou *Windows*. Os participantes também moram em diferentes países como *Brasil*, *Índia* ou *Paquistão*. A mediana do tempo de experiência com vazamentos de recursos em aplicações móveis dos desenvolvedores neste estudo é de 4 anos.

Tabela 4.3 – Relevância das categorias de causas de vazamento.

ID	Categoria	Relevância
C01	Vazamento em caminhos excepcionais	76%
C02	Vazamento causado pela complexidade do ciclo de vida da aplicação	74%
C03	Vazamento causado por liberação inadequada de recursos em loop	71%
C04	Vazamento em caminhos irregulares	70%
C05	Vazamento causado por alocação e desalocação imprópria	70%
C06	Vazamento causado por condição de corrida	70%
C07	Vazamento causado pelo Android SDK	69%
C08	Vazamento completo	68%
C09	Vazamento em caminhos normais	63%
C10	Vazamento causado por recurso não utilizado	55%

4.6 Análise dos Resultados

Os dados obtidos, de acordo com a opinião dos participantes do estudo, indicam 6 categorias (C01, C02, C03, C04, C05 e C06) como as mais relevantes (com valor igual ou maior à mediana calculada de 70%) para representar causas de vazamento em aplicações móveis, conforme mostrado na Tabela 4.3. As duas categorias mais relevantes foram C01 e C02, o que indica que os desenvolvedores devem se preocupar com a liberação de recursos tanto nas possibilidades de exceções da aplicação, quanto dentro dos métodos de *callback* (por exemplo, *onStop()* ou *viewWillDisappear()*) do ciclo de vida da aplicação.

Da análise feita a respeito de quais categorias foram marcadas com o nível de nenhuma relevância (Figura 4.1), pode-se observar que nenhum dos participantes marcou as categorias C01, C05, C06 e C07. Entretanto, foram marcadas as categorias C02, C03 e C04 por um participante (4,54%), a C09 por dois (9,09%) e as C08 e C10 por três (13,63%).

As categorias C07, C08, C09 e C10 não ficaram entre as mais relevantes. No entanto, elas são relevantes para representar causas de vazamentos em aplicações móveis, pois poucos participantes as marcaram como categorias com nenhuma relevância.

Em relação as respostas à pergunta aberta "Qual a sua opinião sobre as categorias apresentadas? Há alguma outra categoria de causas de vazamentos?", podem-se destacar algumas observações, por exemplo, o participante P05 confirmou que os de-

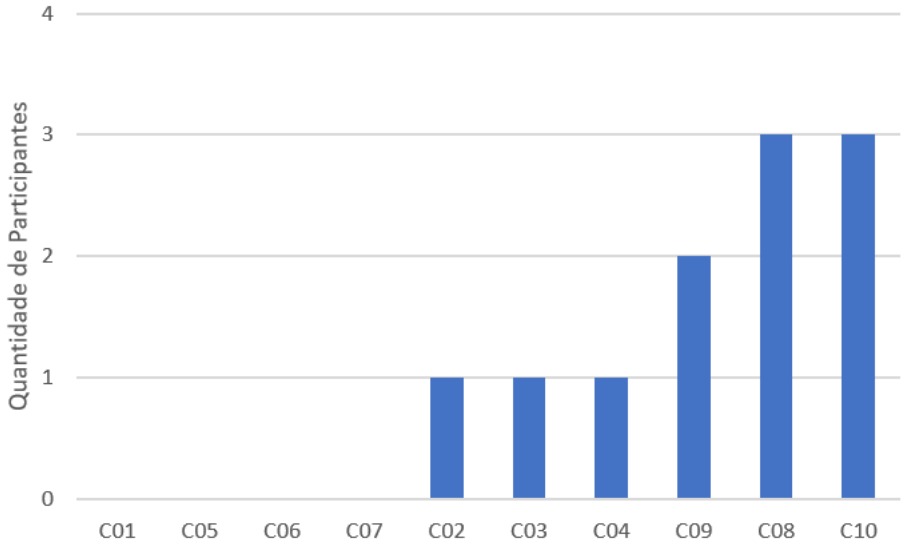


Figura 4.1 – Quantidade de marcações como sendo de nenhuma relevância pelos participantes.

senvolvedores podem falhar completamente na liberação dos recursos da aplicação.

*"Bem, eu concordo com o C08 (os desenvolvedores falham completamente em liberar os recursos adquiridos após o uso), isso é verdade, não costumamos olhar para essas coisas."***(P05)**

Olhando a resposta do participante P07, pode-se inferir que a documentação do Android precisa destacar melhor a necessidade de os desenvolvedores liberarem os recursos de forma explícita para não ocorrer o vazamento de recursos.

"Acho que a documentação da plataforma Android pressupõe aos desenvolvedores que o sistema operacional cuidará do vazamento, e isso pode levar o desenvolvedor a pensar que o vazamento de recurso não será um problema para suas aplicações." **P07**

Os participantes P02, P10 e P19 consideram satisfatórias as categorias apresentadas nesta pesquisa.

"Boas." **P02**

"Não, tudo isso está representado na pesquisa." **P10**

"Eu acho que as categorias apresentadas resumem muito bem as mais importantes." **P19**

O participante P12 considera que algumas dessas categorias não são comuns de ocorrer, pois ele acredita que algumas são fáceis de serem detectadas. No entanto, baseado na resposta dos partipantes P05 e P07, é possível inferir que alguns desenvolvedores ainda desconhecem a importância da correta liberação dos recursos das

aplicações e, consequentemente, deixam de fazê-la.

"Algumas delas não são tão comuns, por exemplo, é fácil capturar quando o recurso não está "fechado", até o analista de código pode verificar e sugerir. A condição de corrida é realmente comum e complicada, leva tempo para identificar e não é tão fácil de depurar. Eu acho que isso é o mais comum nos projetos em que trabalhei." **P12**

Além das categorias presentes no conjunto inicial, foi adicionada uma nova categoria proposta pelo participante P06 com o nome *"Vazamento de memória devido a APIs de terceiros mal construídas"*. O mesmo argumentou: *"E algo que é muito comum é a escolha de APIs de terceiros (Android-Arsenal tem milhares) sem muitos critérios ... e essas terem uma quantidade considerável de problemas do tipo Memory leak. Alias, me atrevo a dizer que essa é a principal origem de vazamentos de memória."* Na Tabela 4.4 é mostrada a lista final de categorias, incluída a categoria adicionada.

Tabela 4.4 – Lista final das categorias de causas de vazamento.

ID	Categoria
C01	Vazamento em caminhos excepcionais
C02	Vazamento causado pela complexidade do ciclo de vida da aplicação
C03	Vazamento causado por liberação inadequada de recursos em loop
C04	Vazamento em caminhos irregulares
C05	Vazamento causado por alocação e desalocação imprópria
C06	Vazamento causado por condição de corrida
C07	Vazamento causado pelo Android SDK
C08	Vazamento completo
C09	Vazamento em caminhos normais
C10	Vazamento causado por recurso não utilizado
C11	Vazamento de memória devido a APIs de terceiros mal construídas**

Legenda: ** é uma categoria adicionada com a sugestão de um participante.

Baseando-se nas respostas obtidas com o estudo, é possível afirmar que a classificação de relevância ajudou a estabelecer uma visão mais clara das causas de vazamentos em aplicações móveis.

4.7 Ameaças à Validade

As ameaças à validade do estudo foram organizadas em 3 categorias: validade de constructo, interna e externa.

Validade de constructo: o estudo está caracterizado pela análise da relevância das categorias de causas de vazamentos no contexto de aplicações móveis. As categorias foram extraídas de estudos primários, por meio de um mapeamento sistemático da literatura, envolvendo experimentos com aplicações móveis.

Validade interna: propôs-se a selecionar profissionais de desenvolvimento que atuam/atuaram em desenvolvimento de aplicações móveis e que tenham experiência em vazamentos de recursos. Deste modo, assumiu-se que eles são representativos para a população do estudo e que podem dar a perspectiva dos desenvolvedores sobre as categorias de causas de vazamento.

O instrumento a ser utilizado (formulário online) passou por revisão e foi submetido a um estudo piloto que apontou melhorias que foram implementadas.

Validade externa: os participantes do estudo em geral podem ser considerados representativos para a população de profissionais de desenvolvimento, pois os dados do questionário sobre a experiência dos participantes foram utilizados para filtrar apenas participantes com o perfil esperado para este estudo.

4.8 Conclusões do Estudo

A análise de relevância de cada uma das 10 categorias, extraídas por meio do mapeamento sistemático, pelos participantes da pesquisa de opinião permitiu obter um conjunto base de 6 categorias (*Vazamento em caminhos excepcionais, Vazamento causado pela complexidade do ciclo de vida da aplicação, Vazamento causado por liberação inadequada de recursos em loop, Vazamento em caminhos irregulares, Vazamento causado por alocação e desalocação imprópria e Vazamento causado por condição de corrida*) mais relevantes para a representação de causas de vazamentos em aplicações móveis. Também foi possível adicionar uma nova categoria (*Vazamento de memória devido a APIs de terceiros mal construídas*) de causas de vazamentos. Essas informações são importantes para melhor

compreensão das origens dos vazamentos, o que permite propor uma estratégia eficaz para detectar a maioria dos vazamentos de recursos.

4.9 Considerações Finais

Neste capítulo foi apresentado uma pesquisa de opinião sobre a relevância das 10 categorias de causas de vazamentos de recursos identificadas no mapeamento sistemático. Com as informações apresentadas neste capítulo, percebe-se a importância da identificação de componentes com vazamentos em aplicações móveis. No próximo capítulo será apresentada a definição da abordagem proposta para a predição de vazamentos de recursos em aplicações móveis.

5

ABORDAGEM PROPOSTA

Este capítulo apresenta a estrutura e a descrição dos elementos que compõem a abordagem proposta neste trabalho.

5.1 Visão Geral da Proposta

A abordagem **LeakPred** (originada da expressão *Leak Prediction*; em português, Predição de Vazamento) foi projetada para identificar vazamentos de recursos em aplicações móveis Android utilizando aprendizado de máquina. Ela possui três processos principais: (1) Preparação dos dados, (2) Execução do treinamento e (3) Classificação dos componentes. Esses processos são apresentados na Figura 5.1 e discutidos nas próximas seções.

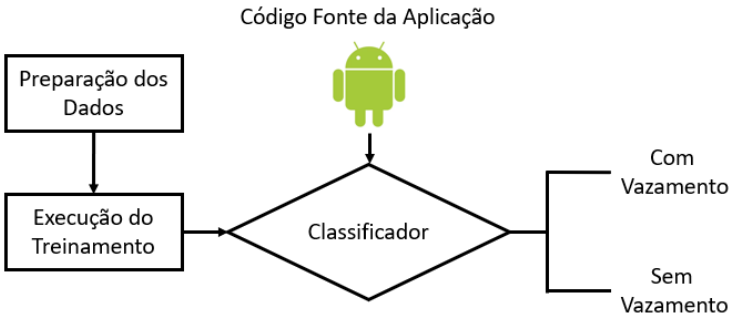


Figura 5.1 – Visão geral da abordagem *LeakPred*.

5.1.1 Preparação dos Dados

Foi realizada uma busca na literatura técnica e encontrou-se a base de dados *DroidLeaks* (LIU et al., 2019), que possui 292 vazamentos de recursos identificados em 32 aplicações Android de código aberto e de larga escala. Usamos essa base por ela ser confiável em avaliar e comparar técnicas de gerenciamento de recursos e técnicas de identificação de vazamentos. Algumas das informações que estão contidas nessa base são: nome da aplicação, nome da classe e nome do componente com defeito. Apesar disso, a quantidade de vazamentos na base é relativamente pequena. Por isso, optou-se por analisar 22 das 170 aplicações disponíveis em (LIU et al., 2019) para aumentar a base. Nesse processo, foram identificados 467 novos componentes com vazamentos de recursos. Esse processo é abordado no Capítulo 7.

Uma vez que a base não possui métricas extraídas para os componentes, montou-se uma nova base de dados. Para tal montagem, foi feita uma análise de um total de 107 métricas, sendo 30 métricas da ferramenta *JHawk*[1] e de 77 métricas da ferramenta *PMD*[2], tendo sido escolhidas seis (*returnTypeIsAResourceClass*, *calledCloseComponent*, *closeResource*, *overrideOnPause*, *overrideOnStop* e *overrideOnDestroy*), relacionadas ao problema, apresentadas na Tabela 5.1. As métricas foram selecionadas por um processo completamente dirigido aos dados. Primeiro, um algoritmo de seleção de características (*ExtraTreesClassifier*) foi empregado sobre o conjunto inicial de métricas extraídas para todos os componentes da base. Em seguida, as métricas selecionadas foram testadas por múltiplos classificadores. Finalmente, foi verificado se os modelos conseguiriam ter um bom aprendizado a partir dos dados com as métricas.

Após a extração das métricas para todos os componentes de cada classe que contenha ao menos um componente com vazamento de recursos, foi rotulado cada componente em nosso conjunto de dados como tendo vazamento de recurso ou não conforme a Equação 5.1.

$$Componente(x) = \begin{cases} 1, \text{ se } x \text{ contém vazamento de recurso (defeito)} \\ 0, \text{ caso contrário} \end{cases} \tag{5.1}$$

[1] <http://www.virtualmachinery.com/jhawkprod.htm>
[2] <https://pmd.github.io/latest/index.html>

Tabela 5.1 – Lista de métricas extraídas.

ID	Nome da Métrica	Descrição da Métrica	Valores	Ferramenta
M1	returnTypeIsAResourceClass	Se o tipo do retorno do componente é uma das classes de recurso, isso indica que o recurso deverá ser fechado por outro componente	0 ou 1	JHawk
M2	calledCloseComponent	Se o componente chama algum componente de fechamento de recurso	0 ou 1	JHawk
M3	closeResource	Se o componente não fecha o recurso	0 ou 1	PMD
M4	overrideOnPause	Se o *onPause* é sobrescrito na classe a qual o componente pertence e se o *onPause* referencia diretamente alguma classe de recurso	0 ou 1	JHawk
M5	overrideOnStop	Se o *onStop* é sobrescrito na classe a qual o componente pertence e se o *onStop* referencia diretamente alguma classe de recurso	0 ou 1	JHawk
M6	overrideOnDestroy	Se o *onDestroy* é sobrescrito na classe a qual o componente pertence e se o *onDestroy* referencia diretamente alguma classe de recurso	0 ou 1	JHawk

O resultado desse processo é a montagem parcial de uma nova base de dados chamada **CompLeaks** (originada da expressão *Component Leaks*; em português, vazamentos de componentes). Essa base parcial foi complementada pela verificação manual de componentes da *DroidLeaks* com valor 1 para a métrica *closeResource*. Isso enriqueceu a base, acrescentando 57 componentes com defeitos que não constavam na DroidLeaks, uma vez que a base original foi construída a partir de *commits* e só possui registro de componentes cujos defeitos de vazamento de recurso foram corrigidos até à época da montagem da base.

Finalmente, foram removidos os componentes que não se relacionam com alguma das 95 classes de recursos listadas na Tabela 5.2. Essa remoção visa diminuir os falsos-positivos, pois se o componente não se relaciona com nenhuma das classes de recursos mapeadas, provavelmente ele não terá vazamentos dos recursos mapeados. A lista de classes de recursos apresentada na Tabela 5.2 foi extraída a partir da base de dados *DroidLeaks* e da extensão dela com as 22 novas aplicações.

5.1.2 Execução do Treinamento

O objetivo deste trabalho é a identificação automática de componentes com vazamento de recurso com base nas métricas selecionadas. Assim, uma vez que a base de dados *CompLeaks* estava montada, o próximo passo foi treinar modelos de classificação.

Existem muitos modelos de classificação de aprendizado de máquina e cada um pode ser melhor indicado para uma aplicação ou tipo de dados diferente. Quando não se tem razão particular para preferir algum modelo sobre outro (um exemplo de

preferência seria quando se sabe se os dados são linearmente separáveis ou não), o ideal é testar e comparar diferentes modelos. Neste trabalho foram usados os seguintes classificadores: *SVM*, *Naive Bayes*, *DNN*, *Regressão Logística*, *Floresta Aleatória* e *KNN*. Esses classificadores são comumente utilizados em estudos para predição de defeito (EFENDIOGLU; SEN; KOROGLU, 2019; LI et al., 2019; MANJULA; FLORENCE, 2019; YUCALAR et al., 2019).

Os classificadores listados acima foram treinados utilizando a base de dados

Tabela 5.2 – Lista de classes de recurso.

ID	**Classe de Recurso**	**ID**	**Classe de Recurso**
ID	Classe de Recurso	ID	Classe de Recurso
CL1	android.app.AlarmManager	CL49	java.io.InputStream
CL2	android.app.NotificationManager	CL50	java.io.InputStreamReader
CL3	android.app.PendingIntent	CL51	java.io.ObjectInputStream
CL4	android.content.BroadcastReceiver	CL52	java.io.ObjectOutputStream
CL5	android.content.res.AssetFileDescriptor	CL53	java.io.OutputStream
CL6	android.content.res.TypedArray	CL54	java.io.OutputStreamWriter
CL7	android.database.Cursor	CL55	java.io.PipedOutputStream
CL8	android.database.sqlite.SQLiteDatabase	CL56	java.io.PrintStream
CL9	android.graphics.Bitmap	CL57	java.io.PrintWriter
CL10	android.graphics.Typeface	CL58	java.io.RandomAccessFile
CL11	android.hardware.Camera	CL59	java.io.Reader
CL12	android.hardware.SensorManager	CL60	java.io.Writer
CL13	android.location.LocationListener	CL61	java.lang.Thread
CL14	android.location.LocationManager	CL62	java.net.DatagramSocket
CL15	android.media.AudioManager	CL63	java.net.HttpURLConnection
CL16	android.media.MediaPlayer	CL64	java.net.Socket
CL17	android.net.http.AndroidHttpClient	CL65	java.net.URL
CL18	android.net.wifi.WifiManager	CL66	java.net.URLConnection
CL19	android.net.wifi.WifiManager.WifiLock	CL67	java.nio.channels.FileLock
CL20	android.nfc.tech.Ndef	CL68	java.sql.PreparedStatement
CL21	android.os.AsyncTask	CL69	java.sql.Statement
CL22	android.os.Parcel	CL70	java.util.concurrent.Semaphore
CL23	android.os.ParcelFileDescriptor	CL71	java.util.Formatter
CL24	android.os.PowerManager.WakeLock	CL72	java.util.logging.FileHandler
CL25	android.os.Vibrator	CL73	java.util.Scanner
CL26	android.renderscript.Allocation	CL74	java.util.Timer
CL27	android.renderscript.RenderScript	CL75	java.util.TimerTask
CL28	android.renderscript.ScriptIntrinsicBlur	CL76	java.util.zip.Deflater
CL29	android.view.MotionEvent	CL77	java.util.zip.DeflaterOutputStream
CL30	android.webkit.WebView	CL78	java.util.zip.ZipOutputStream
CL31	android.widget.BaseAdapter	CL79	javax.microedition.io.file.FileConnection
CL32	au.com.bytecode.opencsv.CSVReader	CL80	javax.microedition.io.HttpConnection
CL33	com.badlogic.gdx.audio.Sound	CL81	javax.microedition.io.StreamConnection
CL34	com.google.android.gms.games.leaderboard.LeaderboardScoreBuffer	CL82	javax.microedition.lcdui.Image
CL35	com.google.common.base.Stopwatch	CL83	javax.microedition.rms.RecordStore
CL36	com.path.android.jobqueue.JobManager	CL84	okhttp3.Response
CL37	java.io.BufferedInputStream	CL85	okhttp3.ResponseBody
CL38	java.io.BufferedOutputStream	CL86	org.apache.http.client.methods.HttpRequestBase
CL39	java.io.BufferedReader	CL87	org.apache.http.impl.client.DefaultHttpClient
CL40	java.io.BufferedWriter	CL88	org.bouncycastle.asn1.ASN1InputStream
CL41	java.io.ByteArrayOutputStream	CL89	org.bouncycastle.asn1.DERSequenceGenerator
CL42	java.io.DataInputStream	CL90	org.robovm.apple.coregraphics.CGBitmapContext
CL43	java.io.DataOutputStream	CL91	org.robovm.apple.coregraphics.CGImage
CL44	java.io.FileInputStream	CL92	org.robovm.apple.foundation.NSAutoreleasePool
CL45	java.io.FileOutputStream	CL93	org.robovm.rt.bro.ptr.IntPtr
CL46	java.io.FileWriter	CL94	org.xmlpull.v1.XmlPullParser
CL47	java.io.FilterInputStream	CL95	org.xmlpull.v1.XmlResourceParser
CL48	java.io.FilterOutputStream		

CompLeaks. Após isso, foi realizada uma análise de qual desses classificadores é o mais eficaz para o problema pesquisado. Esse classificador foi utilizado no processo de classificação dos componentes para as aplicações recebidas como artefato de entrada. A eficácia foi medida com as métricas acurácia (em inglês, *accuracy*), precisão (em inglês, *precision*), revocação (em inglês, *recall*), *ROC AUC* e *F-measure*.

5.1.3 Classificação dos Componentes

Para a utilização da abordagem proposta, é recebido como artefato de entrada o código fonte de uma aplicação Android, a qual deverá ter sido implementada na linguagem Java, pois as ferramentas *JHawk* e *PMD* que foram utilizadas para a extração das métricas têm compatibilidade com essa linguagem de programação.

Cada componente da aplicação é submetido ao processo de extração de métricas discutido na Seção 5.1.1. Essas métricas são dados de entrada para o melhor classificador selecionado na fase de treinamento e a saída do classificador será uma decisão sobre a existência ou não de defeitos de vazamento de recurso em cada componente da aplicação. Após a execução do classificador, os componentes com vazamentos serão apresentados no formato: pacote[3], classe [4], componente, parâmetros e retorno, como ilustrado na Tabela 5.3 (essas cinco informações sobre o componente irá facilitar o desenvolvedor encontrar o componente no código fonte). Com esse conjunto de componentes com vazamentos de recursos identificados, o desenvolvedor poderá implementar a correção do problema.

Tabela 5.3 – Exemplo de um conjunto de componentes com vazamento.

Pacote	**Classe**	**Componente**	**Parâmetros**	**Retorno**
Pacote.2	Classe2	Componente2	()	*String*
Pacote.3	Classe6	Componente5	(*int*)	*int*
Pacote.5	Classe11	Componente15	(*float*)	*float*
Pacote.8	Classe15	Componente23	()	*String*
Pacote.8	Classe4	Componente8	()	*int*
Pacote.10	Classe9	Componente12	(*String*)	*void*
Pacote.10	Classe10	Componente9	()	*void*

[3] Um pacote (em inglês, package) é um *namespace* usado para organizar um conjunto de classes relacionadas.

[4] Uma classe (em inglês, class) é um elemento do código utilizado para representar objetos do mundo real. Ela pode possuir atributos e componentes.

5.2 Considerações Finais

Neste capítulo foram apresentados os detalhes da abordagem proposta para predição de vazamentos de recursos em aplicações móveis. No próximo capítulo, será apresentado um estudo que irá avaliar o desempenho dos classificadores para a predição de componentes com vazamentos de recursos.

6

PROVA DE CONCEITO

6.1 Objetivo do Estudo

Este estudo tem como objetivo analisar a eficácia de classificadores (*SVM*, *Naive Bayes*, *DNN*, *Regressão Logística*, *Floresta Aleatória* e *KNN*) na predição de componentes com vazamentos de recursos. A eficácia será analisada pelas seguintes medidas: acurácia, precisão, revocação, *ROC AUC* e *F-measure*. Portanto, foi definida a questão de pesquisa abaixo:

"*Qual é o desempenho dos classificadores para modelos de predição de defeito (vazamento de recursos) no nível de componente usando métricas de código?*" A resposta dessa pergunta irá ajudar a escolher o melhor classificador para a predição de vazamento de recursos em aplicações móveis.

6.1.1 Base de Dados

A base de dados *CompLeaks* (baseada na base *DroidLeaks* e na inspeção manual de alguns componentes), usada para treinar os classificadores, contém 5.926 componentes, sendo que 313 desses componentes possuem vazamentos de recursos. Pode-se destacar que dos 292 vazamentos da base *DroidLeaks*, identificou-se 256 componentes com vazamentos, tendo 36 defeitos a menos. Isso se deve ao fato de alguns vazamentos de recursos ocorrerem em um bloco de código da classe que não pertencem a um componente e de outros vazamentos identificados estarem em locais diferentes em uma mesma versão

do componente. Também ressalta-se que na inspeção manual de alguns componentes para a montagem da base *CompLeaks*, foram identificados 57 novos componentes com vazamentos de recursos.

Após ser aplicado o filtro de remoção dos componentes que não referenciam alguma das 33 classes de recursos listadas na Tabela 5.2, a base ficou com 854 componentes sem vazamento e 256 componentes com vazamento, totalizando 1110 componentes. Essa base de dados, que será utilizada no experimento, está disponível em <http://encurtador.com.br/ehuV2>.

Tal conjunto de dados é altamente desbalanceado, pois o número de componentes com vazamento é muito pequeno comparado ao número de componentes sem vazamento. Um conjunto de dados desequilibrado pode levar a uma degradação no desempenho dos classificadores. Contudo, várias técnicas são propostas para evitar este tipo de situação. O método mais comumente usado é a técnica de sobreposição de minoria sintética (*SMOTE*) (EFENDIOGLU; SEN; KOROGLU, 2019; CHAWLA et al., 2002). Por esse motivo, ela foi utilizada com sua implementação disponibilizada pela biblioteca *imbalanced-learn*[1]. Além disso, os dados foram normalizados para o uso nos classificadores. Na próxima subseção será explicado como foram otimizados alguns hiperparâmetros dos classificadores utilizados neste estudo.

6.1.2 Otimização de Hiperparâmetros

Como pré-requisito para a construção de um modelo de aprendizado de máquina, é necessário definir os hiperparâmetros que serão utilizados em cada classificador. Este ajuste pode ser feito de forma automática. Neste trabalho foi utilizado o método *Grid Search*[2] com a implementação da biblioteca *scikit-learn*[3], pois ele é utilizado em trabalhos recentes, como em (DIJKHUIS et al., 2018) e (FINDLAY et al., 2018).

Neste método, considera-se exaustivamente todas as combinações de parâmetros.

[1] <https://pypi.org/project/imbalanced-learn/>

[2] <https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html>

[3] A *scikit-learn* é uma biblioteca de aprendizado de máquina de código aberto para a linguagem de programação Python

Em seguida, o algoritmo inicia um aprendizado para cada uma das configurações de hiperparâmetros e seleciona a melhor no final. Abaixo são mostrados os hiperparâmetros que serão otimizados para cada classificador utilizado nesta pesquisa. Na próxima subseção serão explicados os passos seguidos para a execução do estudo.

1. Para o *SVM*, foram testadas três variações: a linear, a Gaussiana (rbf) e a sigmoid. Na versão linear, o hiperparâmetro C foi ajustado com valores de 0,001, 0,01, 0,1, 1 e 10. Para a variação Gaussiana, além de C com os mesmos valores, o parâmetro gama também foi explorado com valores de 0,001, 0,01, 0,1 e 1. A configuração sigmoid seguiu o mesmo padrão de valores para C e gama;

2. O modelo *Naive Bayes* foi analisado considerando três diferentes distribuições: Multinomial, Gaussiana e Bernoulli;

3. Para o *DNN*, os experimentos foram realizados com diferentes funções de ativação: softmax, relu e tanh. Além disso, diversos otimizadores (SGD, Adam e Nadam) foram testados, com um número fixo de épocas de 50 e tamanhos de lote de 10, 20 e 30;

4. No caso da *Regressão Logística*, foram testados dois solucionadores: liblinear e saga, combinados com as penalidades l1 e l2;

5. O modelo de *Floresta Aleatória* foi avaliado com diferentes números de estimadores (n_estimadores) nos valores de 100, 300, 500, 800 e 1000, além de variações no uso de bootstrap (verdadeiro e falso) e no critério de avaliação, considerando tanto gini quanto entropia;

6. Por fim, o *KNN* foi ajustado com diferentes números de vizinhos (n_vizinhos) nos valores de 1, 3, 5, 7, 9 e 11, e considerando dois tipos de pesos: uniforme e distância.

6.1.3 Execução do Estudo

A fim de estimar a capacidade de generalização dos classificadores, todos os modelos de predição foram treinados e validados usando validação cruzada em 10 vias (10-*fold cross-validation*), com base em amostragem estratificada, garantindo que a distribuição de classes nos subconjuntos seja a mesma em todo o conjunto de dados.

O processo de validação cruzada em 10 vias consiste em dividir o conjunto de treinamento em 10 subconjuntos. Para cada configuração de modelo e hiper-parâmetro, são treinados e testados 10 modelos. Em cada passo desse processo, um dos subconjuntos é reservado para testar o modelo que foi treinado nos outros nove. Dessa forma, cada exemplo do conjunto de treinamento é usado uma única vez em um conjunto de teste e em nenhum momento há sobreposição entre dados de treinamento e de teste.

6.1.4 Análise dos Resultados

Na Tabela 6.1 são mostrados os melhores hiperparâmetros para cada classificador em cada uma das 10 execuções, como também a pontuação, a qual é a média da validação cruzada (nos nove subconjuntos utilizados no treino do modelo) dos melhores hiperparâmetros informados pelo *GridSearchCV* a cada execução. Para o *SVM*, a otimização foi realizada separadamente para os *kernels linear*, *Gaussiano* (*rbf*) e *sigmoid*, sendo que a melhor pontuação no treino, considerando a média das 10 execuções, foi para o *kernel Gaussiano* (*rbf*), com 76,58%, pois os *kernels linear* e *sigmoid* ficaram, nessa ordem, com 74,40% e 74,54%. Para o *Naive Bayes*, foram analisadas as distribuições *Multinomial*, *Gaussiano* e *Bernoulli*, que obtiveram a média na pontuação do treino de, respectivamente, 72,04%, 68,37% e 69,74%, ou seja, a distribuição *Multinomial* obteve o melhor resultado. A média para a *Regressão Logística*, *Floresta Aleatória*, *KNN* e *DNNs* são, respectivamente, 70,96%, 76,79%, 76,33% e 74,86%.

Tabela 6.1 – Melhores hiperparâmetros por execução

Execução	**Classificador**	**Melhores Hirperparâmetros**	**Pontuação(%)**
1	*SVM (kernel = linear)*	{'C': 0,1}	78,32%
1	*SVM (kernel = Gaussiano (rbf))*	{'C': 1, 'gama': 1}	79,36%
1	*SVM (kernel = sigmoid)*	{'C': 1, 'gama': 0,01}	78,39%

Tabela - 6.1 continuação da página anterior

Execução	Classificador	Melhores Hirperparâmetros	Pontuação(%)
1	*Naive Bayes Multinomial*	{}	75,65%
1	*Naive Bayes Gaussiano*	{}	72,79%
1	*Naive Bayes Bernoulli*	{}	71,48%
1	*Regressão Logística*	{'penalidade': 'l1', 'solucionador': 'liblinear'}	78,52%
1	*Floresta Aleatória*	{'bootstrap': verdadeiro, 'critério': 'gini', 'n_estimadores': 300}	79,56%
1	*KNN*	{'n_vizinhos': 9, 'pesos': 'distância'}	79,10%
1	*DNN*	{'ativação': 'relu', 'tamanho do lote': 20, 'épocas': 50, 'otimizador': 'Adam'}	79,36%
2	*SVM (kernel = linear)*	{'C': 0,1}	74,09%
2	*SVM (kernel = Gaussiano (rbf))*	{'C': 10, 'gama': 1}	75,33%
2	*SVM (kernel = sigmoid)*	{'C': 10, 'gama': 0,01}	73,83%
2	*Naive Bayes Multinomial*	{}	69,99%
2	*Naive Bayes Gaussiano*	{}	70,96%
2	*Naive Bayes Bernoulli*	{}	68,55%
2	*Regressão Logística*	{'penalidade': 'l2', 'solucionador': 'liblinear'}	68,03%
2	*Floresta Aleatória*	{'bootstrap': verdadeiro, 'critério': 'entropia', 'n_estimadores': 300}	75,59%
2	*KNN*	{'n_vizinhos': 9, 'pesos': 'distância'}	74,93%
2	*DNN*	{'ativação': 'softmax', 'tamanho do lote': 10, 'épocas': 50, 'otimizador': 'Adam'}	69,21%
3	*SVM (kernel = linear)*	{'C': 1}	75,07%
3	*SVM (kernel = Gaussiano (rbf))*	{'C': 10, 'gama': 0,1}	76,95%
3	*SVM (kernel = sigmoid)*	{'C': 0,001, 'gama': 0,01}	73,89%
3	*Naive Bayes Multinomial*	{}	74,02%
3	*Naive Bayes Gaussiano*	{}	71,55%
3	*Naive Bayes Bernoulli*	{}	70,77%
3	*Regressão Logística*	{'penalidade': 'l1', 'solucionador': 'liblinear'}	69,40%
3	*Floresta Aleatória*	{'bootstrap': verdadeiro, 'critério': 'gini', 'n_estimadores': 1000}	77,60%
3	*KNN*	{'n_vizinhos': 7, 'pesos': 'distância'}	76,89%
3	*DNN*	{'ativação': 'softmax', 'tamanho do lote': 10, 'épocas': 50, 'otimizador': 'Nadam'}	76,82%
4	*SVM (kernel = linear)*	{'C': 1}	70,25%
4	*SVM (kernel = Gaussiano (rbf))*	{'C': 10, 'gama': 1}	73,83%
4	*SVM (kernel = sigmoid)*	{'C': 10, 'gama': 0,1}	70,57%
4	*Naive Bayes Multinomial*	{}	66,08%
4	*Naive Bayes Gaussiano*	{}	67,84%
4	*Naive Bayes Bernoulli*	{}	66,15%
4	*Regressão Logística*	{'penalidade': 'l1', 'solucionador': 'liblinear'}	65,69%
4	*Floresta Aleatória*	{'bootstrap': verdadeiro, 'critério': 'gini', 'n_estimadores': 500}	74,02%
4	*KNN*	{'n_vizinhos': 9, 'pesos': 'distância'}	73,50%
4	*DNN*	{'ativação': 'relu', 'tamanho do lote': 30, 'épocas': 50, 'otimizador': 'Nadam'}	73,05%
5	*SVM (kernel = linear)*	{'C': 0,01}	75,23%
5	*SVM (kernel = Gaussiano (rbf))*	{'C': 10, 'gama': 0,01}	76,01%
5	*SVM (kernel = sigmoid)*	{'C': 1, 'gama': 0,01}	75,23%

Tabela - 6.1 continuação da página anterior

Execução	Classificador	Melhores Hirperparâmetros	Pontuação(%)
5	*Naive Bayes Multinomial*	{}	71,07%
5	*Naive Bayes Gaussiano*	{}	71,20%
5	*Naive Bayes Bernoulli*	{}	66,58%
5	*Regressão Logística*	{'penalidade': 'l1', 'solucionador': 'liblinear'}	75,29%
5	*Floresta Aleatória*	{'bootstrap': verdadeiro, 'critério': 'entropia', 'n_estimadores': 500}	75,88%
5	*KNN*	{'n_vizinhos': 11, 'pesos': 'distância'}	75,88%
5	*DNN*	{'ativação': 'relu', 'tamanho do lote': 30, 'épocas': 50, 'otimizador': 'Nadam'}	75,68%
6	*SVM (kernel = linear)*	{'C': 0,1}	77,11%
6	*SVM (kernel = Gaussiano (rbf))*	{'C': 10, 'gama': 0,1}	78,15%
6	*SVM (kernel = sigmoid)*	{'C': 10, 'gama': 0,01}	77,18%
6	*Naive Bayes Multinomial*	{}	75,62%
6	*Naive Bayes Gaussiano*	{}	71,59%
6	*Naive Bayes Bernoulli*	{}	70,81%
6	*Regressão Logística*	{'penalidade': 'l2', 'solucionador': 'liblinear'}	72,82%
6	*Floresta Aleatória*	{'bootstrap': falso, 'critério': 'gini', 'n_estimadores': 500}	78,35%
6	*KNN*	{'n_vizinhos': 5, 'pesos': 'distância'}	78,09%
6	*DNN*	{'ativação': 'softmax', 'tamanho do lote': 10, 'épocas': 50, 'otimizador': 'Nadam'}	77,70%
7	*SVM (kernel = linear)*	{'C': 0,1}	73,02%
7	*SVM (kernel = Gaussiano (rbf))*	{'C': 10, 'gama': 0,1}	77,05%
7	*SVM (kernel = sigmoid)*	{'C': 10, 'gama': 0,1}	73,47%
7	*Naive Bayes Multinomial*	{}	69,96%
7	*Naive Bayes Gaussiano*	{}	69,96%
7	*Naive Bayes Bernoulli*	{}	69,70%
7	*Regressão Logística*	{'penalidade': 'l1', 'solucionador': 'liblinear'}	68,53%
7	*Floresta Aleatória*	{'bootstrap': verdadeiro, 'critério': 'entropia', 'n_estimadores': 100}	76,98%
7	*KNN*	{'n_vizinhos': 7, 'pesos': 'distância'}	76,40%
7	*DNN*	{'ativação': 'relu', 'tamanho do lote': 30, 'épocas': 50, 'otimizador': 'Adam'}	76,07%
8	*SVM (kernel = linear)*	{'C': 1}	76,92%
8	*SVM (kernel = Gaussiano (rbf))*	{'C': 10, 'gama': 0,1}	78,15%
8	*SVM (kernel = sigmoid)*	{'C': 10, 'gama': 0,01}	76,79%
8	*Naive Bayes Multinomial*	{}	74,71%
8	*Naive Bayes Gaussiano*	{}	53,38%
8	*Naive Bayes Bernoulli*	{}	71,65%
8	*Regressão Logística*	{'penalidade': 'l1', 'solucionador': 'liblinear'}	73,28%
8	*Floresta Aleatória*	{'bootstrap': falso, 'critério': 'gini', 'n_estimadores': 100}	78,54%
8	*KNN*	{'n_vizinhos': 7, 'pesos': 'distância'}	78,02%
8	*DNN*	{'ativação': 'relu', 'tamanho do lote': 20, 'épocas': 50, 'otimizador': 'Adam'}	77,57%
9	*SVM (kernel = linear)*	{'C': 1}	72,04%
9	*SVM (kernel = Gaussiano (rbf))*	{'C': 10, 'gama': 1}	75,16%
9	*SVM (kernel = sigmoid)*	{'C': 10, 'gama': 0,1}	72,63%

Tabela - 6.1 continuação da página anterior

Execução	Classificador	Melhores Hirperparâmetros	Pontuação(%)
9	*Naive Bayes Multinomial*	{}	72,43%
9	*Naive Bayes Gaussiano*	{}	68,08%
9	*Naive Bayes Bernoulli*	{}	69,96%
9	*Regressão Logística*	{'penalidade': 'l2', 'solucionador': 'liblinear'}	67,69%
9	*Floresta Aleatória*	{'bootstrap': verdadeiro, 'critério': 'gini', 'n_estimadores': 100}	75,68%
9	*KNN*	{'n_vizinhos': 11, 'pesos': 'distância'}	75,16%
9	*DNN*	{'ativação': 'relu', 'tamanho do lote': 10, 'épocas': 50, 'otimizador': 'SGD'}	73,99%
10	*SVM (kernel = linear)*	{'C': 0,001}	71,91%
10	*SVM (kernel = Gaussiano (rbf))*	{'C': 10, 'gama': 0,1}	75,81%
10	*SVM (kernel = sigmoid)*	{'C': 0,001, 'gama': 0,01}	73,41%
10	*Naive Bayes Multinomial*	{}	70,87%
10	*Naive Bayes Gaussiano*	{}	66,32%
10	*Naive Bayes Bernoulli*	{}	71,72%
10	*Regressão Logística*	{'penalidade': 'l1', 'solucionador': 'liblinear'}	70,35%
10	*Floresta Aleatória*	{'bootstrap': verdadeiro, 'critério': 'gini', 'n_estimadores': 100}	75,68%
10	*KNN*	{'n_vizinhos': 7, 'pesos': 'distância'}	75,36%
10	*DNN*	{'ativação': 'relu', 'tamanho do lote': 20, 'épocas': 50, 'otimizador': 'SGD'}	69,18%

A acurácia é a fração de componentes corretamente previstos (Equação 2.2). De modo geral, em uma base de dados balanceada, quanto maior a acurácia, melhor o desempenho do modelo. A média da acurácia de teste para os classificadores é apresentada na Figura 6.1, onde é possível observar que o classificador *KNN* obteve a melhor acurácia (87,84%), seguido pelos classificadores *DNN* (87,75%), *SVM (kernel = Gaussiano (rbf))* (87,66%) e *Floresta Aleatória* (87,66%).

A matriz de confusão (que é o resultado da soma das matrizes de cada execução) para os classificadores é mostrada na Tabela 6.2, onde se percebe que o *KNN* e a *DNN* têm a mesma quantidade de verdadeiros-negativos (VN) e falsos-positivos (FP). No entanto, o *KNN* tem um verdadeiro-positivo (VP) a mais do que a *DNN* e esta tem um falso-negativo (FN) a mais do que aquele.

A precisão (fração de instâncias recuperadas que são relevantes - Equação 2.3), a revocação (fração de instâncias relevantes que são recuperadas - Equação 2.4) e a *F-measure* (média harmónica ponderada da precisão e revocação - Equação 2.5) podem ser calculadas a partir dos valores constantes da Tabela 6.2. Na Tabela 6.3 observa-se que o *KNN* possui os melhores valores para as três métricas, seguido pelo classificador

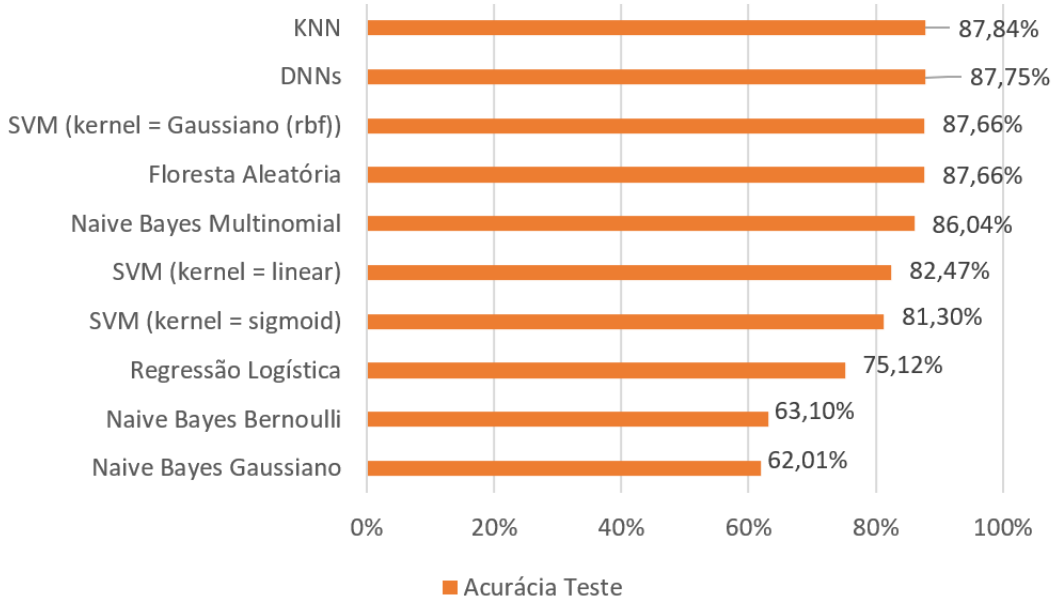


Figura 6.1 – Acurácia de teste para os classificadores.

DNN, que possui o segundo melhor valor para as métricas. Isso indica que eles têm uma boa capacidade de previsão de vazamentos de recursos.

Tabela 6.2 – Matriz de confusão dos classificadores.

Classificador	**VN**	**VP**	**FN**	**FP**
SVM (kernel = linear)	783	133	123	71
SVM (kernel = Gaussiano (rbf))	838	135	121	16
SVM (kernel = sigmoid)	775	128	128	79
Naive Bayes Multinomial	830	125	131	24
Naive Bayes Gaussiano	501	188	68	353
Naive Bayes Bernoulli	505	196	60	349
Regressão Logística	684	151	105	170
Floresta Aleatória	841	132	124	13
KNN	840	135	121	14
DNN	840	134	122	14

Na Figura 6.2 é mostrada a média da área sob a curva ROC (ROC AUC) a partir das pontuações de predição de cada classificador. Analisando esses dados, é possível observar que a *DNN* obteve a melhor área sob a curva (78,93%), sendo seguida pelos classificadores *Regressão Logística* (78,67%), *Floresta Aleatória* (78,33%) e *SVM (kernel = Gaussiano (rbf))* (78,05%). O classificador *KNN* possui a sétima melhor área sob a curva com 75,01%.

Tabela 6.3 – Métricas para os classificadores.

Classificador	**Precisão**	**Revocação**	**F-measure**
SVM (kernel = linear)	85,57%	82,47%	81,05%
SVM (kernel = Gaussiano (rbf))	87,62%	87,66%	86,13%
SVM (kernel = sigmoid)	84,01%	81,30%	79,87%
Naive Bayes Multinomial	86,24%	86,04%	84,51%
Naive Bayes Gaussiano	76,67%	62,01%	64,15%
Naive Bayes Bernoulli	78,21%	63,10%	65,11%
Regressão Logística	81,61%	75,12%	74,78%
Floresta Aleatória	87,93%	87,66%	86,02%
KNN	88,04%	87,84%	86,28%
DNN	87,99%	87,75%	86,17%

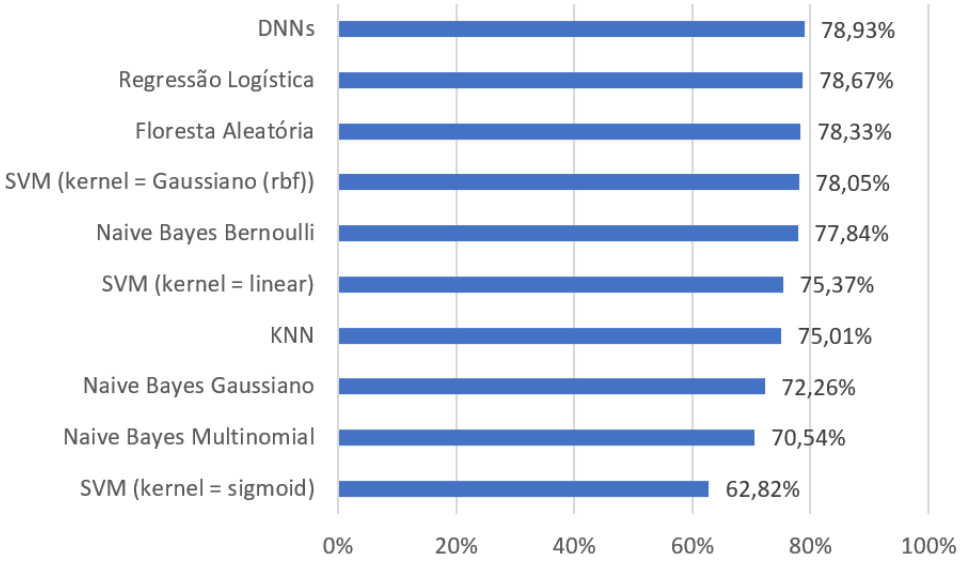


Figura 6.2 – A área sob a curva de característica de operação do receptor (*ROC AUC*) para os classificadores.

Com as informações apresentadas nesta subseção, acredita-se que o classificador *KNN* seja o mais adequado a ser utilizado para a identificação de componentes com vazamentos de recursos em aplicações móveis, pois ele obteve os melhores valores para a maioria das métricas acurácia, precisão, revocação e *f-measure*. Apenas para a métrica *ROC AUC* não se obteve o melhor valor.

6.1.5 Ameaças à Validade

Esta subseção apresenta algumas ameaças quanto à validade dos resultados obtidos neste estudo.

Validade de construção: a base de dados utilizada não é balanceada. Esse desbalanceamento impacta negativamente o treinamento dos modelos. Para diminuir essa ameaça nos resultados do estudo, foi utilizada a técnica *SMOTE* para balancear a base de dados de treinamento. No entanto, essa técnica não foi usada para alterar a base utilizada para o teste.

Validade interna: observando-se o tamanho da base de dados e com o intuito de minimizar essa ameaça, foi realizado o experimento com validação cruzada de 10 *folds*, e em cada iteração foi usado um conjunto diferente de dados para treinamento e teste.

Validade externa: destaca-se a utilização de somente uma base de dados para condução do experimento. Portanto, os resultados podem não ser generalizados e estudos adicionais podem ser conduzidos para uma análise mais aprofundada. No entanto, por se tratar de uma base de dados encontrada na literatura técnica, acredita-se que os resultados sejam válidos.

6.1.6 Conclusões do Estudo

Os resultados mostraram a possibilidade de usar aprendizado de máquina para identificar vazamentos de recursos em aplicações móveis. Essa conclusão foi fundamentada no resultado obtido no estudo realizado em (PEREIRA et al., 2022), o qual versa sobre uma técnica de análise estática (em um ambiente experimental diferente do utilizado nesse trabalho) para a identificação de vazamentos em aplicações móveis, tendo sido obtida uma precisão de 72,5%. No presente estudo, por sua vez, o classificador *KNN* teve uma precisão de 88,04%, seguido pela *DNN* com 87,99%. Esses resultados, somados o melhor resultado (78,93%) na métrica ROC AUC e o segundo melhor resultado em acurácia (87,75%), revocação (87,75%) e f-measure (86,17%) foram importantes para escolher o classificador *DNN* para ser utilizado na abordagem proposta.

6.2 Considerações Finais

Neste capítulo foi apresentado um estudo com o objetivo de analisar a eficácia de alguns classificadores na tarefa de predição de vazamentos, tendo sido escolhido para utilização na abordagem o classificador *DNN*. No próximo capítulo, serão apresentados o planejamento, a execução e os resultados do estudo de viabilidade que visa avaliar a abordagem *LeakPred*.

7

ESTUDO DE VIABILIDADE

7.1 Objetivo do Estudo

Este estudo de viabilidade tem como finalidade a comparação da identificação de componentes com vazamentos de recursos pela abordagem *LeakPred* com ferramentas do estado da arte, na qual foram usadas aplicações móveis da plataforma Android. Com isso, espera-se que os resultados obtidos e o corpo de conhecimento decorrente de sua condução forneçam informações que permitam evoluir a abordagem e melhorar seu uso na identificação de componentes com vazamentos em aplicações móveis Android.

O propósito deste estudo é responder à seguinte questão: "A utilização da abordagem *LeakPred* é viável, analisando a sua eficácia em comparação a outras ferramentas que representam o estado da arte na identificação de componentes com vazamentos de recursos em aplicações móveis na plataforma Android?". Por eficácia, entende-se neste estudo como a quantidade de componentes com vazamentos de recursos identificados em relação à quantidade total de componentes com vazamentos de recursos nas aplicações móveis. Nesse contexto, foi comparada a lista de componentes identificados com vazamentos de recursos pela abordagem *LeakPred* com a lista identificada por cada uma das ferramentas do estado da arte. Assim sendo, a questão de pesquisa definida para este estudo foi:

"*Qual é a eficácia da abordagem LeakPred em relação à identificação realizada pelas ferramentas do estado da arte no que diz respeito à quantidade de componentes com vazamentos de recursos identificados?*"

Para ajudar a responder à questão de pesquisa, foram escolhidas duas métricas:

a taxa de verdadeiros-positivos (em inglês, *True Positive Rate - TPR*) a qual é a probabilidade de um componente com vazamento ser identificado como tendo vazamento (Equação 7.1) e a taxa de falsos-positivos (em inglês, *False Discovery Rate - FDR*) a qual é a razão esperada do número de componentes sem vazamentos identificados como tendo vazamento (falsas descobertas) para o número total de componentes com vazamento identificados (Equação 7.2). Nas Equações 7.1 e 7.2, *VP* é a quantidade de componentes com vazamentos identificados corretamente, *P* é a quantidade de vazamentos que poderiam ser identificados e *FP* é a quantidade de componentes sem vazamentos identificados como tendo vazamento.

$$TaxaVerdadeiroPositivo = \frac{VP}{P} \tag{7.1}$$

$$TaxaFalsoPositivo = \frac{FP}{FP + VP} \tag{7.2}$$

7.1.1 Metodologia para a Identificação dos Componentes com Vazamento de Recursos

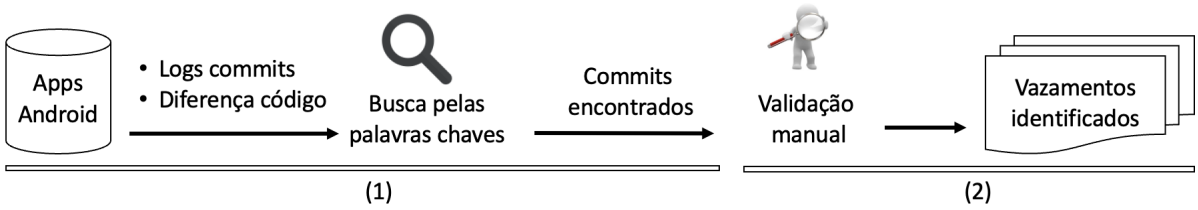


Figura 7.1 – Processo de identificação de vazamento de recursos

Para a identificação dos componentes com vazamento de recursos, foi seguido o mesmo protocolo utilizado para montar a base de dados *DroidLeaks* em (LIU et al., 2019). A Figura 7.1 mostra a visão geral do processo seguido formado por duas etapas: (1) busca de palavras-chave nos logs dos commits (Algumas palavras-chave de exemplo aparecem na Tabela 7.1) e nas diferenças de código do commit (Exemplos de palavras-chave são mostradas na Tabela 7.1); e (2) validação manual dos vazamentos de recursos

Tabela 7.1 – Palavras-chave para logs dos commits e diferenças de código.

Logs dos commits		
leak	leakage	release
recycle	cancel	unload
unlock	unmount	unregister
close		
Diferenças de código		
.close(	.stop(	.stopPreview(
.release(	.abandonAudioFocus(	.stopFaceDetection(
.removeUpdates(	.cancel(	.unregisterListener(
.unlock(	.disableNetwork(	

identificados nos commits encontrados na etapa 1. O processo apresentado nessa subseção será seguido nas aplicações selecionadas na próxima subseção, e algumas das aplicações com vazamento de recursos identificados serão utilizadas para a avaliação da eficácia da abordagem *LeakPred*.

7.1.2 Seleção das Aplicações Móveis

Liu et al. (2019) disponibilizam uma lista com 170 aplicações móveis que atendem aos seguintes critérios:

1. possuem mais de 10.000 *downloads* na loja (a aplicação é popular);
2. possuem um sistema de rastreamento de defeitos público (defeitos são rastreáveis);
3. o repositório de código da aplicação tem mais de 100 revisões de código (a aplicação é mantida ativamente), e;
4. possuem pelo menos 1.000 linhas de código fonte Java (a aplicação possui um médio ou alto nível de complexidade).

Dessas 170 aplicações, Liu et al. (2019) selecionaram 34 para elaborar a base *DroidLeaks*. Das 136 remanescentes, foram selecionadas aleatoriamente 32 aplicações para a execução deste trabalho de doutorado, sendo que uma fração destas foi utilizada para a construção da base de dados *CompLeaks* e o restante foi reservado para o estudo

de viabilidade[1]. Para melhor entendimento, a lista das 32 aplicações escolhidas para esta tese pode ser encontrada na Tabela 7.2 e o diagrama na Figura 7.2 mostra o número exato de aplicações usadas apenas na base *DroidLeaks*, apenas na base *CompLeaks*, em ambas as bases e no estudo de viabilidade. Observe que o estudo de viabiliade envolve 5 aplicações da base *CompLeaks* e 10 aplicações que não estavam presentes em nenhuma das bases. Estas 15 não foram utilizadas como conjunto de treinamento da abordagem *LeakPred*, pois foram usadas para a avaliação da abordagem.

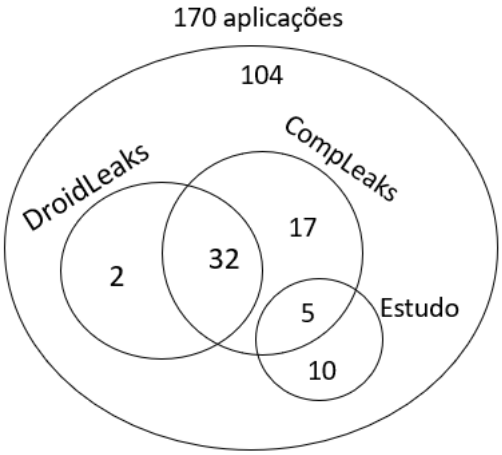


Figura 7.2 – Seleção das Aplicações Móveis.

7.1.3 Seleção das Ferramentas

A abordagem *LeakPred* foi comparada com as seguintes ferramentas do estado da arte para detecção de vazamentos de recursos em aplicações móveis:

1. **Android Lint** que fornece uma varredura de código que ajuda a identificar vazamentos de recursos e outros problemas estruturais de código (ANDROID, 2024);

2. **FindBugs** é um programa que usa análise estática para procurar bugs (incluindo vazamentos de recursos) no código Java (PUGH; LOSKUTOV; LEA, 2024);

3. **Infer** verifica vazamentos de recursos e outros defeitos (FACEBOOK, 2024);

[1] Mais informações, como nome do componente, classe de recurso com vazamento, são encontradas em <https://bit.ly/3vGnSyG>.

Tabela 7.2 – As 32 aplicações selecionadas.

ID	Nome aplicação	Quantidade vazamento	Usada	Compilado com sucesso
1	Aard	8	database/experimento	não
2	Android-Project	20	database	**
3	BART_Runner	9	database	**
4	BeeCount	13	database	**
5	BeTrains-for-Android	52	database/experimento	não
6	BetterWeather	1	database	**
7	Bitcoin	26	database	**
8	BombusMod	78	database	**
9	Dimmer	1	database	**
10	FareBot	3	database/experimento	não
11	FBReader TTS+ Plugin	10	database	**
12	GPSLogger	74	database	**
13	KeepScore	2	database	**
14	Navit	9	database	**
15	OI Notepad	12	database/experimento	sim
16	OI Safe	30	database/experimento	sim
17	Pedometer	41	database	**
18	Public Transport Timisoara	21	database	**
19	RedReader Beta	28	database	**
20	Shattered Pixel Dungeon	25	database	**
21	WiFi Analyzer	3	database	**
22	Yubico Authenticator	1	database	**
23	AnyMemo	12	experimento	sim
24	Avare	22	experimento	sim
25	OpenDocument Reader	-	experimento	não
26	Tinfoil for Facebook	-	experimento	não
27	Wikipedia	-	experimento	não
28	Seafile	42	experimento	sim
29	SMS Backup+	-	experimento	não
30	Chess	-	experimento	não
31	EP Mobile	-	experimento	não
32	Persian Calendar	-	experimento	não

Legenda: Onde está um "-" significa que a quantidade de vazamentos de recursos nessa aplicação é desconhecida. Onde aparece "**" significa que não foi tentado compilar essa aplicação.

4. **Checker Framework** identifica vazamentos de recursos e sugere melhorias para aumentar a qualidade do código (KELLOGG et al., 2021);

5. **EcoAndroid** detecta vazamentos de recursos, como também, sugere refatorações automatizadas para reduzir o consumo de energia de aplicações Java Android (PEREIRA et al., 2022).

7.1.4 Execução do Estudo

A execução do estudo teve 3 passos: (1) compilação das aplicações (algumas ferramentas precisam da aplicação compilada); (2) execução das ferramentas; e (3) relacionamento das classes de vazamentos de recursos por ferramenta. Essas etapas são apresentadas na Figura 7.3.

Uma consideração importante é que não existe um padrão para os processos de

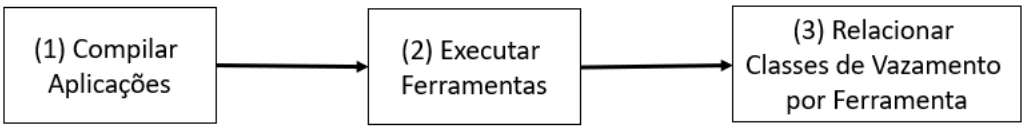


Figura 7.3 – Processo de execução do estudo.

compilação das aplicações ou para execução das ferramentas de análise de vazamento. Cada aplicação pode exigir um processo diferente para ser compilada e cada ferramenta demanda esforço próprio para que seja configurado seu ambiente de execução.

No primeiro passo tentou-se realizar a compilação das 15 aplicações selecionadas para este estudo (as aplicações selecionadas estão marcadas na Tabela 7.2 com a palavra "experimento" na coluna "Usada"). As aplicações com fundo cinza na tabela e com os IDs *1*, *5*, *10*, *25*, *26*, *27*, *29*, *30*, *31* e *32* não foram compiladas devido a dependências de bibliotecas e erros de projeto. Apenas foi possível compilar as aplicações que estão com o fundo verde na tabela, a saber, *OI Notepad (15)*, *OI Safe (16)*, *AnyMemo (23)*, *Avare (24)* e *Seafile (28)*. Como algumas ferramentas de análise exigem que a aplicação esteja compilada, o estudo de viabilidade foi realizado com essas 5 aplicações.

O segundo passo foi executar as ferramentas do estado da arte. As ferramentas *Android Lint*, *FindBugs* e *Infer* foram executadas com sucesso. No entanto, não foi possível configurar e executar as ferramentas *Checker Framework* e *EcoAndroid* em tempo razoável. Portanto, ambas não foram utilizadas no estudo.

O terceiro passo foi fazer a relação de classes de vazamento de recursos que cada ferramenta poderia identificar nas cinco aplicações (Tabela 7.3), pois cada ferramenta identifica algumas classes de vazamentos de recursos e nem todas as ferramentas disponibilizam uma lista das classes de recursos que elas podem identificar. Para isso, foi usada a lista de vazamentos identificados por ao menos uma das ferramentas nas cinco aplicações e a relação de vazamentos que cada ferramenta poderia identificar mapeada em *DroidLeaks* (LIU et al., 2019). Na próxima subseção será mostrada a análise dos resultados, sendo que a taxa de detecção de cada ferramenta foi baseada apenas nos vazamentos que ela poderia identificar em cada aplicação.

Tabela 7.3 – As classes de vazamento de recursos que poderiam ser identificadas por cada ferramenta nas aplicações OI Notepad, OI Safe, AnyMemo, Avare e Seafile.

Concerned java class	LeakPred	Infer	FindBugs	Android Lint
android.app.AlarmManager	X	-	-	-
android.app.NotificationManager	X	-	-	-
android.app.Service	X	-	-	-
android.bluetooth.BluetoothSocket	X	-	-	-
android.content.BroadcastReceiver	X	-	-	-
android.content.res.XmlResourceParser	X	-	-	-
android.content.ServiceConnection	X	-	-	-
android.database.Cursor	X	X	-	X
android.database.sqlite.SQLiteDatabase	X	X	-	-
android.databaseMatrixCursor	-	X	-	-
android.graphics.Bitmap	X	X	X	-
android.hardware.SensorManager	X	-	-	-
android.location.LocationManager	X	-	X	-
android.media.MediaMetadataRetriever	X	-	X	-
android.media.MediaPlayer	X	-	-	-
android.media.SoundPool	X	-	X	-
android.os.AsyncTask	X	-	-	-
android.os.CountDownTimer	X	-	-	-
android.speech.tts.TextToSpeech	X	-	-	-
android.support.v4.app.NotificationManagerCompat	X	-	-	-
com.github.kevinsawicki.http.HttpRequest	X	-	-	-
java.io.BufferedInputStream	X	X	X	-
java.io.BufferedOutputStream	X	X	X	-
java.io.BufferedReader	X	-	X	-
java.io.BufferedWriter	X	X	X	-
java.io.DataInputStream	X	X	-	-
java.io.DataOutputStream	X	-	X	-
java.io.File	X	-	-	-
java.io.FileInputStream	X	X	X	-
java.io.FileOutputStream	X	X	X	-
java.io.FileReader	-	X	-	-
java.io.FileWriter	X	X	-	-
java.io.InputStream	X	X	X	-
java.io.InputStreamReader	X	X	X	-
java.io.OutputStream	X	X	X	-
java.io.PrintWriter	X	-	-	-
java.io.RandomAccessFile	X	X	-	-
java.io.Reader	X	X	-	-
java.io.Writer	X	X	-	-
java.lang.Thread	X	-	X	-
java.net.DatagramPacket	X	-	-	-
java.net.DatagramSocket	X	-	-	-
java.net.HttpURLConnection	X	-	-	-
java.net.URL	X	-	-	-
java.net.URLConnection	X	-	-	-
java.text.Format	X	-	-	-
java.util.Timer	X	-	-	-
java.util.zip.ZipFile	X	X	X	-
java.util.zip.ZipInputStream	X	-	-	-
java.util.zip.ZipOutputStream	X	-	-	-
org.nocrala.tools.gis.data.esri.shapefile.ShapeFileReader	X	-	-	-

Legenda: Onde está um "X" significa que a ferramenta pode identificar vazamentos de recursos da classe de recurso e o "-" representa que não pode.

7.2 Resultados

Os resultados deste estudo serão mostrados para cada uma das aplicações móveis, começando com a aplicação *OI Notepad* e, depois, para as aplicações *OI Safe*, *AnyMemo*, *Avare* e *Seafile*. Na Tabela 7.4 é mostrado um resumo das aplicações, além disso, mais

Tabela 7.4 – Resumo do resultado das aplicações.

	OI Notepad		OI Safe	
Ferramenta	Vazamentos Encontrados	Vazamentos Possíveis	Vazamentos Encontrados	Vazamentos Possíveis
LeakPred	17	20	8	13
Infer	10	19	7	10
Lint	10	13	1	1
Find Bugs	0	2	0	6
	AnyMemo		Avare	
Ferramenta	Vazamentos Encontrados	Vazamentos Possíveis	Vazamentos Encontrados	Vazamentos Possíveis
LeakPred	11	11	20	22
Infer	2	4	0	11
Lint	0	0	0	0
Find Bugs	0	2	2	18
	Seafile			
Ferramenta	Vazamentos Encontrados	Vazamentos Possíveis		
LeakPred	35	41		
Infer	12	32		
Lint	2	21		
Find Bugs	0	8		

Tabela 7.5 – Componentes com vazamentos de recursos na aplicação notepad.

Pacote	**Classe**	**Componente**	**Parâmetros**	**Retorno**
org.openintents.notepad.theme	ThemeUtils	addThemeInfos	(PackageManager,String, ApplicationInfo,List)	void
org.openintents.notepad.search	SearchQueryResultsActivity	doSearchQuery	(Intent,String)	void
org.openintents.notepad.search	SearchSuggestionProvider	getSuggestions	(String,String[])	Cursor
org.openintents.notepad.search	SearchSuggestionProvider	refreshShortcut	(String,String[])	Cursor
org.openintents.notepad.search	FullTextSearch	getCursor	(Context,String)	Cursor
org.openintents.notepad.noteslist	NotesList	updateTagList	()	void
org.openintents.notepad.noteslist	NotesList	sendNoteByEmail	(long)	void
org.openintents.notepad.noteslist	NotesList	encryptNote	(long,String)	void
org.openintents.notepad.noteslist	NotesList	saveFile	(Uri,File)	void
org.openintents.notepad.noteslist	NotesList	writeToFile	(File,String)	void
org.openintents.util	ProviderUtils	getAffectedRows	(SQLiteDatabase,String, String,String[])	long[]
org.openintents.notepad.activity	SaveFileActivity	writeToFile	(Context,File,String)	void
org.openintents.notepad.activity	SaveFileActivity	getFilenameFromNoteTitle	(Uri)	Uri
org.openintents.notepad	NoteEditor	onPause	()	void
org.openintents.notepad	NoteEditor	deleteNote	()	void
org.openintents.notepad	NoteEditor	importNote	()	void
org.openintents.notepad	NotePadProvider	query	(Uri,String[],String, String[],String)	Cursor
org.openintents.notepad	NotePadProvider	insert	(Uri,ContentValues)	Uri
org.openintents.notepad	NotePadProvider	delete	(Uri,String, String[])	int
org.openintents.notepad	NotePadProvider	update	(Uri,ContentValues, String,String[])	int

informações (por exemplo, classe de recurso ou qual ferramenta identificou cada vazamento) sobre os componentes com vazamentos de recursos identificados em cada uma das aplicações estão disponíveis em <https://bit.ly/3P0LGEk>.

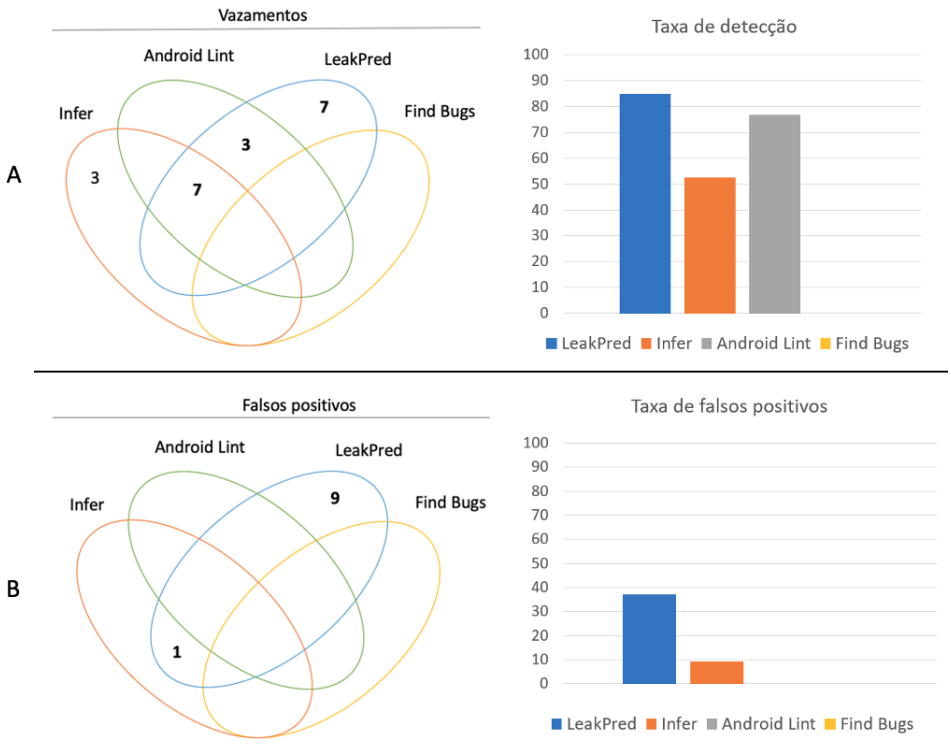


Figura 7.4 – Componentes com vazamentos, taxa de detecção e de falsos positivos na aplicação OI Notepad.

7.2.1 Resultados Aplicação OI Notepad

Na Tabela 7.5 são apresentados os 20 vazamentos de recursos identificados pelas ferramentas na aplicação *OI Notepad*. Para uma melhor compreensão de quantos vazamentos de recursos foram identificados por mais de uma ferramenta, apresenta-se na Figura 7.4 um diagrama de Venn mostrando os vazamentos de recursos e os falsos positivos, no qual pode-se observar que a *LeakPred* identificou 17 vazamentos enquanto a *FindBugs* não conseguiu identificar nenhum vazamento. As ferramentas *Infer* e *Android Lint* identificaram 10 vazamentos cada. Ainda na Figura 7.4 são apresentadas (à direita) a taxa de detecção de vazamentos de recursos e de falsos positivos, onde é possível observar que a abordagem *LeakPred* obteve a melhor cobertura de detecção com 85%, como também a maior porcentagem de falsos positivos com 37,04%. A ferramenta *FindBugs* não encontrou nenhum dos dois vazamentos que poderia identificar e também não teve nenhum falso positivo.

Os três vazamentos que a abordagem *LeakPred* não conseguiu identificar foram dois da classe *java.io.BufferedWriter* e um da classe *android.database.Cursor*. Em

relação aos falsos positivos reportados pela abordagem *LeakPred*, nove são da classe *android.database.Cursor* e um da classe *java.io.InputStream*. Esses resultados indicam que, com melhorias, a abordagem *LeakPred* poderia potencialmente reduzir a quantidade de falsos positivos detectados.

7.2.2 Resultados Aplicação OI Safe

Tabela 7.6 – Componentes com vazamentos na aplicação safe.

Pacote	Classe	Componente	Parâmetros	Retorno
org.openintents.util	SecureDelete	delete	(File)	boolean
org.openintents.safe.service	AutoLockService	onDestroy	()	void
org.openintents.safe	CSVWriter	read	(Clob)	String
org.openintents.safe	CSVWriter	close	()	void
org.openintents.safe	CryptoHelper	encryptFileWithSessionKey	(ContentResolver,Uri)	Uri
org.openintents.safe	CryptoHelper	decryptFileWithSessionKey	(Context,Uri)	Uri
org.openintents.safe	DBHelper	DBHelper	(Context)	void
org.openintents.safe	AskPassword	keypadOnDestroy	()	void
org.openintents.safe	Export	exportDatabaseToWriter	(Context,Writer)	void
org.openintents.safe	CategoryList	backupToFile	(String)	void
org.openintents.safe	PRNGFixes	generateSeed	()	byte[]
org.openintents.safe	Backup	write	(String,OutputStream)	boolean
org.openintents.safe	Restore	restoreFromFile	(String)	void

A Tabela 7.6 apresenta os 13 vazamentos de recursos descobertos pelas ferramentas na aplicação *OI Safe*, sendo que a Figura 7.5 mostra a quantidade de vazamentos de recursos e de falsos positivos que cada ferramenta identificou e quantos foram identificados por mais de uma ferramenta. Por exemplo, a *LeakPred* identificou 8 vazamentos e a *Infer*, 7. Também na Figura 7.5 é mostrada a taxa de detecção de vazamentos de recursos e de falsos positivos, onde a ferramenta *Android Lint* obteve 100% de cobertura, identificando o único componente com vazamento que ela poderia identificar nessa aplicação. Em seguida, está a ferramenta *Infer* com 70% de cobertura, e depois está a abordagem *LeakPred* com 61,54% de cobertura, identificando 8 dos 13 vazamentos considerados.

Vale ressaltar que a abordagem *LeakPred* não identifica vazamentos de recursos em uma classe intermediária que herda a classe de recurso original, e essa aplicação tinha 2 vazamentos de recursos que estavam nessa situação. Os vazamentos de recursos eram da classe *InputStreamData* que herdava da classe *java.io.InputStream*, impossibilitando a identificação desse vazamento de recurso pela abordagem *LeakPred*. Esses 2 vazamentos

de recursos foram contados nos 13 possíveis de serem identificados.

No que se refere aos falsos positivos, a abordagem *LeakPred* teve a maior porcentagem, com 33,33%, seguida pela ferramenta *Infer* com 30%. As ferramentas *Android Lint* e *FindBugs* não reportaram nenhum falso positivo. No entanto, a ferramenta *FindBugs* não encontrou nenhum dos 6 vazamentos de recursos possíveis de serem identificados nessa aplicação.

7.2.3 Resultados Aplicação AnyMemo

Os 12 componentes com vazamentos detectados pelas ferramentas na aplicação *AnyMemo* são apresentados na Tabela 7.7. Para um melhor entendimento de quantos vazamentos foram identificados por cada ferramenta, foi elaborada a Figura 7.6. Nesta figura observa-se que a *Infer* identificou 2 vazamentos e teve um falso positivo. Ainda na Figura 7.6, é mostrada a taxa de detecção de vazamentos e de falsos positivos, na qual se pode destacar que a abordagem *LeakPred* teve a melhor cobertura, identificando

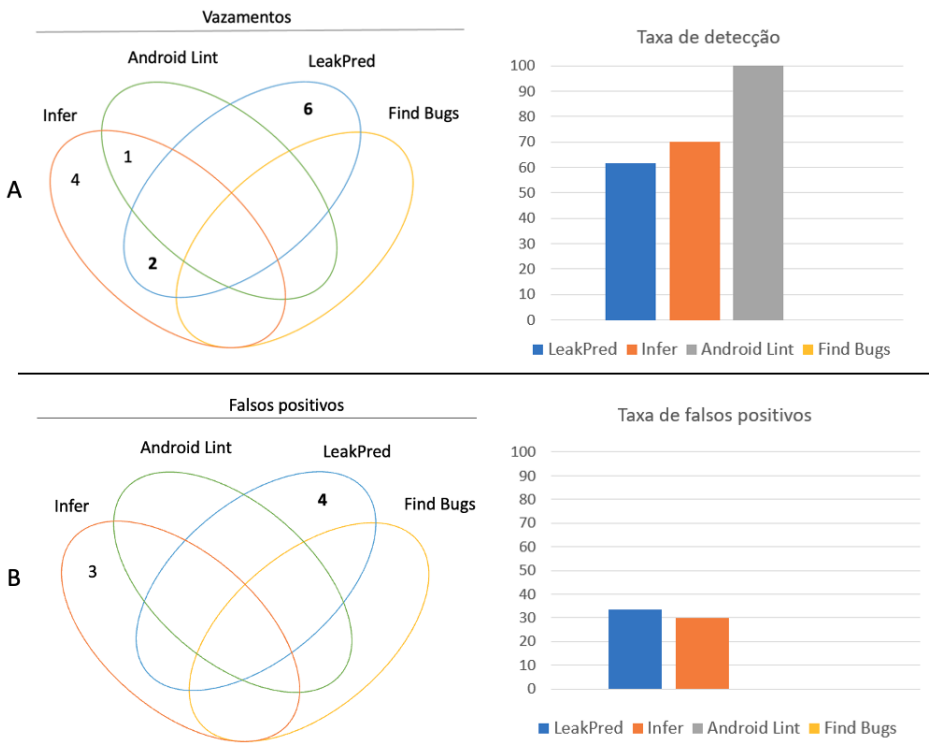


Figura 7.5 – Componentes com vazamentos, taxa de detecção e de falsos positivos na aplicação OI Safe.

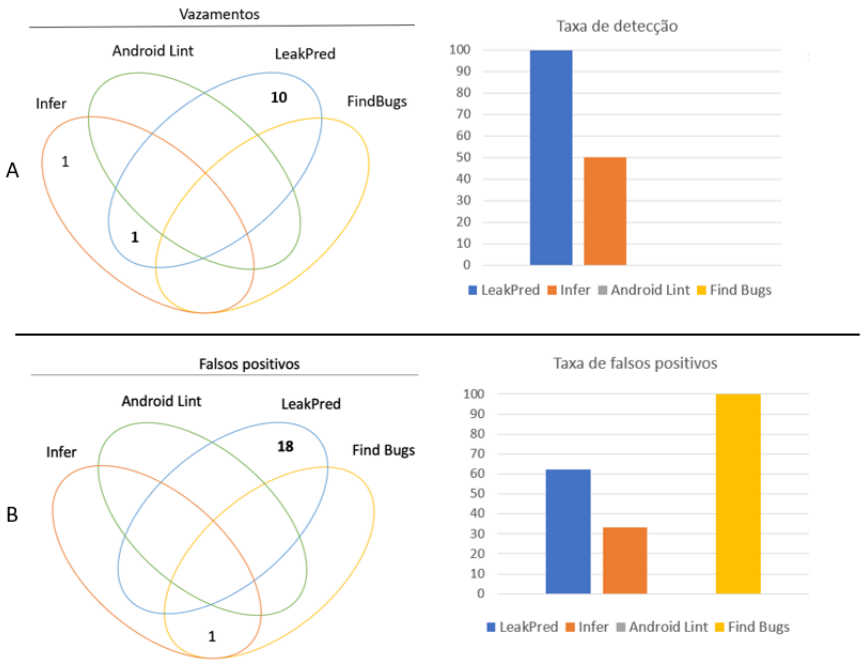


Figura 7.6 – Componentes, taxa de detecção e de falsos positivos com vazamentos na aplicação AnyMemo.

100% dos vazamentos que a abordagem poderia identificar. No que se refere aos falsos positivos, 7 dos 18 eram em arquivos de teste. Portanto, uma melhoria na abordagem seria ignorar os arquivos de teste durante a análise do código.

Tabela 7.7 – Componentes com vazamentos de recursos na aplicação anymemo.

Pacote	Classe	Componente	Parâmetros	Retorno
org.liberty.android .fantastischmemo.converter	Supermemo2008XMLImporter	convert	(String,String)	void
org.liberty.android .fantastischmemo.converter	MnemosyneXMLImporter	convert	(String,String)	void
org.liberty.android .fantastischmemo.converter	CSVExporter	convert	(String,String)	void
org.liberty.android .fantastischmemo.converter	SupermemoXMLImporter	convert	(String,String)	void
org.liberty.android .fantastischmemo.converter	Mnemosyne2CardsImporter	xmlToCards	(File)	List<Card>
org.liberty.android .fantastischmemo.converter	QATxtImporter	convert	(String,String)	void
org.liberty.android .fantastischmemo.converter	Mnemosyne2CardsExporter	createMetadata	(String,File)	void
org.liberty.android .fantastischmemo.tts	AnyMemoTTSImpl	stop	()	void
org.liberty.android .fantastischmemo.receiver	SetAlarmReceiver	cancelNotificationAlarm	(Context)	void
org.liberty.android .fantastischmemo.receiver	SetAlarmReceiver	cancelWidgetUpdateAlarm	(Context)	void
org.liberty.android .fantastischmemo.utils	AMZipUtils	zipDirectory	(File,String, ZipOutputStream)	void
org.liberty.android .fantastischmemo.ui	CardImageGetter	getDrawable	(String)	Drawable

Tabela 7.8 – Componentes com vazamentos de recursos na aplicação avare.

Pacote	Classe	Componente	Parâmetros	Retorno
com.ds.avare .externalFlightPlan	SkvPlanParser	parse	(String, FileInputStream)	ExternalFlightPlan
com.ds.avare.utils	ZipFolder	zipFiles	(String,OutputStream)	boolean
com.ds.avare.utils	ZipFolder	unzipFiles	(String,InputStream)	boolean
com.ds.avare.utils	NetworkHelper	getVersionNetwork	(String)	String
com.ds.avare.utils	Helper	readFromFile	(String)	String
com.ds.avare.utils	Helper	writeFile	(String,String)	boolean
com.ds.avare.shapes	Layer	parse	(String,String)	void
com.ds.avare.shapes	ShapeFileShape	readFile	(String)	ArrayList<ShapeFileShape>
com.ds.avare	LocationActivity	onDestroy	()	void
com.ds.avare	ChecklistActivity	onPause	()	void
com.ds.avare	PlanActivity	onPause	()	void
com.ds.avare	WnbActivity	onPause	()	void
com.ds.avare	ToolsFragment$ImportTask	doInBackground	()	String
com.ds.avare	ToolsFragment$ExportTask	doInBackground	()	String
com.ds.avare.adapters	ChartAdapter$ViewTask	doInBackground	()	Boolean
com.ds.avare.network	Download$DownloadTask	copyInputStream	(InputStream, OutputStream)	void
com.ds.avare.network	Download$DownloadTask	run	()	void
com.ds.avare.adsb	AudibleTrafficAlerts	stopAudibleTrafficAlerts	()	void
com.ds.avare.gps	Gps	stop	()	void
com.ds.avare .instruments	FuelTimer	stop	()	void
com.ds.avare .instruments	UpTimer	stop	()	void
com.ds.avare	StorageService	destroy	()	void

A ferramenta *Infer* teve a segunda melhor cobertura com 50%. Já a ferramenta *Android Lint* não podia identificar nenhum vazamento nessa aplicação e a ferramenta *FindBugs* era capaz de identificar dois, mas não identificou nenhum deles e teve um falso positivo.

7.2.4 Resultados Aplicação Avare

Na Tabela 7.8 são mostrados os 22 vazamentos de recursos encontrados pelas ferramentas na aplicação *Avare*. Na Figura 7.7 é mostrada a quantidade de vazamentos identificados por cada ferramenta. A abordagem *LeakPred* identificou 20 vazamentos e teve 39 falsos positivos. Também na Figura 7.7, é apresentada a taxa de detecção de vazamentos e de falsos positivos, onde observa-se que a abordagem *LeakPred* alcançou a melhor cobertura com 90,91%, bem como a maior porcentagem de falsos positivos, com 62,10%. Em seguida está a ferramenta *FindBugs* com uma cobertura de 11,11% e uma taxa de falsos positivos de 50%.

A ferramenta *Infer* poderia identificar 11 vazamentos, mas não identificou nenhum, e a ferramenta *Android Lint* não poderia identificar nenhum vazamento nessa

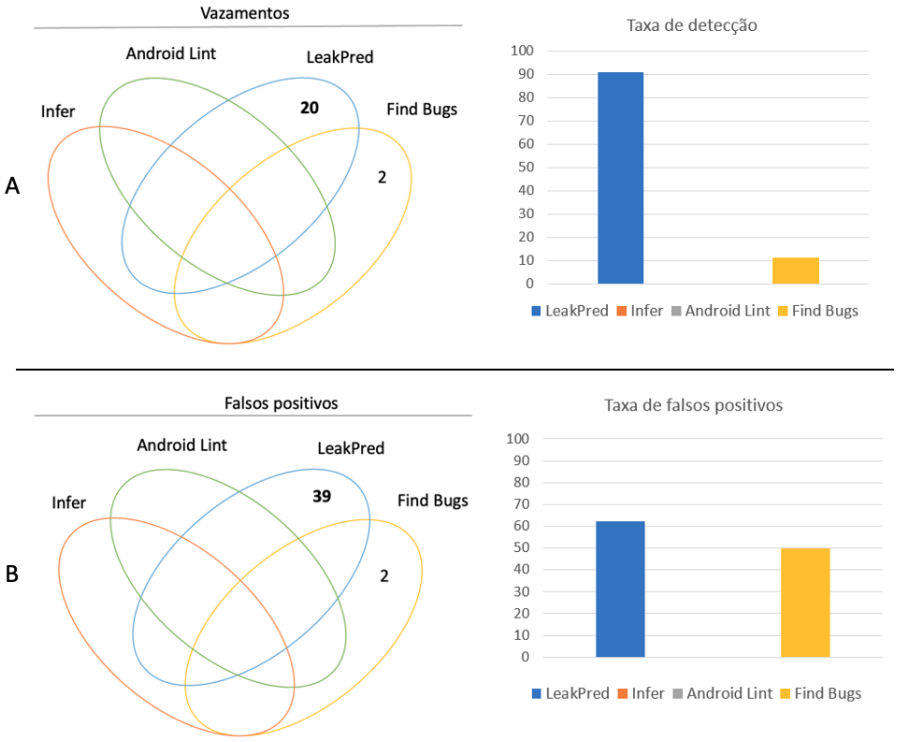


Figura 7.7 – Componentes com vazamentos, taxa de detecção e de falsos positivos na aplicação Avare.

aplicação. Em relação aos falsos positivos da abordagem *LeakPred*, pode-se destacar que 14 dos 39 eram de recursos em variáveis de classe e não de componente, ou seja, geralmente esses recursos são adquiridos em um componente e liberados em um outro componente, o que dificulta a identificação do vazamento e pode levar aos falsos positivos.

7.2.5 Resultados Aplicação Seafile

A Tabela 7.9 exibe os 42 componentes com vazamentos que as ferramentas identificaram na aplicação *Seafile*. Para se distinguir quantos componentes foram identificados por mais de uma ferramenta, foi elaborada a Figura 7.8. Nela, observa-se que teve um mesmo vazamento identificado pelas ferramentas *LeakPred*, *Infer* e *Android Lint*. A abordagem *LeakPred* obteve a melhor cobertura com 85,37% e a maior taxa de falsos positivos com 40,68%, seguida da *Infer*, com 37,50% de cobertura e 25% de falsos positivos.

Tabela 7.9 – Componentes com vazamentos de recursos na aplicação seafile.

Pacote	Classe	Componente	Parâmetros	Retorno
com.seafile.seadroid2	SeafConnection	realLogin	(String,String, boolean)	boolean
com.seafile.seadroid2.gallery	BitmapManager	cancelThreadDecoding	(Thread, ContentResolver)	void
com.seafile.seadroid2.gallery	MultipleImage SelectionActivity	onPause	()	void
com.seafile.seadroid2.gallery	ImageManager	isMediaScannerScanning	(ContentResolver)	boolean
com.seafile.seadroid2.gallery	ImageBlock Manager$ImageBlock	recycle	()	void
com.seafile.seadroid2.notification	BaseNotification Provider	notifyCompleted	(int,String,String)	void
com.seafile.seadroid2.notification	BaseNotification Provider	notifyCompletedWithErrors	(int,String, String,int)	void
com.seafile.seadroid2.notification	BaseNotification Provider	cancelNotification	()	void
com.seafile.seadroid2.data	DatabaseHelper	getFileCacheItem	(String,String, DataManager)	SeafCachedFile
com.seafile.seadroid2.data	DatabaseHelper	getFileCacheItems	(DataManager)	List<SeafCachedFile>
com.seafile.seadroid2.data	DatabaseHelper	getRepoDir	(Account,String)	String
com.seafile.seadroid2.data	DatabaseHelper	getCachedStarredFiles	(Account)	String
com.seafile.seadroid2.data	DatabaseHelper	repoDirExists	(Account,String)	boolean
com.seafile.seadroid2.data	DatabaseHelper	getCachedDirents	(String,String)	String
com.seafile.seadroid2.data	DatabaseHelper	getCachedDirentUsage	(String)	int
com.seafile.seadroid2.data	DatabaseHelper	getEnckey	(String)	Pair<String,String>
com.seafile.seadroid2.ui.activity	SeafilePath ChooserActivity	onDestroy	()	void
com.seafile.seadroid2.ui.activity	ShareToSeafile Activity	getSharedFilePath	(Uri)	String
com.seafile.seadroid2.ui.activity	BrowserActivity $SAFLoadRemoteFileTask	doInBackground	()	File[]
com.seafile.seadroid2.util	Utils	copyFile	(File,File)	void
com.seafile.seadroid2.util	Utils	getFilenamefromUri	(Context,Uri)	String
com.seafile.seadroid2.util	Utils	getPath	(Context,Uri)	String
com.seafile.seadroid2.util	SeafileLog	writeLogtoFile	(String,String, String)	void
com.seafile.seadroid2.provider	SeafileProvider $Runnable	run	()	void
com.seafile.seadroid2.ui.dialog	SslConfirmDialog	onCreateDialog	(Bundle)	Dialog
com.seafile.seadroid2.ssl	CertsDBHelper	getCertificate	(String)	X509Certificate
com.seafile.seadroid2.avatar	AvatarDBHelper	hasAvatar	(Account)	boolean
com.seafile.seadroid2.avatar	AvatarDBHelper	getAvatarList	()	List<Avatar>
com.seafile.seadroid2.monitor	SeafileObserver	startWatching	()	void
com.seafile.seadroid2.monitor	FileMonitor Service	onDestroy	()	void
com.seafile.seadroid2.monitor	Monitor DBHelper	getAutoUploadInfos	()	List<AutoUpdateInfo>
com.seafile.seadroid2.account	Account DBHelper	getAccountList	(SQLiteDatabase)	List<Account>
com.seafile.seadroid2.account	Account DBHelper	getServerInfo	(SQLiteDatabase, String)	ServerInfo
com.seafile.seadroid2.cameraupload	CameraSync Adapter	onPerformSync	(Account,Bundle, String, ContentProviderClient,SyncResult)	void
com.seafile.seadroid2.cameraupload	GalleryBucket Utils	getVideoBucketsBelowApi29	(Context)	List<Bucket>
com.seafile.seadroid2.cameraupload	GalleryBucket Utils	getImageBuckets	(Context)	List<Bucket>
com.seafile.seadroid2.cameraupload	CameraUpload DBHelper	isUploaded	(File)	boolean
com.seafile.seadroid2.gesturelock	LockPassword Utils	checkPassword	(String)	boolean
com.seafile.seadroid2.gesturelock	LockPattern Utils	saveLockPattern	(List)	void
com.seafile.seadroid2.gesturelock	LockPattern Utils	checkPattern	(List)	boolean
com.seafile.seadroid2.account	Account DBHelper	migrateAccounts	(Context)	void
com.seafile.seadroid2.editor	EditorActivity	readToString	(File)	String

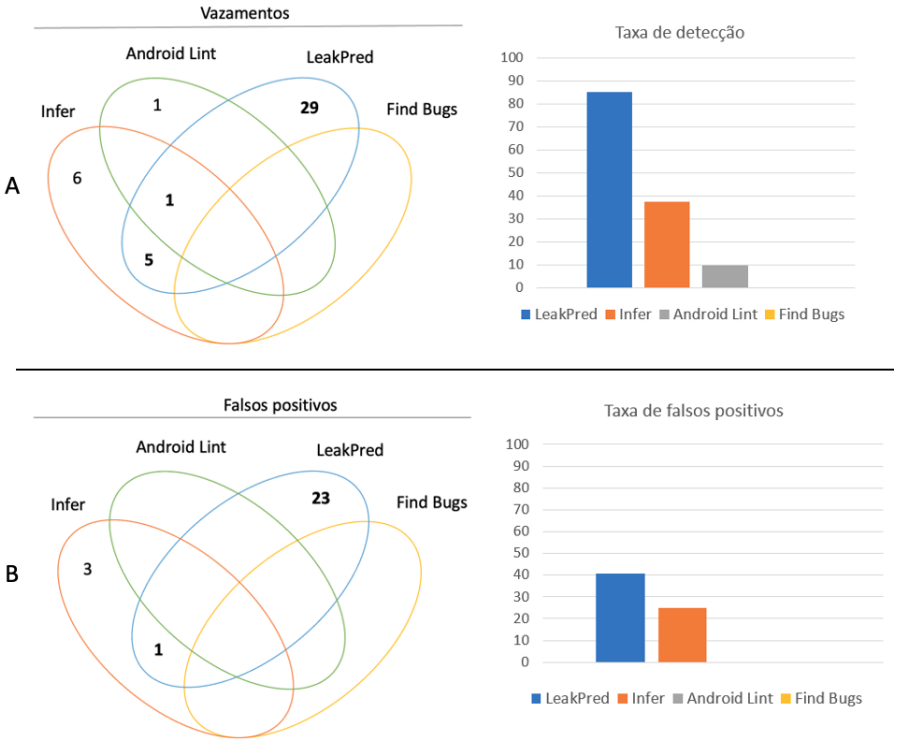


Figura 7.8 – Componentes com vazamentos, taxa de detecção e de falsos positivos na aplicação Seafile.

A ferramenta *Android Lint* não teve nenhum falso positivo, mas somente identificou 2 de 21 vazamentos possíveis (9,52%), e a ferramenta *FindBugs* não tinha nenhum componente com vazamento que poderia ser identificado nessa aplicação.

A abordagem *LeakPred* alcançou a melhor cobertura (média de 84,56% e mediana de 85,37%) de vazamentos identificados e com uma cobertura de 96,15% das classes de vazamentos que poderiam ser identificados nas cinco aplicações. Essa abordagem também ficou com a maior taxa de falsos positivos (média de 47,84% e mediana de 40,68%). Portanto, percebeu-se a necessidade de refinamento para diminuir os falsos positivos.

Nos falsos positivos da abordagem *LeakPred* temos três padrões que podemos destacar: (1) ao ter uma variável nível de classe sendo instanciada em um componente e liberada em outro, uma possível forma de melhoria seria encontrar métricas que conseguissem representar essa situação, (2) foram analisados arquivos de testes, o que ocasionou alguns falsos positivos nesses arquivos, uma solução seria ignorar os arquivos de teste durante a análise do código, e (3) foi observado que durante a manipulação de

arquivos, uma classe de recurso é instanciada e passada como parâmetro para instanciar outra classe de recurso e assim sucessivamente, algumas vezes sendo necessário fechar apenas uma delas para o recurso ser liberado corretamente, um meio de solucionar esse problema seria definir uma heurística para tratar esse fenômeno. Outra forma de diminuir os falsos positivos seria aumentar a quantidade de vazamentos da base de dados.

A ferramenta *Android Lint* obteve a segunda maior cobertura (média de 62,15% e mediana de 76,92%) de componentes com vazamentos de recursos identificados e nenhum falso positivo. No entanto, ela pode identificar apenas a classe de recurso *android.database.Cursor* (1,92%) dentre as classes de vazamentos das cinco aplicações do estudo.

A abordagem *LeakPred* pode identificar vazamentos em várias classes de recursos. Isso se apresenta como uma vantagem, pois o fato de uma ferramenta poder identificar várias categorias de vazamentos diminui a quantidade de ferramentas a serem configuradas e executadas, o que pode ajudar na utilização durante o desenvolvimento de aplicações móveis e consequentemente na redução de custos e/ou tempo ao longo do projeto.

7.3 Ameaças à Validade

Este estudo possui ameaças internas, de construção e externas que devem ser examinadas. Esta seção detalhará cada uma.

Validade Interna: a amostra dos projetos não foi totalmente aleatória, visto serem selecionados aleatoriamente da lista de aplicações de código aberto levantadas em (LIU et al., 2019). Como este é um estudo de viabilidade, acredita-se que essa questão não representa uma ameaça significativa.

Validade de Construção: o estudo de viabilidade utilizou cinco aplicações da plataforma Android de diferentes categorias desenvolvidas na linguagem Java. O estudo pode não ser representativo para outras categorias de aplicações móveis.

Validade Externa: para diminuir essa ameaça foram utilizadas cinco aplicações

de quatro diferentes categorias, a saber, produtividade, mapas e navegação, ferramentas e educação. Assim como, comparou-se a abordagem *LeakPred* com mais de uma ferramenta do estado da arte. Futuramente, pretende-se aumentar o número de aplicações de diferentes categorias.

7.4 Conclusões do Estudo

Os resultados encontrados fornecem uma compreensão básica da viabilidade da abordagem *LeakPred*. Onde observou-se que a abordagem pode auxiliar os desenvolvedores na identificação de componentes com vazamentos de recursos, visto que ela atingiu a melhor mediana (85,37%) de componentes com vazamentos de recursos identificados e teve a maior cobertura (96,15%) de classes de vazamentos de recursos nas aplicações.

7.5 Considerações Finais

Neste capítulo foi apresentado o estudo de viabilidade para a abordagem *LeakPred* visando analisá-la, por meio de um experimento controlado, com respeito à eficácia em identificar componentes com vazamentos de recursos comparando com ferramentas do estado da arte. Os resultados mostraram a possibilidade de usar a abordagem *LeakPred* para identificar vazamentos de recursos em aplicações móveis, pois ela obteve a maior mediana de porcentagem de cobertura de vazamentos de recursos identificados com 85,37%, bem como, teve a maior porcentagem de cobertura de classes de vazamentos, 96,15%.

Com os resultados do estudo também foi percebida uma possibilidade de refinamento da abordagem *LeakPred* para diminuir a quantidade de falsos positivos. No próximo capítulo, serão apresentadas as considerações finais e contribuições deste trabalho, assim como as perspectivas de trabalhos futuros para a continuação desta pesquisa.

8

CONCLUSÕES E TRABALHOS FUTUROS

8.1 Considerações Finais

Os dispositivos móveis contêm alguns recursos, por exemplo, câmera, bateria e memória, que são adquiridos, usados e depois liberados por aplicações móveis. Sempre que um recurso é adquirido e não é liberado corretamente, ocorre um defeito chamado **vazamento de recurso**, que pode causar enorme consumo de bateria, degradação da usabilidade e responsividade, enorme consumo de memória, degradação de desempenho, falha da aplicação e até mesmo problema em outra aplicação devido aos recursos exclusivos.

Este trabalho apresentou um mapeamento sistemático (Seção **??**) realizado com o intuito de identificar categorias de causas de vazamentos de recurso. Foram identificadas 10 categorias por meio do mapeamento, as quais foram avaliadas em relação à relevância mediante uma pesquisa de opinião (Seção **??**) com desenvolvedores de aplicações móveis. Como resultado deste estudo, foi obtido um conjunto base de 6 categorias mais relevantes, bem como foi acrescentada uma nova categoria de causas de vazamentos de recurso, totalizando 11 categorias, as quais são:

1. *Vazamento em caminhos excepcionais*;
2. *Vazamento causado pela complexidade do ciclo de vida da aplicação*;

3. *Vazamento causado por liberação inadequada de recursos em loop*;
4. *Vazamento em caminhos irregulares*,
5. *Vazamento causado por alocação e desalocação imprópria*;
6. *Vazamento causado pela condição de corrida*;
7. *Vazamento causado pelo Android SDK*;
8. *Vazamento completo*;
9. *Vazamento em caminhos normais*;
10. *Vazamento causado por recurso não utilizado*; e
11. *Vazamento de memória devido a APIs de terceiros mal construídas*.

Com isso, pode-de destacar que a presente proposta de tese teve como objetivo construir uma abordagem automatizada (Capítulo 5) para a identificação de componentes que possuem vazamentos de recursos em aplicações móveis, utilizando aprendizado de máquina baseado em métricas de software. O conjunto de seis métricas escolhidas para caracterizar os componentes são: *returnTypeIsAResourceClass*, *callsCloseComponent*, *closeResource*, *overrideOnPause*, *overrideOnStop* e *overrideOnDestroy*. E as seis técnicas de aprendizado de máquina escolhidas para se analisar a eficácia na previsão de componente com vazamentos são: máquina de vetor de suporte (SVM), naive bayes, rede neural profunda (DNN), regressão logística, floresta aleatória e k-vizinhos mais próximos (KNN). Dessa maneira, podemos destacar que a abordagem tem a finalidade de dar suporte aos desenvolvedores na identificação de componentes com vazamentos de recursos em suas aplicações.

Foi realizado um estudo (Capítulo 6) visando analisar a eficácia de classificadores (SVM, Naive Bayes, DNN, Regressão Logística, Floresta Aleatória e KNN) na predição de componentes com vazamentos de recursos. O resultado deste estudo empírico possibilitou a escolha do classificador *DNN* para ser utilizado na abordagem proposta, pois obteve o melhor resultado na métrica *ROC AUC* (78,93%), a qual foi escolhida por ser recomendada para quando a base de dados é desbalanceada.

Os resultados obtidos com o estudo de viabilidade, apresentado no Capítulo 7, permitiram aceitar a hipótese definida para esta pesquisa. A abordagem *LeakPred* atingiu a melhor mediana (85,37%) de componentes com vazamentos de recursos identificados. No entanto, é importante reconhecer que a *LeakPred* também apresentou uma maior taxa de falsos positivos. Apesar disso, a abordagem se destaca como a melhor opção, pois sua capacidade de identificar vazamentos de recursos supera as limitações impostas pelos falsos positivos, oferecendo uma ferramenta robusta para o desenvolvimento de aplicações móveis Android.

8.2 Contribuições

As principais contribuições desta pesquisa oferecidas à comunidade acadêmica e à indústria são:

- Identificação e análise de relevância de 11 categorias (*Vazamento em caminhos excepcionais*, *Vazamento causado pela complexidade do ciclo de vida da aplicação*, *Vazamento causado por liberação inadequada de recursos em loop*, *Vazamento em caminhos irregulares*, *Vazamento causado por alocação e desalocação imprópria*, *Vazamento causado pela condição de corrida*, *Vazamento causado pelo Android SDK*, *Vazamento completo*, *Vazamento em caminhos normais*, *Vazamento causado por recurso não utilizado*, e *Vazamento de memória devido a APIs de terceiros mal construídas*) de causas de vazamento de recursos em aplicações móveis;

- Identificação de 6 métricas (*returnTypeIsAResourceClass*, *calledCloseComponent*, *closeResource*, *overrideOnPause*, *overrideOnStop* e *overrideOnDestroy*) relacionadas ao problema pesquisado;

- Definição de uma abordagem para a identificação de vazamentos de recursos em aplicações móveis, envolvendo aprendizado de máquina e métricas de software;

- Avaliação da eficácia de 6 classificadores (*SVM*, *Naive Bayes*, *DNN*, *Regressão Logística*, *Floresta Aleatória* e *KNN*) na identificação de componentes com vazamentos de recursos em aplicações móveis, onde o classificador *KNN* teve uma precisão de

88,04%, seguido pela *DNN* com 87,99%. O Classificador *DNN* foi escolhido para ser usado na proposta por ter o melhor resultado na métrica ROC AUC (78,93%) e o segundo melhor resultado em acurácia (87,75%), revocação (87,75%) e f-measure (86,17%);

- A criação da base de dados *CompLeaks* com 54 aplicações contendo 780 componentes com vazamentos de recursos, disponível em <https://bit.ly/3uJEu80>;

- Análise da eficácia da abordagem *LeakPred* em relação a 5 ferramentas do estado da arte na identificação de componentes com vazamento em aplicações móveis Android, na qual a abordagem *LeakPred* teve a melhor cobertura com a mediana de 85,37%, seguida pela ferramenta *Android Lint* com 79,92%;

- Artigos publicados:

  Previsão de Vazamento de Recursos em Aplicações Android usando Aprendizado de Máquina - SBC 2020 (LIMA; GIUSTI; DIAS-NETO, 2020);

  Resource Leak Prediction in Android Applications Using Machine Learning - BJD 2021 (LIMA; GIUSTI; DIAS-NETO, 2021);

  LeakPred: An Approach for Identifying Components with Resource Leaks in Android Mobile Applications - Computers 2024 (LIMA; GIUSTI; DIAS-NETO, 2024).

8.3 Limitações e Decisões do Projeto de Implementação

Durante o desenvolvimento da pesquisa, algumas limitações foram observadas e certas decisões do projeto de implementação foram tomadas para otimizar a abordagem *LeakPred*:

- A implementação atual da *LeakPred* não verifica automaticamente as superclasses quando analisa uma classe que pode herdar de uma classe de recurso original. Por exemplo, a classe *InputStreamData*, que herda da classe *java.io.InputStream*, pode

não ser considerada na análise. Essa limitação pode ser superada com a expansão da análise para incluir também as superclasses das classes analisadas.

- A análise está atualmente restrita às classes de recursos que existem na base de dados. Poderia ser incluída uma opção de ler um arquivo de texto com classes de recursos que não estão na base de dados, o que permitiria uma análise mais abrangente. No entanto, expandir a análise para incluir outras classes de recursos poderia aumentar a taxa de falsos positivos, o que influenciou a decisão de manter um escopo mais restrito.
- A versão da abordagem implementada inclui a análise de arquivos de teste, o que é recomendável ser ignorado, pois pode introduzir falsos positivos na análise da aplicação.

8.4 Trabalhos futuros

Para os trabalhos futuros desta pesquisa, diversos aprimoramentos podem ser considerados, poderia ser realizado um estudo aumentando a quantidade de aplicações móveis de código aberto, bem como utilizando aplicações de empresas. Outra possibilidade seria avaliar a eficiência das ferramentas, medindo o tempo de execução necessário para realizar a análise de vazamentos de recursos. O problema foi atualmente modelado como uma classificação binária, poderia ser expandido para uma classificação multi-classe, o que permitiria uma categorização mais detalhada dos tipos de vazamentos de recursos.

Além disso, seria interessante separar a classificação dos recursos que não são liberados ao longo do ciclo de vida da aplicação daqueles que são. Outra linha de investigação seria a adaptação do método para outras linguagens de programação, ampliando seu escopo de aplicação. Adicionalmente, a classificação poderia levar em conta grupos de recursos, oferecendo uma análise mais complexa. O uso de inteligência artificial generativa para corrigir automaticamente os vazamentos identificados também apresenta um potencial significativo. Ademais, uma evolução técnica importante seria a

extração do código-fonte diretamente do APK da aplicação, eliminando a dependência do código-fonte original.

Outra pesquisa futura interessante seria remover a limitação da abordagem *LeakPred* na identificação de vazamentos em uma classe intermediária que herda a classe de vazamento original e a remoção da análise dos arquivos de teste. Por fim, existe a possibilidade de adaptar a abordagem e realizar um estudo de viabilidade para a plataforma iOS.

REFERÊNCIAS

AHN, S. Automation of memory leak detection and correction on android jni (poster). In: *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*. [S.l.: s.n.], 2019. p. 533–534. 47, 49, 51

ALAM, F. et al. Energy optimization in android applications through wakelock placement. In: IEEE. *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. [S.l.], 2014. p. 1–4. 47, 49, 51

AMALFITANO, D. et al. Do memories haunt you? an automated black box testing approach for detecting memory leaks in android apps. *IEEE Access*, IEEE, v. 8, p. 12217–12231, 2020. 18, 47, 49, 51

ANDROID. *Android Lint*. 2024. <https://developer.android.com/studio/write/lint> (Accessed 08 July 2024). Disponível em: <https://developer.android.com/studio/write/lint>. 71, 72, 103

ARAUJO, J. et al. An investigative approach to software aging in android applications. In: IEEE. *2013 IEEE international conference on systems, man, and cybernetics*. [S.l.], 2013. p. 1229–1234. 47, 49, 51

BANERJEE, A. et al. Energypatch: Repairing resource leaks to improve energy-efficiency of android apps. *IEEE Transactions on Software Engineering*, IEEE, v. 44, n. 5, p. 470–490, 2017. 40, 47, 49, 51

BANERJEE, A. et al. Detecting energy bugs and hotspots in mobile apps. In: *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. [S.l.: s.n.], 2014. p. 588–598. 18

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. The experience factory. *Encyclopedia of Software Eng.: Vol*, v. 1, p. 469–476, 1994. 74

BHATT, B. N.; FURIA, C. A. Automated repair of resource leaks in android applications. *arXiv preprint arXiv:2003.03201*, 2020. 28, 70, 72

BHATT, B. N.; FURIA, C. A. Automated repair of resource leaks in android applications. *Journal of Systems and Software*, Elsevier, v. 192, p. 111417, 2022. 18, 47, 49, 51

BIAU, G.; SCORNET, E. A random forest guided tour. *Test*, Springer, v. 25, p. 197–227, 2016. 33

CATAL, C.; DIRI, B. Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. *Information Sciences*, Elsevier, v. 179, n. 8, p. 1040–1058, 2009. 34

CHANG, B.-Y. E. Refuting heap reachability. In: SPRINGER. *International Conference on Verification, Model Checking, and Abstract Interpretation*. [S.l.], 2014. p. 137–141. 47, 49, 51

CHAWLA, N. V. et al. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, v. 16, p. 321–357, 2002. 90

CHEN, B.; LI, X.; ZHOU, X. Powersensor: A method for power optimization of smartphone through sensing wakelock application. In: IEEE. *2017 9th International Conference on Wireless Communications and Signal Processing (WCSP)*. [S.l.], 2017. p. 1–6. 47, 49, 51

CICHY, R. M.; KAISER, D. Deep neural networks as scientific models. *Trends in cognitive sciences*, Elsevier, v. 23, n. 4, p. 305–317, 2019. 32

CLARK, R. D.; WEBSTER-CLARK, D. J. Managing bias in roc curves. *Journal of computer-aided molecular design*, Springer, v. 22, n. 3, p. 141–146, 2008. 36

CORTES, C.; VAPNIK, V. Support-vector networks. *Machine learning*, Springer, v. 20, n. 3, p. 273–297, 1995. 32

CUI, B. et al. Detection of java basic thread misuses based on static event analysis. In: IEEE. *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2023. p. 1049–1060. 51

DIAS-NETO, A. C.; TRAVASSOS, G. H. Surveying model based testing approaches characterization attributes. In: *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. [S.l.: s.n.], 2008. p. 324–326. 75

DIJKHUIS, T. B. et al. Personalized physical activity coaching: a machine learning approach. *Sensors*, Multidisciplinary Digital Publishing Institute, v. 18, n. 2, p. 623, 2018. 90

EFENDIOGLU, M.; SEN, A.; KOROGLU, Y. Bug prediction of systemc models using machine learning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, IEEE, v. 38, n. 3, p. 419–429, 2019. 31, 36, 86, 90

FACEBOOK. *Infer*. 2024. <https://fbinfer.com/> (Accessed 08 July 2024). Disponível em: <https://fbinfer.com/>. 71, 72, 103

FERRARI, A. et al. Detecting energy leaks in android app with poem. In: IEEE. *2015 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*. [S.l.], 2015. p. 421–426. 47, 49, 51

FINDLAY, M. R. et al. Machine learning provides predictive analysis into silver nanoparticle protein corona formation from physicochemical properties. *Environmental Science: Nano*, Royal Society of Chemistry, v. 5, n. 1, p. 64–71, 2018. 90

FRANK, E. et al. Naive bayes for regression. *Machine Learning*, Springer, v. 41, p. 5–25, 2000. 32

FUKUNAGA, K.; NARENDRA, P. M. A branch and bound algorithm for computing k-nearest neighbors. *IEEE transactions on computers*, IEEE, v. 100, n. 7, p. 750–753, 1975. 34

GHANEM, T.; ZEIN, S. A model-based approach to assist android activity lifecycle development. In: IEEE. *2020 4th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*. [S.l.], 2020. p. 1–12. 47, 49, 51

GOOGLE. *Application Fundamentals*. 2024. <https://developer.android.com/guide/components/fundamentals> (Accessed 08 July 2024). Disponível em: <https://developer.android.com/guide/components/fundamentals>. 26, 27

GOOGLE. *Understand the Activity Lifecycle*. 2024. <https://developer.android.com/guide/components/activities/activity-lifecycle> (Accessed 08 July 2024). Disponível em: <https://developer.android.com/guide/components/activities/activity-lifecycle>. 26, 27

GUO, C. et al. Characterizing and detecting resource leaks in android applications. In: IEEE. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2013. p. 389–398. 28, 29, 30, 40, 42, 47, 49, 50, 51, 68, 72

HAGOS, T.; HAGOS, T.; ANGLIN. *Learn Android Studio 3 with Kotlin*. [S.l.]: Springer, 2018. 26, 27

HALL, S.; NATARAJ, S.; KIM, D.-K. Detecting no-sleep energy bugs using reference counted variables. In: *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. [S.l.: s.n.], 2018. p. 161–165. 47, 49, 51

HAMBURG, M. Basic statistics: A modern approach. *Journal of the Royal Statistical Society*, v. 143, n. 1, 1980. 77

HOSHIEAH, N. et al. A static analysis of android source code for lifecycle development usage patterns. *Journal of Computer Science*, Science Publications, v. 15, n. 1, p. 92–107, 2019. 47, 49, 51, 70, 72

JIANG, H. et al. Detecting energy bugs in android apps using static analysis. In: SPRINGER. *International Conference on Formal Engineering Methods*. [S.l.], 2017. p. 192–208. 20, 40, 47, 49, 51, 69, 72

JUN, M. et al. Leakdaf: An automated tool for detecting leaked activities and fragments of android applications. In: IEEE. *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. [S.l.], 2017. v. 1, p. 23–32. 47, 49, 51

KALINOWSKI, M. Introdução à inspeção de software. *Revista Engenharia de Software: Qualidade de software*, v. 1, 2008. 34

KAMEI, Y.; SHIHAB, E. Defect prediction: Accomplishments and future challenges. In: IEEE. *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*. [S.l.], 2016. v. 5, p. 33–45. 35

KAUR, H.; PANNU, H. S.; MALHI, A. K. A systematic review on imbalanced data challenges in machine learning: Applications and solutions. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 52, n. 4, p. 1–36, 2019. 33

KELLOGG, M. et al. Lightweight and modular resource leak verification. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. [S.l.: s.n.], 2021. p. 181–192. 70, 72, 73, 104

KHAN, M. U. et al. Detecting wake lock leaks in android apps using machine learning. *IEEE Access*, IEEE, v. 9, p. 125753–125767, 2021. 47, 49, 51

KHAN, M. U. et al. Wake lock leak detection in android apps using multi-layer perceptron. *Electronics*, MDPI, v. 10, n. 18, p. 2211, 2021. 47, 49, 51

KIM, K.; CHA, H. Wakescope: runtime wakelock anomaly management scheme for android platform. In: IEEE. *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*. [S.l.], 2013. p. 1–10. 40, 47, 49, 51

KITCHENHAM, B.; CHARTERS, S. Guidelines for performing systematic literature reviews in software engineering. Citeseer, 2007. 39

KRAMER, O.; KRAMER, O. K-nearest neighbors. *Dimensionality reduction with unsupervised nearest neighbors*, Springer, p. 13–23, 2013. 34

LAVALLEY, M. P. Logistic regression. *Circulation*, Am Heart Assoc, v. 117, n. 18, p. 2395–2399, 2008. 33

LE, H. A. Analyzing energy leaks of android applications using event-b. *Mobile Networks and Applications*, Springer, v. 26, n. 3, p. 1329–1338, 2021. 47, 49, 51

LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. *nature*, Nature Publishing Group, v. 521, n. 7553, p. 436–444, 2015. 32

LI, Z. et al. Heterogeneous defect prediction with two-stage ensemble learning. *Automated Software Engineering*, Springer, v. 26, n. 3, p. 599–651, 2019. 31, 36, 86

LIMA, J. G.; GIUSTI, R.; DIAS-NETO, A. C. Previsão de vazamento de recursos em aplicações android usando aprendizado de máquina. In: SBC. *Anais Estendidos do XIX Simpósio Brasileiro de Qualidade de Software*. [S.l.], 2020. p. 31–38. 30, 121

LIMA, J. G.; GIUSTI, R.; DIAS-NETO, A. C. Resource leak prediction in android applications using machine learning. *Brazilian Journal of Development*, v. 7, n. 5, p. 47820–47837, 2021. 121

LIMA, J. G.; GIUSTI, R.; DIAS-NETO, A. C. Leakpred: An approach for identifying components with resource leaks in android mobile applications. *Computers*, MDPI, v. 13, n. 6, p. 140, 2024. 121

LIU, Q. et al. Dynamic detection of asynctask related defects. In: IEEE. *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. [S.l.], 2021. p. 357–366. 19, 47, 49, 51

LIU, Y. et al. Droidleaks: a comprehensive database of resource leaks in android apps. *Empirical Software Engineering*, Springer, v. 24, n. 6, p. 3435–3483, 2019. 19, 20, 23, 28, 42, 47, 49, 50, 51, 84, 101, 102, 105, 116

LIU, Y.; XU, C. Veridroid: Automating android application verification. In: *Proceedings of the 2013 Middleware Doctoral Symposium*. [S.l.: s.n.], 2013. p. 1–6. 40, 47, 49, 51

LU, Y. et al. Detecting resource utilization bugs induced by variant lifecycles in android. In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. [S.l.: s.n.], 2022. p. 642–653. 47, 49, 51

MA, J. et al. Lesdroid: a tool for detecting exported service leaks of android applications. In: *Proceedings of the 26th Conference on Program Comprehension*. [S.l.: s.n.], 2018. p. 244–254. 47, 49, 51, 69, 72

MAFRA, S. N.; BARCELOS, R. F.; TRAVASSOS, G. H. Aplicando uma metodologia baseada em evidência na definição de novas tecnologias de software. In: *Proceedings of the 20th Brazilian Symposium on Software Engineering (SBES 2006)*. [S.l.: s.n.], 2006. v. 1, p. 239–254. 22

MANJULA, C.; FLORENCE, L. Deep neural network based hybrid approach for software defect prediction using software metrics. *Cluster Computing*, Springer, v. 22, n. 4, p. 9847–9863, 2019. 31, 36, 86

MOHRI, M.; ROSTAMIZADEH, A.; TALWALKAR, A. *Foundations of machine learning*. [S.l.]: MIT press, 2018. 1 p. 30

NANAVATI, J. et al. Critical review and fine-tuning performance of flutter applications. In: IEEE. *2024 5th International Conference on Mobile Computing and Sustainable Informatics (ICMCSI)*. [S.l.], 2024. p. 838–841. 51

NASEER, A.; NADEEM, A.; ZAMAN, Q. U. A gui based approach to detect energy bugs in android applications. In: IEEE. *2021 16th International Conference on Emerging Technologies (ICET)*. [S.l.], 2021. p. 1–6. 18, 47, 49, 51

NGUYEN, T.; VU, P.; NGUYEN, T. Code recommendation for exception handling. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. [S.l.: s.n.], 2020. p. 1027–1038. 47, 49, 51

NGUYEN, T. T.; VU, P. M.; NGUYEN, T. T. An empirical study of exception handling bugs and fixes. In: *Proceedings of the 2019 ACM Southeast Conference*. [S.l.: s.n.], 2019. p. 257–260. 47, 49, 51

NOBLE, W. S. What is a support vector machine? *Nature biotechnology*, Nature Publishing Group UK London, v. 24, n. 12, p. 1565–1567, 2006. 32

PALOMBA, F. et al. On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology*, Elsevier, v. 105, p. 43–55, 2019. 47, 49, 51

PATHAK, A. et al. What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps. In: *Proceedings of the 10th international conference on Mobile systems, applications, and services*. [S.l.: s.n.], 2012. p. 267–280. 47, 49, 51

PEREIRA, R. B. et al. Extending ecoandroid with automated detection of resource leaks. 2022. 20, 47, 49, 51, 71, 72, 73, 98, 104

PETERSEN, K. et al. Systematic mapping studies in software engineering. In: *12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12*. [S.l.: s.n.], 2008. p. 1–10. 40, 42

PETERSEN, K.; VAKKALANKA, S.; KUZNIARZ, L. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, Elsevier, v. 64, p. 1–18, 2015. 39, 40, 41

PUGH, B.; LOSKUTOV, A.; LEA, K. *FindBugs*. 2024. <https://findbugs.sourceforge.net/bugDescriptions.html> (Accessed 08 July 2024). Disponível em: <https://findbugs.sourceforge.net/bugDescriptions.html>. 71, 72, 103

QIAN, J.; ZHOU, D. Prioritizing test cases for memory leaks in android applications. *Journal of Computer Science and Technology*, Springer, v. 31, n. 5, p. 869–882, 2016. 20, 30, 47, 49, 51, 68, 72

RAWAT, M. S.; DUBEY, S. K. Software defect prediction models for quality improvement: a literature study. *International Journal of Computer Science Issues (IJCSI)*, Citeseer, v. 9, n. 5, p. 288, 2012. 21, 34, 35

RICKY, M. Y.; PURNOMO, F.; YULIANTO, B. Mobile application software defect prediction. In: IEEE. *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. [S.l.], 2016. p. 307–313. 34

RIGANELLI, O.; MICUCCI, D.; MARIANI, L. Controlling interactions with libraries in android apps through runtime enforcement. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, ACM New York, NY, USA, v. 14, n. 2, p. 1–29, 2019. 19, 47, 49, 51

RIGANELLI, O.; MICUCCI, D.; MARIANI, L. From source code to test cases: A comprehensive benchmark for resource leak detection in android apps. *Software: Practice and Experience*, Wiley Online Library, v. 49, n. 3, p. 540–548, 2019. 47, 49, 51

RUSSELL, S. J.; NORVIG, P. *Artificial intelligence*. [S.l.]: Pearson Education, 2010. 31

SAJU, N. et al. Green mining for android based applications using refactoring approach. In: IEEE. *2021 9th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*. [S.l.], 2021. p. 1–6. 47, 49, 51

SAKHARE, P. B.; KIM, D.-K.; HAMDI, M. Detecting no-sleep bugs using sequential reference counts. In: IEEE. *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*. [S.l.], 2019. v. 1, p. 940–941. 47, 49, 51

SANTHANAKRISHNAN, G.; CARGILE, C.; OLMSTED, A. Memory leak detection in android applications based on code patterns. In: IEEE. *2016 International Conference on Information Society (i-Society)*. [S.l.], 2016. p. 133–134. 47, 49, 51

SCIKIT-LEARN. *1.1.11. Logistic regression*. 2024. <https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression> (Accessed 08 July 2024). Disponível em: <https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression>. 33

SCIKIT-LEARN. *1.4. Support Vector Machines*. 2024. <https://scikit-learn.org/stable/modules/svm.html> (Accessed 08 July 2024). Disponível em: <https://scikit-learn.org/stable/modules/svm.html>. 32

SCIKIT-LEARN. *1.9. Naive Bayes*. 2024. <https://scikit-learn.org/stable/modules/naive_bayes.html> (Accessed 08 July 2024). Disponível em: <https://scikit-learn.org/stable/modules/naive_bayes.html>. 32

SEHGAL, R. et al. Green software: Refactoring approach. *Journal of King Saud University-Computer and Information Sciences*, Elsevier, 2020. 47, 49, 51

SERRANO, N.; HERNANTES, J.; GALLARDO, G. Mobile web apps. *IEEE software*, IEEE, v. 30, n. 5, p. 22–27, 2013. 25

SHAHOOR, A. et al. Leakpair: Proactive repairing of memory leaks in single page web applications. In: IEEE. *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2023. p. 1175–1187. 51

SHAHRIAR, H.; NORTH, S.; MAWANGI, E. Testing of memory leak in android applications. In: IEEE. *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*. [S.l.], 2014. p. 176–183. 47, 49, 51

SHAHROKNI, A.; FELDT, R. A systematic review of software robustness. *Information and Software Technology*, Elsevier, v. 55, n. 1, p. 1–17, 2013. 42

SONG, S.; WEDYAN, F.; JARARWEH, Y. Empirical evaluation of energy consumption for mobile applications. In: IEEE. *2021 12th International Conference on Information and Communication Systems (ICICS)*. [S.l.], 2021. p. 352–357. 47, 49, 51

SONG, W.; ZHANG, J.; HUANG, J. Servdroid: detecting service usage inefficiencies in android applications. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. [S.l.: s.n.], 2019. p. 362–373. 46, 47, 49, 51

STATISTA. *Average consumer spend on mobile apps per smartphone as of 3rd quarter 2023*. 2024. <https://www.statista.com/statistics/289909/mobile-app-spend-on-per-user-mobile-apps-quarter/> (Accessed 08 July 2024). Disponível em: <https://www.statista.com/statistics/289909/mobile-app-spend-on-per-user-mobile-apps-quarter/>. 17

STATISTA. *Market share of mobile operating systems worldwide from 2009 to 2024, by quarter*. 2024. <https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/> (Accessed 08 July 2024). Disponível em: <https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/>. 26

STATISTA. *Number of mobile app downloads worldwide from 2016 to 2023*. 2024. <https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/> (Accessed 08 July 2024). Disponível em: <https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/>. 17

STATISTA. *Number of smartphone mobile network subscriptions worldwide from 2016 to 2023, with forecasts from 2023 to 2028*. 2024. <https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/> (Accessed 08 July 2024). Disponível em: <https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/>. 17

TOFFALINI, F.; SUN, J.; OCHOA, M. Practical static analysis of context leaks in android applications. *Software: Practice and Experience*, Wiley Online Library, v. 49, n. 2, p. 233–251, 2019. 47, 49, 51, 70, 72

VEKRIS, P. et al. Towards verifying android apps for the absence of no-sleep energy bugs. In: *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems*. [S.l.: s.n.], 2012. 20, 47, 49, 51

VILK, J.; BERGER, E. D. Bleak: automatically debugging memory leaks in web applications. *ACM SIGPLAN Notices*, ACM New York, NY, USA, v. 53, n. 4, p. 15–29, 2018. 47, 49, 51

WANG, C. et al. *LLM-based Resource-Oriented Intention Inference for Static Resource Leak Detection*. 2024. Disponível em: <https://arxiv.org/abs/2311.04448>. 51

WU, H.; WANG, Y.; ROUNTEV, A. Sentinel: Generating gui tests for android sensor leaks. In: IEEE. *2018 IEEE/ACM 13th International Workshop on Automation of Software Test (AST)*. [S.l.], 2018. p. 27–33. 49

WU, H.; YANG, S.; ROUNTEV, A. Static detection of energy defect patterns in android applications. In: *Proceedings of the 25th International Conference on Compiler Construction*. [S.l.: s.n.], 2016. p. 185–195. 47, 49, 51

WU, H. et al. Sentinel: generating gui tests for sensor leaks in android and android wear apps. *Software Quality Journal*, Springer, v. 28, n. 1, p. 335–367, 2020. 47, 49, 51

WU, L. et al. Bug analysis of android applications based on jpf. In: SPRINGER. *International Conference on Smart Computing and Communication*. [S.l.], 2016. p. 173–182. 47, 49, 51

WU, T. et al. Light-weight, inter-procedural and callback-aware resource leak detection for android apps. *IEEE Transactions on Software Engineering*, IEEE, v. 42, n. 11, p. 1054–1076, 2016. 40, 47, 49, 51

XIA, M. et al. Why application errors drain battery easily? a study of memory leaks in smartphone apps. In: *Proceedings of the Workshop on Power-Aware Computing and Systems*. [S.l.: s.n.], 2013. p. 1–5. 47, 49, 51

XU, Z.; WEN, C.; QIN, S. State-taint analysis for detecting resource bugs. *Science of Computer Programming*, Elsevier, v. 162, p. 93–109, 2018. 40, 47, 49, 51, 70, 72

YAN, D.; YANG, S.; ROUNTEV, A. Systematic testing for resource leaks in android applications. In: IEEE. *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. [S.l.], 2013. p. 411–420. 28, 30, 42, 47, 49, 51

YUCALAR, F. et al. Multiple-classifiers in software quality engineering: Combining predictors to improve software fault prediction ability. *Engineering Science and Technology, an International Journal*, Elsevier, 2019. 36, 86

ZEIN, S.; SALLEH, N.; GRUNDY, J. Static analysis of android apps for lifecycle conformance. In: IEEE. *2017 8th International Conference on Information Technology (ICIT)*. [S.l.], 2017. p. 102–109. 47, 49, 51

ZHANG, H.; WU, H.; ROUNTEV, A. Automated test generation for detection of leaks in android applications. In: *Proceedings of the 11th International Workshop on Automation of Software Test*. [S.l.: s.n.], 2016. p. 64–70. 40, 47, 49, 51, 69, 72

ZHANG, L. et al. Adel: An automatic detector of energy leaks for smartphone applications. In: *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. [S.l.: s.n.], 2012. p. 363–372. 47, 49, 51, 68, 72

A

Anexos

LimeSurvey

0%

What are the categories of causes of resource leaks in mobile applications?

Hello, we are Software Engineering researchers from Brazil. Our goal is to identify and analyze the relevance of the categories that represent causes of resource leaks in mobile applications.

Resource leaks in mobile applications occur as a result of the incorrect release of a resource (for example, Camera, Media Player or Sensor) (or when the release does not occur), which can result in failures, insufficient responsiveness, unnecessary battery life consumption and negative user experience. Therefore, it is necessary for developers to explicitly request and release such resources.

This academic research has no commercial or profitable interests. The results will be published and can be accessed without restriction, in order to benefit everyone, guaranteeing the anonymity of those who respond to the survey. After the data is collected and analyzed, your email will be removed from the data and will not be used at any time during the analysis or when presenting the results. If, at some point in the survey, you wish to leave, you can do so without negative consequences.

Thanks a lot!

Arilo Claudio Dias Neto - Federal University of Amazonas (UFAM) - arilo@icomp.ufam.edu.br

Josias Gomes Lima - Federal University of Amazonas (UFAM) - josias@icomp.ufam.edu.br

There are 15 questions in this survey.

Next

0%

Your Profile

* 1 On which platforms have you developed mobile applications?

Check all that apply

- [ ] Android
- [ ] iOS
- [ ] Windows
- [ ] Other:

LimeSurvey

* 2 How long (in years) have you been concerned about resource leaks in the mobile applications you develop?

* 3 In which country do you live?

4 What is your email?

Only if you want to receive updates on this survey.

LimeSurvey

50%

Cause categories of resource leaks

* 5 ***Complete leak***: Developers completely fail to release the resources acquired after their use.

Click here to see an example of a complete leak

Please select the option below with respect to the relevance of this category to represent causes of leakage.

	No relevance	Low relevance	Neutral	Medium relevance	High relevance
	○	○	○	○	○

LimeSurvey

* 6 ***Leak on normal paths***: Resources are released only in a few paths in the code.

Click here to see an example of a leak on normal paths

Please select the option below with respect to the relevance of this category to represent causes of leakage.

	No relevance	Low relevance	Neutral	Medium relevance	High relevance
	○	○	○	○	○

* 7 ***Leak on exceptional paths***: Resources are not released if exceptions occur.

Click here to see an example of a leak on exceptional paths

Please select the option below with respect to the relevance of this category to represent causes of leakage.

	No relevance	Low relevance	Neutral	Medium relevance	High relevance
	○	○	○	○	○

LimeSurvey

* 8 ***Leak on irregular paths***: The resources are released in an inappropriate location, that is, the application has a release operation performed only when a specific event occurs, such as ***onClick***, ***onKeyDown*** and so on. If the user does not trigger any of these events, the related resources cannot be released.

Click here to see an example of a leak on irregular paths

Please select the option below with respect to the relevance of this category to represent causes of leakage.

	No relevance	Low relevance	Neutral	Medium relevance	High relevance
	○	○	○	○	○

* 9 ***Leak caused by race condition***: It occurs when the management of a resource can be performed (that is, acquired and released) by different ***threads*** in the application. In the common case, one ***thread*** activates the resource and, some time later, another ***thread*** disables the resource, resulting in the normal behavior of using the component. However, the order in which threads attempt to access the shared resource cannot be expected due to several factors.

Click here to see an example of a leak caused by race condition

Please select the option below with respect to the relevance of this category to represent causes of leakage.

	No relevance	Low relevance	Neutral	Medium relevance	High relevance
	○	○	○	○	○

LimeSurvey

* 10 ***Leak caused by unused resource***: The resource is acquired, but is not used.

Click here to see an example of a leak caused by unused resource

Please select the option below with respect to the relevance of this category to represent causes of leakage.

	No relevance	Low relevance	Neutral	Medium relevance	High relevance
	○	○	○	○	○

* 11 ***Leak caused by complex app lifecycle***: Poor management of resources during the life cycle of a component, either due to carelessness or ignorance of how the Android component life cycle works.

Click here to see an example of a leak caused by complex app lifecycle

Please select the option below with respect to the relevance of this category to represent causes of leakage.

	No relevance	Low relevance	Neutral	Medium relevance	High relevance
	○	○	○	○	○

LimeSurvey

* 12 ***Leak caused by improper allocation and deallocation***: Inadequate allocation and deallocation of space in native memory, misuse of API (Application Programming Interface), lack or loss of references to resource objects can cause an unexpected increase in memory used.

Click here to see an example of a leak caused by improper allocation and deallocation

Please select the option below with respect to the relevance of this category to represent causes of leakage.

	No relevance	Low relevance	Neutral	Medium relevance	High relevance
	○	○	○	○	○

* 13 ***Leak caused by improper resource liberation in loop***: A resource is acquired repeatedly within a loop, but it is not released enough times before exiting the application.

Click here to see an example of a leak caused by improper resource liberation in loop

Please select the option below with respect to the relevance of this category to represent causes of leakage.

	No relevance	Low relevance	Neutral	Medium relevance	High relevance
	○	○	○	○	○

LimeSurvey

* 14 ***Leak caused by Android SDK***: Leaks can be caused by the Android SDK and the developer has few ways to fix them.

Click here to see an example of a leak caused by Android SDK

Please select the option below with respect to the relevance of this category to represent causes of leakage.

	No relevance	Low relevance	Neutral	Medium relevance	High relevance
	○	○	○	○	○

15 What is your opinion about the categories presented? Are there any other categories of causes of leaks?

It is important that we receive your comments.

Previous

Submit

LimeSurvey

Thank you for your participation in this survey!