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

RAQUEL MARIA SANTOS DE OLIVEIRA

Uma Ferramenta para Detecção Estática de Vazamentos de Recursos em Aplicações Android

Recife

2023

RAQUEL MARIA SANTOS DE OLIVEIRA

Uma Ferramenta para Detecção Estática de Vazamentos de Recursos em Aplicações Android

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Engenharia de Software e Linguagens de Programação

Orientador: Leopoldo Motta Teixeira

Recife

2023

Catalogação na fonte
Bibliotecária Nataly Soares Leite Moro, CRB4-1722

O48f Oliveira, Raquel Maria Santos de
Uma ferramenta para detecção estática de vazamentos de recursos em aplicações Android / Raquel Maria Santos de Oliveira – 2023.
55 f.: il., fig.

Orientador: Leopoldo Motta Teixeira.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2023.
Inclui referências.

1. Engenharia de software e linguagens de programação. 2. Android. 3. Vazamento de recursos. 4. Análise estática do código. 5. Java. I. Teixeira, Leopoldo Motta (orientador). II. Título

005.1 CDD (23. ed.) UFPE - CCEN 2023 – 98

Raquel Maria Santos de Oliveira

“Uma Ferramenta para Detecção Estática de Vazamentos de Recursos em Aplicações Android”

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

Aprovado em: 28/02/2023

BANCA EXAMINADORA

_____

Prof.Dr. Breno Alexandro Ferreira de Miranda
Centro de Informática / UFPE

_____

Prof. Dr. Baldoino Fonseca dos Santos Neto
Instituto de Computação / UFAL

_____

Prof. Dr. Leopoldo Motta Teixeira
Centro de Informática / UFPE
(**Orientador**)

Aos meus pais.

AGRADECIMENTOS

Nesse percurso desafiador, a trajetória se tornou possível graças ao apoio e incentivo das pessoas que acreditaram em mim e não desistiram de ficar ao meu lado.

Primeiramente agradeço à Deus por ter me dado forças para continuar esse processo quando eu pensei em inúmeras vezes desistir, nos momentos que me senti fraca e não acreditava na minha capacidade de chegar ao fim.

Sou grata aos meus pais, Laudicéa e Enilton, que são tudo pra mim, sempre me apoiando em tudo, sempre acreditando em mim. Serei eternamente grata por essa base que tenho e todo apoio e incentivo.

Ao meu tio Lula (in memoriam), que sempre torceu pelas minhas conquistas e sei que está orgulhoso por mais um processo e desafio ser concluído pela sua sobrinha preferida.

Agradeço ao apoio e palavras de incentivo que a Silvia me deu, quando duvidei de mim, ela me fez lembrar da minha capacidade e acreditar que tudo ia dar certo.

Sou grata à Bárbara que desde o início dessa trajetória esteve presente e sempre me apoiando em chegar até o fim, obrigada por tudo e por essa ser mais uma etapa que você se fez presente.

Ao meu professor orientador, Leopoldo, por todos ensinamentos e pela oportunidade de me orientar mais uma vez no meio desse mundo acadêmico.

Aos meus familiares e amigos que contribuíram e deram suporte nesse processo, o apoio e suporte de vocês fizeram a diferença.

A todos que me ajudaram de forma direta ou indireta, meus sinceros agradecimentos.

RESUMO

Um problema comum em aplicações desenvolvidas para dispositivos móveis é o vazamento de recursos. Adquirir recursos sem corretamente liberá-los após o seu uso é uma das principais causas. No contexto de dispositivos móveis, estes problemas podem causar danos à experiência dos usuários por conta de problemas com desempenho, travamentos, ou comportamento incorreto. Este trabalho foca em identificar vazamentos de recursos em aplicações desenvolvidas para a plataforma Android. As ferramentas existentes têm como requisito executar a aplicação ou não estão disponíveis publicamente. Este trabalho propõe o FindLeak, uma ferramenta baseada em análise estática para identificar possíveis vazamentos em classes de recursos do tipo Camera, Cursor e MediaPlayer. O FindLeak identifica automaticamente classes que contém possíveis vazamentos de recurso em aplicações Android desenvolvidas em Java. A ferramenta foi avaliada utilizando projetos extraídos de bases de dados previamente definidas em outros trabalhos, como DroidLeaks e AppLeak, assim como minerando repositórios disponíveis no GitHub. Com isto, foi possível aplicar a ferramenta em 966 repositórios de projetos reais, detectando 494 arquivos de classes com vazamentos de recursos. Os resultados fornecem evidência inicial de que o FindLeak pode ser utilizado para auxiliar os desenvolvedores a detectar vazamentos de recursos de forma automática e simples.

Palavras-chave: Android; vazamento de recursos; análise estática do código; aplicativos; java.

ABSTRACT

A common problem with applications developed for mobile devices is resource leaks. One of the main causes is acquiring resources without properly releasing them after use. In the context of mobile devices, these issues can affect the user experience through performance issues, crashes, or incorrect behavior. This work focuses on identifying resource leaks in applications developed for the Android platform. Existing tools require the program to run or are not publicly available. This work proposes FindLeak, a tool based on static analysis to identify possible leaks in resource classes such as Camera, Cursor, and MediaPlayer. FindLeak automatically identifies classes that contain potential resource leaks in Android applications developed in Java. The tool was evaluated using projects extracted from databases previously defined in other work such as DroidLeaks and AppLeak, and by mining repositories available on GitHub. In this way, it was possible to apply the tool to 966 real project repositories and detecting 494 class files with resource leaks. The results are a first proof that FindLeak can help developers to automatically and easily detect resource leaks.

Keywords: Android; resource leaks; static code analysis; apps; java.

LISTA DE FIGURAS

LISTA DE CÓDIGOS

SUMÁRIO

1 INTRODUÇÃO

A tecnologia está avançando cada vez mais e vem trazendo grandes mudanças para a sociedade, onde o uso de smartphones está crescendo muito, o que gera uma expectativa de qualidade das aplicações. Um relatório publicado pela *Strategy Analytics* (ANALYTICS, 2021), uma empresa de pesquisa e consultoria de mercado, informou que no ano de 2021, a estimativa para o número de pessoas que possuem smartphones é de $3,85$ bilhões, o que representa 50% de todo o mundo. Já no evento *Google I/O* de 2021 (ALMENARA, 2022) foi anunciado que cerca de 1 bilhão de dispositivos Android foram ativados, o que equivale que uma a cada oito pessoas na população mundial teria ativado um aparelho no sistema operacional Android no ano de 2020 e essa contagem corresponde apenas a smartphones.

A tendência de números de aplicativos para dispositivos móveis no sistema operacional Android é crescer proporcionalmente com o número de dispositivos ativos. Com esse crescimento, torna-se importante alcançar bons níveis de qualidade no desenvolvimento de aplicações. Os desafios para desenvolver uma aplicação com bom desempenho incluem vários tipos de problemas. O problema alvo desta dissertação está relacionado ao vazamento de recursos que limita a utilização dos recursos dos dispositivos quando não faz a liberação do recurso após o seu uso.

1.1 MOTIVAÇÃO

Buscamos na literatura sobre os tipos de vazamentos que poderiam acontecer ao longo do desenvolvimento e como a comunidade de pesquisa estava tratando esses assuntos. Foram encontrados vários trabalhos relacionados à vazamentos de recursos em vários cenários, alguns trabalhos possuíam tipos de análise dinâmica, como o FunesDroid (AMALFITANO et al., 2020), outros eram voltados para análise estática, como o Relda2 (WU et al., 2016) e até mesmo um misto dos dois tipos de análises, o EnergyPatch (BANERJEE et al., 2018). Diversas ferramentas foram desenvolvidas nos anos recentes, mas ao tentar executar as mesmas, era possível observar um nível de dificuldade para tanto, onde até mesmo algumas não possuíam disponibilidade para seu uso. Também foi possível encontrar alguns exemplos de *Benchmarks* que são banco de dados que armazenam bugs, como por exemplo o AppLeak (RIGANELLI; MICUCCI; MARIANI, 2018) e DroidLeaks (LIU et al., 2019).

Muitas pesquisas vêm propondo técnicas para ajudar os desenvolvedores a gerenciar os recursos do sistema, como de exemplo o FunesDroid (AMALFITANO et al., 2020) que é uma ferramenta de caixa preta para auxiliar na detecção de vazamentos de memória relacionado ao ciclo de vida da Activity. Não foi possível encontrar uma ferramenta que realizasse esse auxílio aos desenvolvedores no decorrer do desenvolvimento da aplicação. Portanto, decidimos focar no suporte a encontrar vazamentos de recurso em tempo de desenvolvimento nesta pesquisa.

O principal objetivo deste trabalho é desenvolver uma ferramenta voltada para análise estática que realiza a detecção de vazamentos de recursos em Aplicações Android, dessa forma, os aplicativos podem melhorar o seu nível de qualidade durante o seu desenvolvimento ao detectar vazamento de forma estática e evitar que aconteçam possíveis falhas com os vazamentos de recursos. A análise será realizada em classes escritas na linguagem Java, que terá como propósito auxiliar os desenvolvedores no processo da implementação de aplicativos para solucionar vazamentos de recursos e melhorar o seu desempenho. A ferramenta foi construída de forma a ser fácil de executar. Como base para a avaliação da ferramenta, usamos o AppLeak e DroidLeaks para servirem de fonte de dados para analisar os tipos de vazamentos de recursos que mais acontecem em aplicações reais e extrair informações dos projetos para o nosso estudo.

O restante deste trabalho está estruturado da seguinte forma:

- Capítulo 2 apresenta a fundamentação teórica que descreve os principais conceitos e tecnologias utilizadas durante o desenvolvimento do trabalho.
- Capítulo 3 descreve a metodologia utilizada para conduzir o estudo. Detalhando sobre as questões de pesquisa e retratando as classes de recursos que foram o objetivo de verificação da ferramenta.
- Capítulo 4 apresenta a ferramenta que foi desenvolvida com todos os seus detalhes de arquitetura, implementação e como foram realizadas as coletas dos repositórios junto com os casos de testes que foram realizados antes de aplicar em um experimento, também são detalhados os resultados que foram obtidos.
- Capítulo 5 apresenta as considerações finais do trabalho, divididas em conclusão e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta a fundamentação necessária para o restante deste trabalho, descrevendo tópicos com os principais conceitos e tecnologias utilizadas durante o estudo para o desenvolvimento da ferramenta.

2.1 DESENVOLVIMENTO ANDROID

Android é um sistema operacional de código aberto que surgiu em 2003, fundado pela Android Inc, e adquirido pela Google em 2005. O Android é um sistema operacional completo com código-fonte personalizável que pode ser portado para praticamente qualquer dispositivo. Aplicativos Android podem ser escritos usando as linguagens de programação Kotlin, Java e linguagens C++ (DEVELOPERS, 2022).

Ao realizar o desenvolvimento de aplicativos Android, é muito importante ter o conhecimento de como funcionam os recursos do sistema para que sejam utilizados de forma adequada. Um bom aproveitamento dos recursos faz com que o aplicativo seja desenvolvido com melhor qualidade e desempenho.

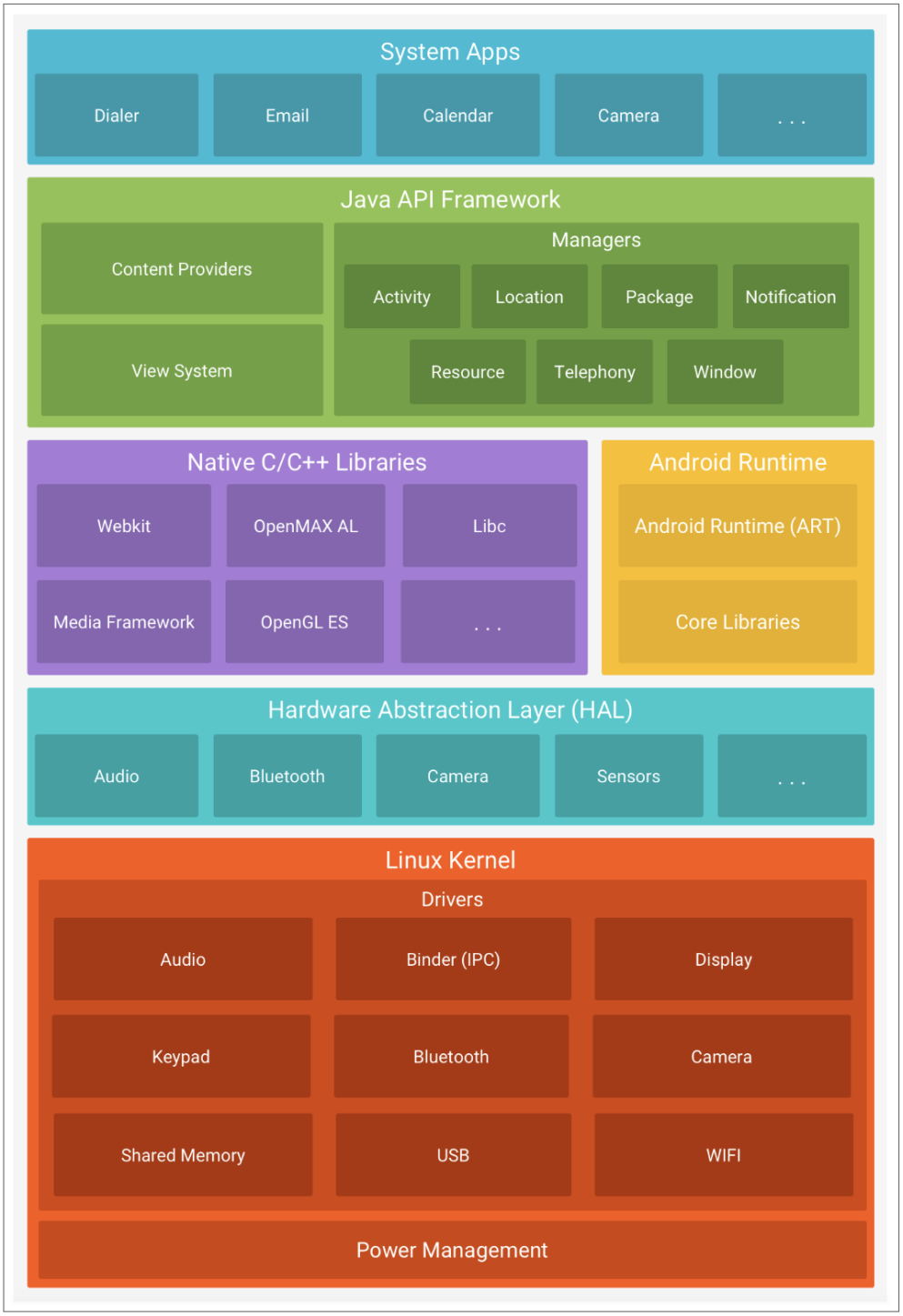
Nos tópicos a seguir, apresentamos a arquitetura da plataforma Android para ter um entendimento de como funcionam as camadas existentes no Android, em seguida serão relatados os principais componentes do Android e uma breve introdução da linguagem Java.

2.1.1 Arquitetura da Plataforma

O Android é uma pilha de software que consiste em algumas camadas onde a base é uma versão customizada do Kernel do Linux. Possui código aberto com uma coleção de várias bibliotecas que são expostas por meio de serviços para diversos dispositivos. A seguir, na Figura 1, é possível visualizar como é distribuído os componentes da plataforma Android.

As camadas da pilha são bem estruturadas e distribuídas, a camada base é dedicada ao Kernel do Linux, seguida pela camada de abstração do hardware, as bibliotecas escritas em linguagem nativa C/C++ juntamente com o Android Runtime é a próxima camada na qual é seguida pela camada de estrutura de Java API e por último, na camada do topo está dedicada aos aplicativos do sistema. Adiante, vamos detalhar brevemente a responsabilidade de cada

Figura 1 – Pilha de Software do Android



Fonte: Documentação Oficial do Android (2020)

camada:

- Kernel do Linux: É a camada base da pilha, responsável por realizar a execução de tarefas de baixo nível. É a camada fundação da plataforma Android, onde seu uso permite que o Android aproveite os recursos de seguranças principais e os fabricantes dos dispositivos

realizem o desenvolvimento da criação de drivers de hardware para um sistema conhecido, bem documentado e independente. Realiza a execução de tarefas de baixo nível mais próximas do hardware do dispositivos, como o gerenciamento de memória.

- Camada de Abstração de Hardware (HAL): A sua principal função é divulgar as capacidades do hardware do dispositivo para a API de alto nível, onde é utilizada pelos desenvolvedores em seus aplicativos. Consiste em módulos de bibliotecas que implementam uma interface para um tipo específico de componente de hardware. Quando uma API realiza a chamada para acessar o hardware do dispositivo, como a Câmera, Sensores ou Bluetooth, o sistema Android carrega o módulo da biblioteca para este componente de hardware.

- Android Runtime: A Android Runtime, ou ART, é um ambiente de tempo de execução gerenciado onde são executados os aplicativos e alguns serviços do sistema. É projetado para executar várias máquinas virtuais em dispositivos de baixa memória, executando arquivos no formato Dalvik Executable, ou DEX, onde encontramos um tipo de bytecode otimizado para o baixo consumo de memória e especialmente projetado para o Android. O ART possui alguns principais recursos como o Garbage Collector otimizado para funcionar com pouco consumo e fragmentação de memória, também possui a melhor compatibilidade de depuração, exceções de diagnóstico detalhadas e geração de relatórios de erros.

- Bibliotecas C/C++ Nativas: Diversos componentes e principais serviços do sistema Android são implementados por código nativo que exige bibliotecas nativas programadas em C e C++. A plataforma Android disponibiliza as Java Framework APIs para deixar visíveis as funcionalidades de algumas dessas bibliotecas nativas aos aplicativos. Como por exemplo, é possível acessar o OpenGL ES pela Java OpenGL API da estrutura do Android para incluir a capacidade de desenhar e manipular gráficos em 2D ou 3D do aplicativo desenvolvido.

- Estrutura da Java API: Essa camada é responsável por conter os componentes necessários para criar um aplicativo utilizando serviços e acessando recursos do sistema operacional. Essa API de alto nível implementa diversas funcionalidades e os recursos pelas APIs são programadas na linguagem Java. As diversas funcionalidades disponibilizadas nessa camada são a criação de views, gerenciamento de recursos, gerenciamento

de notificação, gerenciamento de atividades e provedores de conteúdo. Os desenvolvedores possuem o acesso completo aos mesmos Framework APIs que os aplicativos do sistema Android utilizam.

- Aplicativos do Sistema: O Android possui um conjunto de aplicativos principais como email, calendário, navegador de internet e outros. Os aplicativos que são incluídos na plataforma não possuem um tratamento diferenciado entre os aplicativos que são instalados pelos usuários, dessa forma é possível instalar um aplicativo terceirizado e ser o novo aplicativo de envio de SMS. Os aplicativos do sistema tem sua principal função de fornecer capacidades principais que podem ser acessadas pelos desenvolvedores através dos seus próprios aplicativos. Caso o aplicativo desenvolvido precise realizar o envio de SMS, não é necessário criar uma nova funcionalidade para exercer essa função, basta invocar o aplicativo de SMS que está instalado para realizar esse serviço.

2.1.2 Principais Componentes do Android

Um aplicativo Android tem 4 principais pilares que são os principais componentes que realizam a construção de uma aplicação Android. Cada componente é um ponto de entrada onde o sistema ou o usuário pode entrar no aplicativo, alguns componentes possuem a dependência de outros, eles serão descritos brevemente a seguir.

- Activity: Uma Activity é o ponto de entrada para a interação com o usuário, ela é representada com uma única tela como interface do usuário. Como por exemplo, um aplicativo de enviar SMS que tem uma Activity para mostrar a lista de SMS recebidos e outra para criar um novo SMS. Esse aplicativo possui várias Activities que possuem interações com o usuário, para que ao iniciar a aplicação e uma determinada Activity precise ser a principal, é necessário especificar no código qual a Activity deve ser inicializada primeiro. As Activities do exemplo do aplicativo de SMS, funcionam em conjunto trazendo uma experiência funcional para o usuário, mas elas são independentes entre si. Dessa forma, um aplicativo diferente pode iniciar qualquer Activity, desde que o aplicativo de SMS conceda essa permissão, então um aplicativo qualquer pode invocar uma Activity do SMS para abrir apenas a tela de escrever uma nova mensagem. A Activity traz muitas facilidades de interação para o usuário.

- Service: Um Service é um componente executado em segundo plano que realiza operações de longa duração ou trabalho para processos remotos. Esse componente não apresenta uma interface para o usuário, mas ele é um componente que realiza execuções para experiência do usuário, por exemplo, um service pode tocar uma música em segundo plano enquanto o usuário está realizando a interação com outra aplicação. Uma activity pode iniciar um Service e deixá-lo executar ou ter um vínculo com ele para realizar a interação.

- Broadcast Receiver: Os Broadcast Receivers respondem a mensagens de Broadcast de outros aplicativos ou sistema. É um componente que faz o sistema entregar eventos ao aplicativo que não estão no fluxo comum do usuário, permitindo que o aplicativo responda a anúncios de transmissão por todo o sistema. É possível o sistema entregar transmissões para aplicativos que não estão em execução no momento. Por exemplo, um aplicativo de alarme pode ser programado para notificar ao usuário sobre um evento futuro, ao entregar esse alarme ao receptor de transmissão, o aplicativo não precisa ficar em execução para o alarme ser disparado.

- Content Providers: Um Content Provider disponibiliza dados de um aplicativo para outros diante de solicitações. É um gerenciamento de compartilhamento de dados do aplicativo que podem ser armazenados em banco de dados ou em qualquer local de armazenamento persistente e acessível ao aplicativo. Por meio do Content Provider, outros aplicativos podem consultar ou até modificar dados, caso seja permitido.

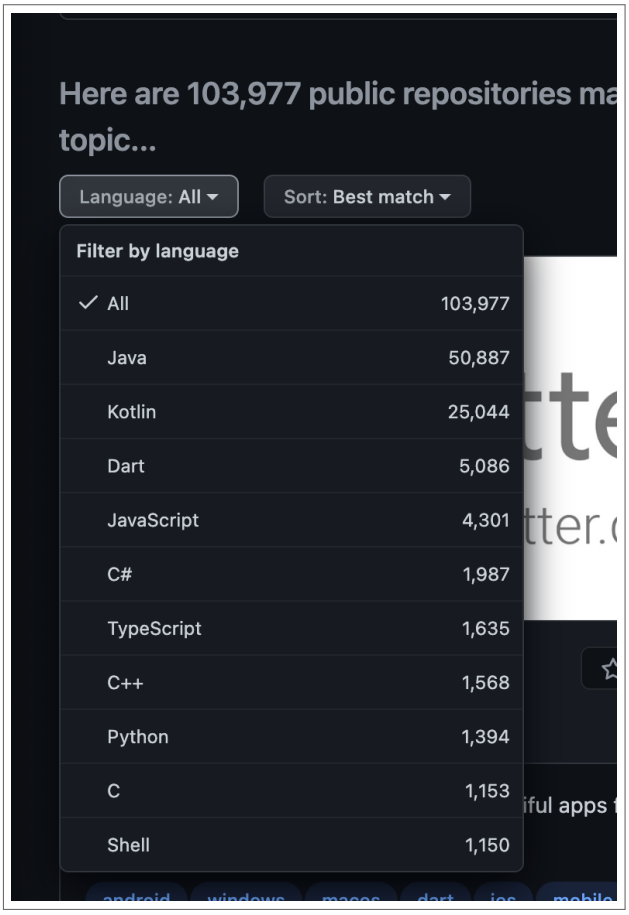
2.1.3 Linguagem de Programação Java

Desde o início, a linguagem de programação utilizada para o desenvolvimento de aplicações Android é o Java que por um tempo foi determinada a linguagem oficial, mas em 2017 foi anunciado a mudança da linguagem oficial para o Kotlin que se tornou possível realizar a interoperabilidade com o Java, essa mudança aconteceu com o objetivo de trazer melhorias em relação aos problemas que estavam acontecendo nos processos de desenvolvimento que era utilizado com a linguagem Java.

Mesmo diante da mudança da linguagem oficial, muitos projetos ainda são desenvolvidos em Java e projetos mais antigos ainda não fizeram a migração, possuindo muita parte do código legado, ou seja, partes dos projetos que não avançaram no ritmo das atualizações da tecnologia

e pode gerar uma manutenção de alto custo, isso acontece com determinada frequência em aplicativos que são mais antigos. Foi possível identificar através de uma pesquisa no GitHub[1] que é uma plataforma de hospedagem de código fonte, uma filtragem pelo tópico Android e o número de projetos que possuem a linguagem em Java chega em torno de 50% da quantidade total de projetos, como mostra na Figura 2, então é possível observar que a quantidade de aplicações que utilizam o Java como uma das linguagens para o desenvolvimento é muito alta.

Figura 2 – Quantidade de Repositórios baseados na Linguagem de Programação



Fonte: Github (2023)

2.2 ANÁLISE ESTÁTICA DE CÓDIGO

A análise estática de código é uma prática que faz a verificação da qualidade do código no código fonte e consiste em uma série de verificações automatizadas realizadas no código fonte (JETBRAINS, 2000). Ferramentas que realizam a análise estática fazem a avaliação do software de forma abstrata, sem precisar executar o software (AYEWAH et al., 2008). Essa

[1] <https://github.com/>

técnica é bastante utilizada na engenharia de software para evitar e/ou amenizar erros. Sua popularidade como técnica de inspeção de software no processo de verificação e validação se dá por conta da simplicidade de não precisar executar a aplicação com entradas, bastando o código-fonte. A análise estática possui algumas limitações porque as ferramentas ficam restritas a um conhecimento pré estabelecido, a verificação é realizada com base em um conjunto de regras. Elas só apontam vulnerabilidades em pontos que foram programados para detectar (TEIXEIRA, 2007).

A ferramenta desenvolvida neste trabalho utiliza a análise estática para verificar o código em busca de vazamentos de recursos que estão dentro do escopo dela, essa busca é realizada através de algumas regras de codificação que foram implementadas para esses tipos de análises de vazamentos. O processo de utilizar a análise estática traz grandes benefícios por apontar qual a classe que foi encontrada o vazamento e isso traz muita facilidade para futuras correções, além do tempo que pode ser reduzido ao desenvolvedor para encontrar o erro e também por ser bastante prático. Para acontecer a verificação dos projetos a serem analisados através da ferramenta, é importante ficar atento se as classes possuem acesso público, pois como apresenta uma verificação no código fonte e se por um acaso o acesso estiver no modo privado, a ferramenta irá apontar uma exceção e a verificação não poderá ser realizada corretamente, dificultando assim sua execução.

2.2.1 JavaParser

O analisador de código automatiza a maior parte do processo de análise do código fonte, realizando um método de depuração do programa examinando o código, com esse processo é possível validar a estrutura de acordo com os padrões que serão analisados para os vazamentos de recursos. Como irá ser realizada a análise estática de código, é necessário utilizar uma ferramenta para executar essa etapa, dessa forma, foi utilizado o JavaParser[2] que analisa o código fonte escrito em Java e procura os padrões do interesse da análise, facilitando a busca dos padrões de vazamentos de recursos.

O JavaParser é uma biblioteca voltada para classes escritas em Java e faz a análise do código baseada em uma árvore de sintaxe abstrata (AST), permitindo que seja possível navegar e identificar expressões, variáveis declaradas e outras formas de acordo com a necessidade. O JavaParser foi utilizado para realizar a navegação da AST e antes de explorar as branches que

[2] <https://javaparser.org>

podem ser analisadas, é necessário adicionar a configuração com base no arquivo que vai ser percorrido. Por exemplo, como podemos ver no Código 1.

Código Fonte 1 – Exemplo de configuração do JavaParser

```
1
  CompilationUnit compilationUnit = setupConfiguration(filePath);
3
  private static CompilationUnit setupConfiguration(String filePath) throws
      IOException {
5   Path projectRoot = FileSystems.getDefault().getPath(filePath);
    TypeSolver typeSolver = new ReflectionTypeSolver();
7   JavaSymbolSolver symbolSolver = new JavaSymbolSolver(typeSolver);
    StaticJavaParser.getConfiguration().setSymbolResolver(symbolSolver);
9
    return StaticJavaParser.parse(projectRoot);
11 }
```

Após a configuração, precisamos para este trabalho extrair expressões de variáveis que são inicializadas para identificar a utilização das classes de recursos, o exemplo do Código 2 mostra como realizar a extração da variável.

Código Fonte 2 – Exemplo para obter expressões de uma Classe

```
1
  private static void getExpression() {
3       compilationUnit.findAll(VariableDeclarationExpr.class)
                .forEach(expression -> {
5           String variableDeclaration = String.valueOf(expression);
        });
7   }
```

Foi imprescindível verificar se o método de fechamento de recursos estava sendo utilizado corretamente, para isso, foi necessário verificar todos os métodos que são chamados em branches do código, como podemos ver no Código 3.

Código Fonte 3 – Exemplo para extrair Métodos que são chamados na Clase

```
1
  private static void getMethods() {
3       compilationUnit.findAll(MethodCallExpr.class)
                .forEach(method -> {
5           String method = String.valueOf(method);
        });
7   }
```

2.3 VAZAMENTO DE RECURSOS

No Android existem vários tipos de recursos que estão disponíveis para serem utilizados na aplicação. Vazamentos de recursos acontece quando existe o consumo de recursos por uma aplicação e essa aplicação não libera os recursos adquiridos após serem utilizados. A implementação desses recursos deve ser realizada de forma correta, quando isso não acontece pode causar vazamentos. Essa condição normalmente pode gerar diferentes tipos de consequências, como desempenho ruim para o aplicativo, travamentos e até falhas.

Os vazamentos de recursos geralmente ocorrem devido a erros de programação e um dos exemplos comuns acontece na falha ao fechar os recursos que foram abertos. Existem alguns tipos de recursos que são utilizados durante o desenvolvimento das aplicações onde os vazamentos ocorrem devido a ausência do seu fechamento.

Os vazamentos podem ser analisados verificando se o recurso foi criado e, após seu uso, se foi fechado corretamente, enquanto outros tipos de vazamentos precisam de mais informações para evitar que aconteça um vazamento. Durante o nosso estudo conseguimos observar que existem várias formas de evitar vazamentos de recursos e que cada recurso tem a sua peculiaridade em relação a ser utilizado corretamente. A nossa ferramenta FindLeak tem o foco em detectar, através de análise estática, os vazamentos que são criados e não são liberados corretamente. No tópico a seguir, iremos relatar algumas possíveis causas de vazamentos e suas possíveis soluções.

2.3.1 Possíveis Causas e Soluções

Muitas aplicações utilizam vários recursos e chegam a possuir um desempenho ruim, pois ao longo do desenvolvimento alguns recursos foram ignorados em darem mais atenção para obterem seu uso corretamente pelos desenvolvedores durante a implementação.

Alguns recursos têm uma frequência alta de utilização pelos desenvolvedores de aplicações Android. A seguir listamos alguns exemplos de possíveis causas para vazamentos, como também algumas soluções para serem aplicadas no código para evitar tais problemas.

- Singleton: É um padrão de projeto que garante a existência de apenas uma instância de

um objeto. No cenário de aplicações Android, quando o Singleton tem o ciclo de vida de acordo com a aplicação e possui uma referência de uma Activity, podem facilmente serem encontrados alguns fluxos que causam vazamentos. Por exemplo, ao rotacionar o dispositivo, o objeto que corresponde a uma Activity é criado novamente, e como o Singleton tinha a primeira referência antes do rotacionamento, o objeto continua existindo e não é eliminado. Desse modo, após o rotacionamento, uma nova referência é criada e passada para o Singleton. Se por acaso esse cenário ocorrer várias vezes, a memória irá ficar lotada com referências sem uso, por não serem eliminadas.

Neste cenário, uma solução para esse problema é usar referências para o contexto da aplicação ao invés da Activity, como mostra no Código 4.

Código Fonte 4 – Uso da referência de contexto da aplicação

```
1 SingletonManager.getInstance(getApplicationContext());
```

Quando o contexto da Activity for necessário, deve remover no método *onDestroy* verificando se o contexto que foi passado para a classe Singleton está nulo como ser visto no Código 5.

Código Fonte 5 – Remover o contexto no onDestroy

```
1 public void onDestroy() {
      if(context != null) {
3         context = null;
      }
5 }
```

Se o objeto Singleton possuir uma referência a uma atividade e durar mais do que sua atividade, então o vazamento irá acontecer, para isso, pode fornecer um método no objeto Singleton que limpe a referência no método *onDestroy* .

- Broadcast Receivers: São componentes do aplicativo que permitem o recebimento de eventos transmitidos pelo sistema ou por outros aplicativos. Quando é necessário registrar um receptor em uma Activity, é necessário realizar o cancelamento, pois mesmo que a Activity seja fechada, a referência para o objeto permanece. No Código 6 é possível ver um exemplo de registro e cancelamento.

Código Fonte 6 – Registro e Cancelamento do BroadcastReceiver

```
1  public class BroadcastReceiverActivity extends AppCompatActivity {

3      private BroadcastReceiver broadcastReceiver;

5      @Override
       protected void onStart() {
7          super.onStart();
           broadcastReceiver = new BroadcastReceiver();
9          IntentFilter filter = new IntentFilter(ConnectivityManager.
               CONNECTIVITY_ACTION);
           filter.addAction(Intent.ACTION_AIRPLANE_MODE_CHANGED);
11         this.registerReceiver(broadcastReceiver, filter);
       }
13
       @Override
15     protected void onStop() {
           super.onStop();
17         if(broadcastReceiver != null) {
               unregisterReceiver(broadcastReceiver);
19         }
       }
21 }
```

É muito importante também verificar onde está realizando o registro e cancelamento do receptor, caso o registro seja no método *onCreate* o cancelamento tem que ser no *onDestroy* , pois evita que ocorra o vazamento do contexto da Activity. E caso seja registrado no *onResume* é necessário cancelar o registro em *onPause* para evitar que seja registrado várias vezes caso não queira receber transmissão quando pausado (DEVELOPERS, 2022). Realizando esse processo, pode reduzir a sobrecarga desnecessária do sistema.

- Threads: Dada a execução de uma aplicação, o sistema cria um thread de execução que é chamado de principal. Essa thread é encarregada dos eventos da interface do usuário. As atividades de gerenciamento de interface como também outros componentes importantes se concentram na thread. Por isso é essencial que a capacidade de resposta da interface do usuário do aplicativo não seja bloqueada na thread principal. Caso seja necessário realizar operações que não sejam instantâneas, é preciso realizar em threads separadas, chamadas de segundo plano.

O uso do AsyncTask permite que seja utilizado de forma assíncrona na interface do usuário. Ela realiza as atividades em uma thread separada e publica os resultados da thread de interface sem que precise lidar com os gerenciadores. No Código 7 pode ser visto o exemplo de chamada do AsyncTask.

Código Fonte 7 – Exemplo de chamada do AsyncTask

```
1
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
3     protected Long doInBackground(URL... urls) {

5     }

7     protected void onProgressUpdate(Integer... progress) {

9     }

11    protected void onPostExecute(Long result) {

13    }
 }
15
new DownloadFilesTask().execute(url1, url2, url3);
```

O cancelamento dessa task pode ser realizada a qualquer momento chamando o *cancel(boolean)* .

2.4 TRABALHOS RELACIONADOS

Nesta seção será apresentada uma análise das ferramentas e benchmarks que realizaram diferentes abordagens de propostas destinadas a vazamentos de recursos em aplicações Android. As ferramentas citadas nos trabalhos analisados não possuíam artefatos disponibilizados para serem testados, algumas ferramentas possuem links para seu direcionamente que não estavam funcionando, enquanto outras ferramentas não tinham sucesso na sua execução, dessa forma, impossibilita a reprodução para a realização de testes.

2.4.1 FunesDroid

FunesDroid (AMALFITANO et al., 2020) é uma ferramenta de caixa preta para detecção automática de vazamento de memória nos ciclos de vida da Activity. Implementa uma abordagem de teste que pode encontrar vazamentos de memória, analisando duplicações desnecessárias de pilhas de objetos após a execução de três sequências diferentes de ciclo de vida dos eventos. A ferramenta é completamente caixa preta no qual não precisa do código do app, apenas a apk. Ele compara o estado da memória antes da execução do evento e depois da execução. Caso o código da aplicação seja disponibilizado, a informação fornecida pela ferramenta também pode ser explorada pelo desenvolvedor para encontrar a causa raiz do vazamento.

A ferramenta não depende de nenhuma técnica de exploração de aplicativo e é capaz de testar diretamente as acitivities do aplicativo. Tem o objetivo de avaliar o tamanho e a tendência de crescimento dos vazamentos de memória detectados.

Uma diferença importante entre o FunesDroid e o FindLeak é o foco na detecção de vazamentos de recursos comparado com os vazamentos de memória. Enquanto o FindLeak se concentra em identificar vazamentos de recursos específicos, como Camera, Cursor e MediaPlayer, o FunesDroid é direcionado à detecção de vazamentos de memória nos ciclos de vida da Activity. Outra diferença é o modo de funcionamento das ferramentas. O FindLeak é baseado em análise estática e requer acesso ao código fonte da aplicação para identificar possíveis vazamentos de recursos. Por outro lado, o FunesDroid é uma ferramenta de caixa preta que analisa o estado da memória antes e depois da execução de eventos, sem precisar do código fonte da aplicação.

2.4.2 Relda2

Relda2 (WU et al., 2016) trabalha diretamente com o byte-code da aplicação e para fazer a troca entre escalabilidade e precisão, a ferramenta suporta duas análises técnicas: flow insensitive que ignora o fluxo de controle de informações e pode rapidamente analisar um app e o flow sensitive que pode eliminar um número de falso negativo com mais custo de tempo, porém mais aceitável.

Analisa diretamente o arquivo .apk do aplicativo para detecção de bug e com um foco em recursos específicos do Android, como câmera, reprodutor de mídia e sensores. Identifica os pontos para liberação dos recursos e pode ser configurado para realizar análise flow-sensitive

ou flow-insensitive para se adaptar a diferentes requisitos de precisão e eficiência de análise. Não pode capturar fluxos de controle entre threads e, portanto, perderia erros ou relataria alarmes falsos quando as operações de aquisição e liberação de recursos residem em dois threads diferentes.

O FindLeak e Relda2 têm o objetivo de detectar bugs e vazamentos de recursos em aplicações Android, cada uma com o seu foco em recursos específicos do Android. Ambas as ferramentas apresentam abordagens e características distintas, cada uma com suas vantagens e limitações.

2.4.3 DroidLeaks

DroidLeaks (LIU et al., 2019) é um banco de dados de bugs, a limitação do DroidLeaks é que não fornece casos de teste que podem acionar os bugs de vazamento de recursos, isso ocorre porque a maioria dos aplicativos que foram estudados não possuem suítes de teste associadas. Outra limitação é que não foi estudado o impacto do vazamento de recursos que foram identificados. DroidLeaks é limitado em usabilidade e contém apenas código fonte de potencialmente aplicativos defeituosos e não inclui artefatos que podem confirmar a presença de falhas e facilitar a reprodução, ou seja, o código executável dos aplicativos com falha não foi coletado e os casos de testes que revelam as falhas não foram implementados.

O FindLeak e DroidLeaks tem uma principal diferença entre elas, que é a abordagem utilizada. Enquanto o FindLeak oferece uma ferramenta específica para análise estática de vazamentos de recursos, o DroidLeaks é um banco de dados de bugs que fornece informações sobre falhas conhecidas, mas não oferece a mesma capacidade de detecção automatizada dos vazamentos.

2.4.4 AppLeak

AppLeak (RIGANELLI; MICUCCI; MARIANI, 2018) é um benchmark resultante de coleta, análise e reprodução de vários vazamentos de recursos que afetam aplicativos. Pode ser utilizado por pesquisadores para avaliar e comparar técnicas de análise estática e dinâmica. Para cada vazamento de recursos, é disponibilizado o código fonte e o código executável do aplicativo com falhas e do aplicativo corrigido, o caso de teste que exerce o vazamento de recursos e informações sobre configurações específicas que são necessárias para reproduzir o problema.

Em comparação com os outros benchmarks, o AppLeak inclui originalmente todos os artefatos necessários para reproduzir o vazamento de recursos com o mínimo de esforço.

Enquanto o FindLeak é uma ferramenta de análise estática para detecção de vazamentos de recursos, o AppLeak é um benchmark que fornece uma coleção de casos de vazamento de recursos e os artefatos necessários para reproduzi-los. Ambas as ferramentas têm objetivos diferentes, mas podem ser complementares na avaliação e desenvolvimento de técnicas de detecção de vazamentos de recursos.

2.4.5 LeakDroid

LeakDroid (YAN; YANG; ROUNTEV, 2013) é uma ferramenta que gera testes percorrendo caminhos no modelo de GUI. A abordagem é baseada em um modelo de GUI com critérios de cobertura baseados por ciclos neutros. Ciclos neutros é uma sequência de eventos da GUI que deve ter efeitos neutros, ou seja, não levar a aumentos de uso de recursos. Alguns percursos de um ciclo neutro: girar a tela várias vezes, alternar entre aplicativos, abrir e fechar um arquivo repetidamente. É uma ferramenta para geração automática de casos de teste expondo vazamento de memória e problemas de threading em aplicativos Android. Técnica baseada em modelo GUI para produzir casos de teste que realizam execuções repetidas de eventos que devem ter efeitos neutros no consumo de recursos e não devem levar a aumentos no uso de recursos, seu foco está em memory, thread e binder. Identificam uma sequência de ações perigosas do usuário que geralmente levam a um vazamento e as reproduzem automaticamente para simular comportamentos perigosos.

FindLeak é uma ferramenta de análise estática para detecção de vazamentos de recursos em classes específicas, enquanto o LeakDroid é uma ferramenta que gera testes baseados em modelos de GUI para detectar vazamentos de memória e problemas de threading, dessa forma, o foco de cada ferramenta é diferente.

2.4.6 Relda

Relda (GUO et al., 2013) analisa automaticamente as operações de recursos e localiza os vazamentos. A análise é realizada em três recursos: recursos exclusivos, recursos que consomem memória e recursos que consomem energia. A ferramenta recebe os .apks como entrada e depois de terminar a execução, os itens de vazamentos são relatados e também os arquivos

que registram onde cada recurso é solicitado é liberado.

FindLeak utiliza análise estática do código fonte para identificar possíveis vazamentos, a ferramenta Relda analisa as operações de recursos diretamente nos arquivos .apks. Além disso, o escopo da análise também difere, com o FindLeak focado em classes de recursos específicos e o Relda abrangendo recursos exclusivos, de memória e de consumo de energia.

2.4.7 EnergyPatch

EnergyPatch (BANERJEE et al., 2018) utiliza combinação de análise estática e dinâmica para detectar, validar e reparar bugs de energia em aplicativos Android. Um bug de energia é quando um aplicativo não libera todos os recursos que consomem muita energia adquiridos por ele durante a execução, antes de interromper a execução. Bugs de energia afeta indiretamente a vida útil da bateria e a experiência do usuário. Os problemas apresentados por bugs de energia tem características diferentes dos vazamentos de memória e recursos.

O FindLeak é uma ferramenta de análise estática focada na detecção de vazamentos de recursos específicos, enquanto o EnergyPatch utiliza análise estática e dinâmica para identificar e reparar bugs de energia em aplicativos Android.

3 METODOLOGIA

O objetivo deste trabalho é realizar uma análise estática em aplicações Android escritas em Java para detectar vazamentos de recursos e identificar o arquivo que está apresentando o vazamento. Como um guia do estudo, foram estabelecidas algumas questões de pesquisa que irão ser apresentadas neste capítulo.

3.1 QUESTÃO DE PESQUISA

Com base nos objetivos, é possível identificar a questão de pesquisa.

RQ_1: Qual é a precisão da detecção de vazamentos de recursos em aplicações Android escritas em Java, utilizando uma abordagem de análise estática simples baseada em ASTs?

$RQ_1.1$: É possível identificar o arquivo específico que está apresentando o vazamento?

A questão tem por objetivo ajudar a entender a forma que irá ser desenvolvida a análise estática para detectar os vazamentos de recursos, com o propósito de identificar e buscar as classes que são apontadas com possíveis vazamentos de forma simples para o desenvolvedor ter a facilidade de realizar a correção.

3.2 CLASSES DE RECURSOS

Dispositivos Android disponibilizam um hardware com grande capacidade e com muitos recursos do tipo: Câmera, Bluetooth, Cursor, banco de dados SQLite, Media Player, entre outros tipos de recursos.

O FindLeak atualmente tem o foco em três tipos de recursos disponíveis nos dispositivos Android. Esses tipos serão analisados nas aplicações para detectar se estão sendo utilizados corretamente para evitar possíveis vazamentos. Os tipos das classes de recursos verificados pela ferramenta são: Camera, Cursor e MediaPlayer. Esses tipos possuem um padrão nas implementações referentes a criação e ao fechamento do recurso, dessa forma, elas foram selecionadas para serem analisadas pelo FindLeak. Nos tópicos a seguir, serão detalhadas cada classe de recurso e como funciona a implementação para o seu uso no desenvolvimento de aplicações Android, além do mais será indicado a forma que deve ser utilizado durante a implementação para evitar vazamentos.

3.2.1 Camera

O Android disponibiliza acesso total ao hardware da câmera do dispositivo, permitindo a criação de aplicativos de câmera. Ou, caso seja necessário apenas fornecer para o usuário tirar uma foto, é preciso apenas solicitar que o app de câmera capture uma foto e a retorne (DEVELOPERS, 2022).

A classe Camera foi descontinuada e é recomendado utilizar o CameraX, ou para alguns casos específicos, Camera2. O CameraX e Camera2 oferecem suporte ao Android 5.0 (nível 21 da API) e versões mais recentes. Como nosso estudo está sendo baseado ainda em versões de aplicações que estão utilizando a linguagem Java e muitas dessas aplicações ainda não fizeram suas atualizações, então escolhemos fazer o uso da classe Camera para produzir a análise, pois a implementação da análise do CameraX ou Camera2 seria um esforço adicional para essa fase da ferramenta.

Para ter acesso a Camera, o primeiro processo que deve ser realizado é abrir a Camera em uma Thread. O recomendado é que esse processo seja realizado no método *onCreate* , pois uilizando essa abordagem, já que realizar o acesso à câmera pode ser um processo demorado e corre o risco de sobrecarregar a linha de execução de UI, então é uma boa prática. Para casos de implementações mais básicas, o processo de abrir a câmera pode ser utilizado no método *onResume* que facilitará a reutilização do código e deixa o fluxo de controle simples. A abertura da Camera é implementada como mostra no Código 8.

Código Fonte 8 – Realizando a abertura da classe Camera

```
1  /** Uma maneira segura de obter uma instancia do objeto Camera. */
   public static Camera getCameraInstance(){
3      Camera c = null;
       try {
5          c = Camera.open(); // tentativa de obter uma instancia
       }
7      catch (Exception e){
           // nao esta disponivel (em uso ou nao existe)
9      }
       return c; // retorna null se nao estiver disponivel
11 }
```

Fonte: DEVELOPERS (2022)

Para níveis de API mais recentes e nível 9, a biblioteca da câmera oferece suporte para várias câmeras. Caso ocorra a utilização da API legada e for chamado o *open* sem passar nenhum argumento, a câmera que será utilizada é a primeira câmera traseira. Dessa forma,

a ferramenta está analisando se o método *open* está sendo chamado, independente se passar argumentos ou não, ou seja, está abrangendo as opções dos tipos de chamada.

É importante também registrar qualquer exceção quando está sendo utilizado o método *Camera.open* pois se não acontecer e caso a câmera esteja sendo utilizada ou não exista, ocorre da aplicação ser encerrada pelo sistema. O exemplo de tratamento de exceção, é exibido no Código 8.

O hardware da câmera é um recurso que permite compartilhamento, portanto precisa ter um gerenciamento cauteloso para que o aplicativo não entre em conflito com outras aplicações que também possam querer utilizá-lo. Para isso, após fazer o uso da classe Camera, o aplicativo precisa liberar de forma adequada para que outros aplicativos possam utilizá-lo. A liberação é implementada como é ilustrada no Código 9.

Código Fonte 9 – Realizando a liberação da classe Camera

```
1 private void releaseCamera() {
      if (camera != null) {
3         camera.release();
          camera = null;
5     }
  }
```

Fonte: DEVELOPERS (2022)

Para evitar que ocorra vazamento do recurso, é importante ocorrer a liberação do objeto Camera e isso acontece chamando o *Camera.release* sendo assim as tentativas de acesso ao objeto pelo próprio aplicativo ou por outros, ocorrem de maneira adequada, evitando apresentar falhas e que o aplicativo seja encerrado indevidamente.

3.2.2 Cursor

O Cursor é uma interface utilizada no Android para realizar leitura ou gravação de informações através de algum retorno utilizado pelo banco de dados. Na utilização do Cursor, não é necessário ter a preocupação com a implementação concreta, pois basta saber o uso dos métodos necessários que já são implementados. A principal finalidade básica do Cursor é apontar para uma linha de resultado que foi realizada na busca da consulta e esse carregamento é realizado pelo objeto Cursor.

Para a criação do objeto Cursor é necessário utilizar o banco de dados que vai ser realizado a consulta, basta adicionar o comando como mostra no Código 10.

Código Fonte 10 – Criando o objeto do Cursor

```
Cursor cursor = sqLiteDatabase.rawQuery(query, null);
```

Como padrão para os recursos que são utilizados, também é de extrema importância que após o seu uso ele seja fechado corretamente. Para tanto, basta utilizar o método *close* como mostra o Código 11.

Código Fonte 11 – Fechando o objeto do Cursor

```
1  cursor.close();
```

3.2.3 MediaPlayer

O dispositivo Android oferece compatibilidade com diversos tipos de mídias comuns, podendo obter integração com áudio, vídeo e imagens para os aplicativos. Esses recursos de mídia são realizados através da API MediaPlayer que também possibilita reproduzir áudio ou vídeo de arquivos de mídia armazenados no aplicativo, esta é a classe principal para reprodução de som e vídeo. Existem vários formatos de mídia compatíveis com o Android, como também várias fontes que podem ser utilizadas na mídia, por exemplo, pode ser através de recursos locais, URLs externas e URIs internas, é permitido ter várias possibilidade e formas de uso.

Para utilizar a classe do MediaPlayer na aplicação basta adicioar o comando como mostra no Código 12

Código Fonte 12 – Utilizando a classe MediaPlayer

```
1
   MediaPlayer mediaPlayer = new MediaPlayer();
3    ...
   mediaPlayer.start();
```

Para garantir que a aplicação está utilizando corretamente a classe é necessário realizar a liberação do recurso, pois o MediaPlayer pode consumir recursos valiosos do sistema. Por isso, precisa garantir que não está amarrando uma instância por tempo além do suficiente para seu uso. Então, ao terminar de utilizar, deve chamar o método *release* para garantir que os recursos do sistema que foram alocados, sejam liberados de forma correta. Um exemplo em que a aplicação está utilizando uma MediaPlayer e a sua atividade recebe uma chamada para

o *onStop* então deve liberar a MediaPlayer, pois não irá ser mais utilizada já que o usuário não está realizando a interação, só iria ser possível continuar o uso se a aplicação estivesse sendo executada a mídia em segundo plano, daí iria entrar em outro cenário. Dessa forma, quando a atividade é retomada ou reiniciada, uma nova instância da MediaPlayer será criada e deverá ser preparada novamente para utilização (DEVELOPERS, 2022).

Para evitar esse vazamento de recursos, a forma para realizar a liberação e anular a MediaPlayer é ilustrada no Código 13.

Código Fonte 13 – Realizando a liberação da classe MediaPlayer

```
1
    mediaPlayer.release();
3   mediaPlayer = null;
```

4 FERRAMENTA

Este capítulo irá apresentar o FindLeak, uma ferramenta voltada para detectar vazamentos de recursos em aplicações Android. A seguir iremos dar uma introdução sobre a ferramenta e depois será organizado da seguinte ordem: Arquitetura, Implementação, Coleta de Repositórios, Casos de Teste, Experimento e por fim, apresenta os Resultados.

FindLeak é uma ferramenta automatizada que tem como principal objetivo analisar projetos de aplicações Android e identificar qual arquivo do projeto está causando vazamentos de recursos e qual o tipo de vazamento foi detectado. As classes de vazamentos de recursos que a ferramenta analisa são de três tipos: Camera, Cursor e Media Player. Caso o projeto esteja utilizando um desses três tipos de recursos e precisa analisar se está sendo utilizado de forma correta, o FindLeak ajuda a relatar se o projeto está causando vazamentos ou não.

4.1 FLUXO DE FUNCIONAMENTO

Como pré-requisito para rodar o script da ferramenta é necessário ter um arquivo em .txt com a lista de todos os caminhos locais dos projetos que irão ser analisados pela ferramenta. O executável tem a extensão .jar e ao ser rodado precisa passar três parâmetros como entrada, o primeiro argumento é o caminho do arquivo .txt que foi criado anteriormente com todos os caminhos locais dos projetos, o segundo argumento é o caminho que deseja salvar o documento que a ferramenta irá criar com todas as classes .java dos projetos selecionados que irão ser processadas e por fim, o terceiro argumento é o caminho onde deseja salvar o arquivo no formato em excel que será criado com todas as informações sobre os projetos que tem vazamento de recursos, esse arquivo irá ser o artefato final que traz informações do tipo de vazamento de recurso detectado e o caminho detalhado do arquivo que foi apontado o vazamento. No Código 14 é possível observar o exemplo de como executar a ferramenta e como são passados seus argumentos de forma correta.

Código Fonte 14 – Linha de comando para rodar o FindLeak

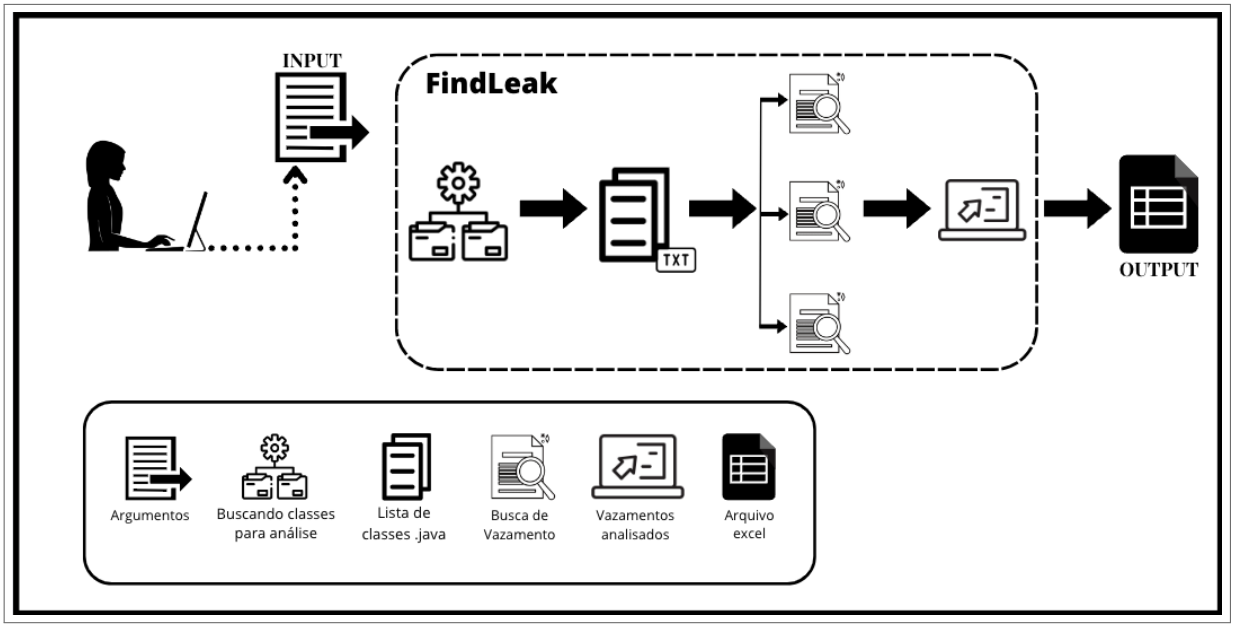
```
java -jar FindLeak.jar /Users/Documents/diretories.txt /Users/Documents/
    filesInJava.txt /Users/Documents/FindLeaks.xlsx
```

Após receber as informações das entradas necessárias, a ferramenta faz uma análise dos projetos que foram adicionados na lista e cria um arquivo .txt listando todas as classes Java

existentes nos projetos. Como a ferramenta faz a análise apenas de códigos escritos na linguagem Java, então ela realiza a extração de todos os arquivos existentes com a extensão .java para poder analisar posteriormente. Quando o arquivo com todas as classes em Java que irão passar por análises for criado, ele irá passar por uma varredura em cada tipo de recurso que a ferramenta suporta e verifica se existe vazamento, e para cada detecção de um vazamento em determinado projeto, a informação do tipo de vazamento e o caminho do arquivo que está com esse vazamento é armazenada. No fim da execução, essas informações são adicionadas no arquivo final que é gerado em formato de uma planilha e salvo no caminho que foi adicionado como argumento ao executar a ferramenta.

A Figura 3 mostra como funciona todo o fluxo de arquitetura do FindLeak, desde o fluxo de entrada até o artefato final com as informações relevantes da análise. Como é possível analisar no fluxo, na primeira etapa é necessário ter os argumentos que foram explicados anteriormente para inserir como entrada na execução da ferramenta. O FindLeak analisa todos os caminhos dos projetos que foram passados e adiciona todos os arquivos com extensão .java em um arquivo .txt no caminho que foi especificado no argumento, em seguida o arquivo com as classes em Java passa por três análises, a primeira é para saber sobre vazamentos de recursos do tipo de Camera, a segunda é do tipo Cursor e por último do tipo Media Player. Ao ter todas as classes em Java dos projetos que foram passados para serem analisados, as análises de cada tipo de recurso faz uma varredura estática para saber se os recursos daquele determinado arquivo foi usado corretamente, caso não tenha sido usado conforme as regras, o caminho do arquivo é armazenado junto com o tipo de recurso que foi detectado. Ao final das três análises, os arquivos com possíveis vazamentos e suas informações são adicionados em um arquivo do formato em excel e salvo no caminho que foi passado como argumento, esse arquivo é o artefato de resultado final da execução da ferramenta.

Figura 3 – Arquitetura do FindLeak



Fonte: O autor (2023)

4.2 MODO DE USO

A implementação foi realizada na linguagem Java. Para facilitar a experiência do usuário, foi gerado um arquivo Java Archive que é mais conhecido como um arquivo JAR, que inclui as classes implementadas de forma compacta e facilita na execução. A aplicação do FindLeak contida neste arquivo pode ser executada de acordo com o comando mostrado no Código 15. A aplicação em formato JAR está disponível na seguinte URL: <https://github.com/RaquelMSantos/FindLeak>

Código Fonte 15 – Template do comando para executar o FindLeak

```
1 java -jar FindLeak.jar comando1 comando2 comando3
```

Baseado na arquitetura da nossa ferramenta, a única ação necessária do usuário é passar por linha de comando as entradas com as extensões necessárias para a ferramenta executar corretamente. No trecho de código apresentado anteriormente o *comando1* e *comando2* são em formato TXT e o *comando3* é no formato XLSX. Após os argumentos de entrada, o FindLeak inicializa realizando a leitura de cada parâmetro, em seguida extrai as informações dos projetos passados no *comando1* e percorre em todos os arquivos para filtrar apenas as classes que possuem a extensão em JAVA, após ter a lista com todas as classes, é adicionado em um arquivo criado no caminho passado no *comando2*. Em seguida cria um terceiro arquivo

que será salvo no *comando3*, esse último arquivo será o artefato final com as informações após o processamento da análise para verificar se possui vazamentos de recursos. Para o processamento final, o FindLeak seleciona cada classe que foi extraída e adiciona na análise dos tipos de classes de recursos que são: Camera, Cursor e Media Player. Cada tipo de análise possui sua característica para saber se a classe do projeto está de acordo com os padrões de uso correto do recurso, caso a classe não esteja utilizando corretamente, seu caminho e o tipo de classe de recurso é adicionado na planilha do excel e no fim de todas as análises, o artefato final é preenchido onde a primeira coluna representa o caminho da classe com possível vazamento e a segunda coluna é o tipo de classe que foi detectado o vazamento. Na Figura 4 possui um exemplo de saída do artefato final.

Figura 4 – Exemplo de saída do FindLeak

PATH	Type of Leak
/Users/Documents/Repositorios/AppLeak/repositorios/callmeter/CallMeter3G/src/main/java/de/ub0r/android/callmeter/ui/LogsFragment.java	Cursor
/Users/Documents/Repositorios/AppLeak/repositorios/ChatSecureAndroid/src/info/guardianproject/otr/app/im/app/AccountActivity.java	Cursor
/Users/Documents/Repositorios/AppLeak/repositorios/connectbot/app/src/main/java/org/connectbot/util/HostDatabase.java	Cursor
/Users/Documents/Repositorios/AppLeak/repositorios/k-9/app/core/src/main/java/com/fsck/k9/mailstore/LocalFolder.java	Cursor
/Users/Documents/Repositorios/DroidLeak/repositorios/android-bankdroid/app/src/main/java/com/liato/bankdroid/db/DBAdapter.java	Cursor
/Users/Documents/Repositorios/DroidLeak/repositorios/android/libraries/cyclestreets-track/src/main/java/net/cyclestreets/track/DbAdapter.java	Cursor
/Users/Documents/Repositorios/DroidLeak/repositorios/AnySoftKeyboard/ime/app/src/main/java/com/anysoftkeyboard/dictionaries/content/ContactsDictionary.java	Cursor
/Users/Documents/Repositorios/DroidLeak/repositorios/zxing/android/src/com/google/zxing/client/android/camera/open/OpenCameraInterface.java	Camera
/Users/Documents/Repositorios/DroidLeak/repositorios/open-gpstracker/studio/service/src/main/java/nl/sogeti/android/gpstracker/service/logger/LoggerNotification.java	Media Player

Fonte: O autor (2023)

Resumindo, o FindLeak é simples de ser usado e o seu processo de análise é totalmente automático. Dada uma lista de repositórios que precisam ser analisados, o usuário só precisa preencher as informações dos argumentos e executar a ferramenta para obter a planilha com os arquivos que possuem vazamentos de recursos.

4.3 COLETA DE REPOSITÓRIOS

Os repositórios selecionados para o experimento foram extraídos de três fontes distintas. A escolha dessas bases foi baseada em alguns banco de dados que foram definidos em trabalhos anteriores. Os banco de dados utilizados foram o DroidLeaks (LIU et al., 2019) e o AppLeak (RIGANELLI; MICUCCI; MARIANI, 2018). Estes banco de dados possuem um banco de repositórios de aplicações Android, então resolvemos aproveitar essas fontes para adicionar os projetos citados por eles ao nosso experimento. Primeiro extraímos os 15 repositórios do banco de dados do AppLeak e 22 repositórios do banco de dados do DroidLeaks, como a junção dessas duas fontes estava equivalente a uma quantidade pequena de repositórios para inicializarmos nosso experimento, então buscamos uma terceira opção. Para nossa terceira fonte de dados

escolhemos utilizar a API do GitHub. Para iniciar a seleção dos projetos disponibilizados nesta plataforma foram adicionados alguns filtros para extrair 1.000 repositórios, essa quantidade é o suporte máximo de repositórios extraídos com aplicações de filtros que a API do GitHub disponibiliza, então resolvemos fechar a quantidade de repositórios para analisar inicialmente dessa fonte de dados com esse valor.

A extração dos repositórios do GitHub API foi baseada em filtros para selecionar repositórios relevantes para o nosso experimento. O primeiro filtro foi ordenar os repositórios de acordo com a quantidade de números de estrelas que possuíam e que incluíam parte do projeto escrito na linguagem Java, em seguida adicionamos o filtro para os projetos que apresentavam a label aplicada para o desenvolvimento em Android. Com esses filtros pré-estabelecidos, selecionamos os 1.000 repositórios que possuíam essas características e eram ordenados dos que tinham mais estrelas para os que tinham uma quantidade menor. Depois de escolher esses repositórios com os filtros aplicados, desenvolvemos um script utilizando o Pydriller para analisar os projetos que possuíam a classe AndroidManifest.xml, pois dessa forma poderíamos validar apenas os projetos que fossem realmente voltados para o desenvolvimento Android que é o nosso alvo principal. Após rodar o script com o filtro da classe do AndroidManifest.xml, restaram apenas 929 repositórios do GitHub API para realizarmos o experimento final. Portanto, a quantidade de repositórios extraídos das três fontes totalizou uma amostra de 966 repositórios de projetos reais para darmos início ao nosso experimento utilizando o FindLeak.

4.4 CASOS DE TESTE

Antes de realizar experimentos em grande quantidade de aplicações Android, foi preparado um escopo com cenário pequeno para executar a ferramenta e observar seu comportamento olhando com detalhes na sua execução e investigando os possíveis cenários que poderiam ser apresentados.

De início foi criado um código base com exemplos de vazamentos de recursos para os tipos de Camera, Cursor e Media Player. Esses exemplos eram supervisionados e possuíam um controle para adicionar cenários que poderiam ocorrer vazamentos de diferentes formas e também para os casos do caminho feliz onde os recursos estavam sendo criados corretamentes sem apresentar vazamentos. Vários objetos do mesmo tipo de recurso foram criados e foram adicionadas várias probabilidades de vazamento de recursos entre eles, como por exemplo, quando tinham mais de um objeto criado no mesmo bloco de comandos e apenas um deles

estava sendo utilizado de forma correta, como pode ser visto no Código 16.

Código Fonte 16 – Exemplo de vazamento de recursos de objetos no mesmo bloco

```
1 private void startMediaPlayer() {
      MediaPlayer mediaPlayer = new MediaPlayer();
3     mediaPlayer.start();
      ...
5     MediaPlayer mp = new MediaPlayer();
      mp.start();
7     ...
      mediaPlayer.release();
9 }
```

Com esse misto de cenários foi possível observar como a ferramenta estava se comportando diante de várias maneiras.

Após rodar a ferramenta com alguns exemplos de códigos, inicializamos um novo teste com o código de uma classe que pertencia a uma aplicação real, esse código foi extraído de um dos repositórios citados pelo DroidLeaks e que apontava para um vazamento de recurso do tipo Cursor, esse código estava dentro do nosso escopo de experimento por isso optamos por essa escolha. Após a realização desses testes preliminares, concluímos que poderíamos começar um experimento mais amplo, pois a ferramenta teve um bom comportamento e foi executada como o esperado.

Para uma última etapa de experimento com repositórios de teste, foram escolhidos dois repositórios apontados pelo DroidLeaks, AnkiDroid (ANKIDROID, 2021) e FBReaderJ (PULTSIN, 2017). Essa seleção ocorreu porque ambos os repositórios tinham uma boa quantidade de vazamentos de Cursor em classes diferentes, segundo o DroidLeaks. Quando aplicamos a ferramenta nesses dois repositórios, extraímos os caminhos das classes que apontavam possíveis vazamentos e analisamos manualmente para verificar se a ferramenta estava cobrindo os cenários corretamente. Para o primeiro repositório foram detectadas 13 classes que possuíam vazamentos e dentre essas classes apenas 2 não deveriam ser detectadas, mas foi observado que elas se tratavam de classes de teste, então para o primeiro repositório a ferramenta foi bem avaliada. Já no segundo repositório foi observado que eles possuíam cenários de criação de Cursor que era implementado por outra classe, ou seja, a classe que foi detectada como vazamento de recurso possuía uma classe de extensão para o objeto do tipo Cursor, mas ainda assim, esses objetos não estavam sendo fechados corretamente no que leva a vazamentos. Então, após a análise manual, concluímos que a ferramenta estaria apta para um experimento

mais robusto e com mais projetos reais.

4.5 LIÇÕES APRENDIDAS

Antes de realizarmos o experimento, passamos por uma etapa no qual encontramos alguns falsos positivos detectados pela ferramenta. Tivemos um total de 539 arquivos que a ferramenta detectou e estavam distribuídos da seguinte forma: 8 arquivos do tipo de vazamento de Camera, 498 arquivos voltados para o recurso de Cursor e 33 arquivos para o MediaPlayer. Após a extração dos arquivos com possíveis vazamentos detectados pela ferramenta, inicializamos uma análise manual de alguns arquivos..

A análise manual constituiu em validar primeiro os grupos que tinham uma quantidade de arquivos menor. Dessa forma o grupo do vazamento do tipo Camera foi o primeiro a ser analisado e foi possível observar que 50% do total de arquivos detectados eram falsos positivos equivalentes a 4 arquivos. Seguimos a análise para o segundo grupo do MediaPlayer e também tivemos alguns arquivos apontados como falsos positivos que foram no total de 13 arquivos, equivalentes a aproximadamente 39%. Para o terceiro grupo do Cursor como foram detectados muitos arquivos, então utilizamos o cálculo de 95% de confiança para 5% de margem de erro utilizando a calculadora (QUALTRICS, 2020) que resultou em 217 arquivos para serem analisados manualmente. Infelizmente encontramos também arquivos apontados como falsos positivos que resultaram no total de 24 arquivos. Como nos três grupos existiam falsos positivos, foi realizada uma análise mais detalhada para entender o que estava ocasionando esses falsos positivos e conseguimos observar que eles estavam seguindo um padrão para todos os grupos, alguns arquivos estavam utilizando os recursos e fechando o mesmo objeto mais de uma vez, então a ferramenta não estava conseguindo detectar que esse fechamento estava acontecendo corretamente. Dessa maneira, foi gerada uma nova versão da ferramenta que cobria esse cenário de erro e antes de rodar novamente todos os projetos, realizamos alguns testes com alguns arquivos que constaram como falsos positivos para verificar se o problema tinha sido de fato solucionado. Após a correção da ferramenta, conseguimos rodar de fato o experimento e conseguimos ter um bom resultado que será detalhado a seguir.

4.6 EXPERIMENTO

Antes de dar início ao experimento, foi criado um script para extrair os links dos projetos que foram coletados e adicionar o link de cada um em uma planilha para facilitar o controle das análises. Esse script foi rodado apenas nos repositórios da API do GitHub, pois os projetos provenientes do AppLeak e DroidLeaks já possuíam os links de fácil acesso dos projetos para serem baixados.

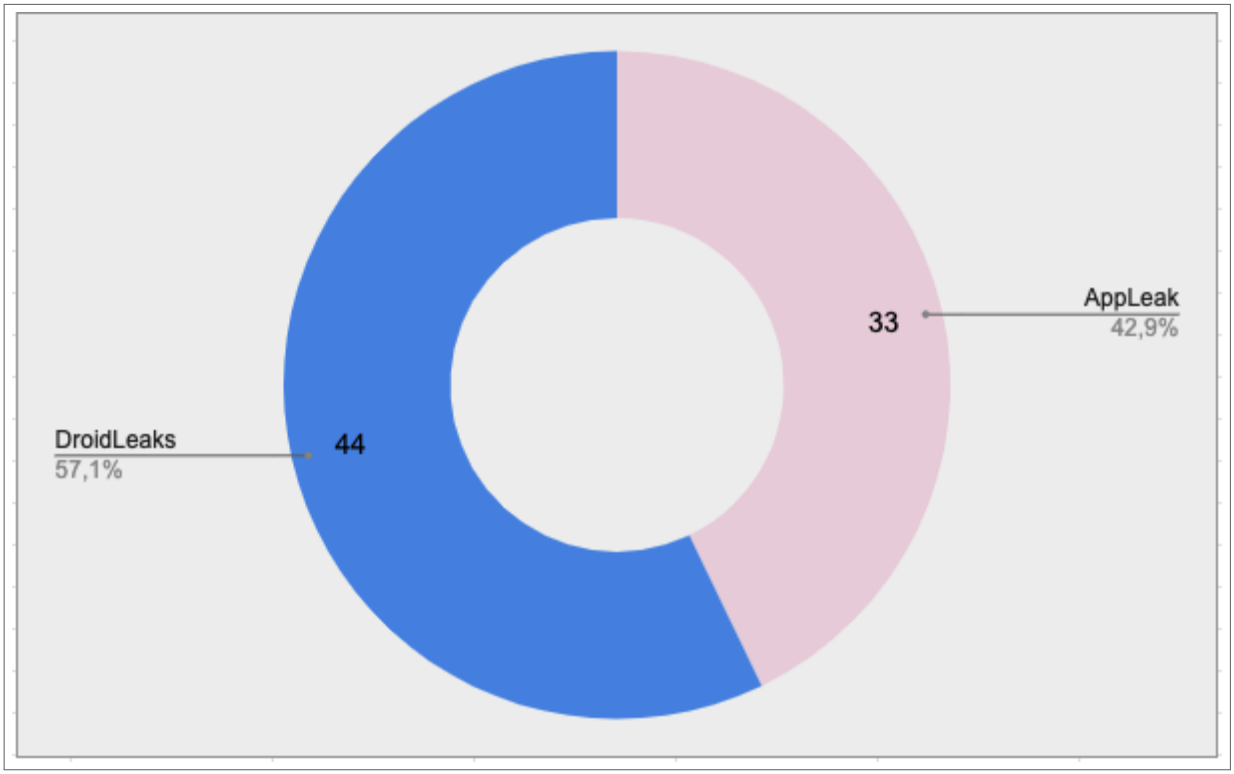
O experimento dos repositórios aplicados na ferramenta se inicializou com os 929 repositórios que foram extraídos do GitHub, esses projetos foram baixados localmente e para cada projeto, coletamos o caminho local onde estava armazenado e adicionamos cada caminho em um arquivo no formato .txt para que pudesse ser passado como parâmetro para a ferramenta FindLeak. Após realizarmos a conclusão da população do arquivo .txt, preparamos os argumentos necessários para executar a ferramenta corretamente e incluimos o arquivo criado como um dos parâmetros. Quando o arquivo em excel foi criado pela ferramenta, conseguimos analisar a quantidade de arquivos que foram detectados com possíveis vazamentos de recursos e para cada arquivo tinha o tipo de recurso identificado e o caminho da classe detectada. Separamos esses arquivos por categorias e essas categorias foram os tipos de classes de recursos que o FindLeak analisa, ou seja, separamos em Camera, Cursor e MediaPlayer. Ao total tivemos 453 arquivos que a ferramenta detectou e estavam distribuídos da seguinte forma: 4 arquivos do tipo de vazamento de Camera, 429 arquivos voltados para o recurso de Cursor, porém 36 desses arquivos eram de testes e 20 arquivos para o MediaPlayer. Após a extração dos arquivos com possíveis vazamentos detectados pela ferramenta, inicializamos uma análise manual de alguns arquivos para chegar a uma conclusão mais concreta sobre como o FindLeak estava se comportando.

A análise manual constituiu em validar primeiro os grupos que tinham uma quantidade de arquivos menor. Dessa forma o grupo do vazamento do tipo Camera foi o primeiro a ser analisado e o MediaPlayer o segundo. Para o primeiro e segundo grupo que possuíam uma quantidade de 4 e 20 arquivos, respectivamente, conseguimos verificar que todos tiveram sucesso na detecção do vazamento sem nenhum arquivo apontado como falso positivo, concluindo que 4 arquivos possuíam vazamentos do tipo de Camera e 20 arquivos do vazamento do tipo MediaPlayer. Já para o terceiro grupo, precisamos gerar novamente um cálculo para verificar quantos arquivos iriam ser analisados manualmente, tivemos um total de 429 arquivos detectados e observamos que 36 eram arquivos de testes e eles foram eliminados

do grupo, então aplicamos o cálculo de 95% de confiança para 5% de margem de erro utilizando a calculadora (QUALTRICS, 2020) para os 393 arquivos e resultou em 195 arquivos para serem analisados manualmente. Quando foi realizada a análise manual, percebemos que 33 arquivos foram apontados como falsos positivos, pois estavam fechando os recursos através da chamada de um método que realiza essa ação, então a ferramenta não conseguiu identificar quais recursos estavam sendo utilizados corretamente. Dessa maneira restaram 162 arquivos que tiveram sucesso. Portanto, para o nosso experimento, dos 929 repositórios extraídos do GitHub, tivemos um total de 417 arquivos de projetos reais que foram detectados com possíveis vazamentos de recursos.

Na segunda rodada para aplicar os repositórios na ferramenta, foram utilizados os projetos das fontes de dados do AppLeak e DroidLeaks. Por possuírem uma quantidade pequena de repositórios, foi realizada a junção deles e feitos todos os procedimentos necessários que são pré requisitos para preparar e rodar o FindLeak. Na base de dados do AppLeak foram encontrados 15 repositórios e no DroidLeaks um total de 22 repositórios. Com a junção temos 37 projetos reais que foram baixados localmente e passados como parâmetros diante dos requisitos do FindLeak para rodar corretamente. Ao rodar os projetos na ferramenta, foi gerado um total de 35 arquivos da base do AppLeak com possíveis vazamentos de recursos e 44 arquivos do DroidLeaks. Para os 35 arquivos, apenas 33 foram para o processo da análise manual, pois 2 se tratavam de arquivos de teste, então foi realizado uma análise manual dos 33 arquivos do AppLeak e 44 arquivos do DroidLeaks, totalizando 77 arquivos como mostra na figura a 5 equivalência para as duas bases.

Figura 5 – Arquivos detectatos pelo AppLeak e DroidLeaks



Fonte: O autor (2023)

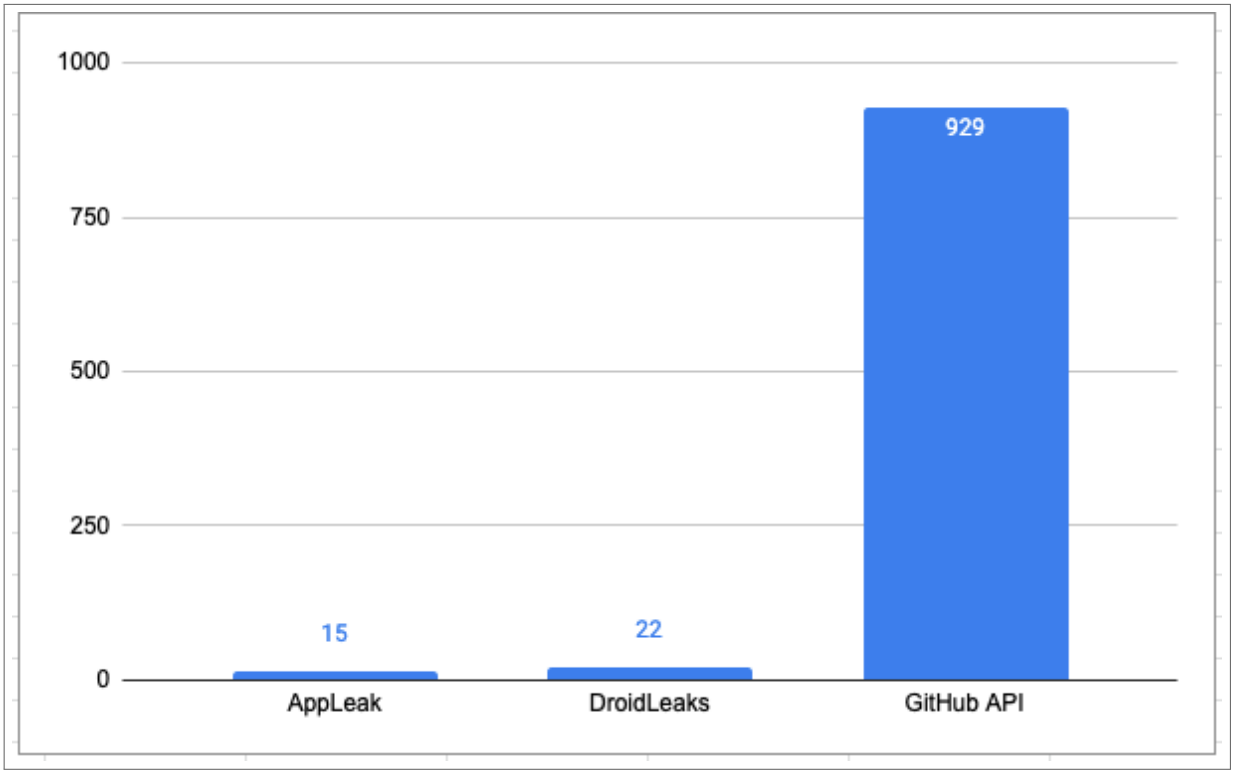
4.7 RESULTADOS

Esta seção apresenta os dados que foram coletados do experimento aplicado na ferramenta FindLeak em projetos reais que foram selecionados previamente. Foram escolhidos 966 repositórios de três fonte de dados para realizar o experimento, a representação da quantidade de repositórios por cada fonte de dados pode ser vista na Figura 6 e desses 966 repositórios, após utilizar a ferramenta FindLeak tivemos 494 arquivos detectados com possíveis vazamentos de recursos como é possível visualizar na Figura 7 com seus respectivas base de dados.

O FindLeak é voltado para três classes de recursos e então dividimos esses arquivos identificados em três categorias: Camera, Cursor e MediaPlayer. Como foram realizados os processos de execução da ferramenta em três bases de dados diferentes, então iremos realizar a primeira análise dos resultados para o primeiro grupo que foi executado pelo FindLeak que é o GitHub API.

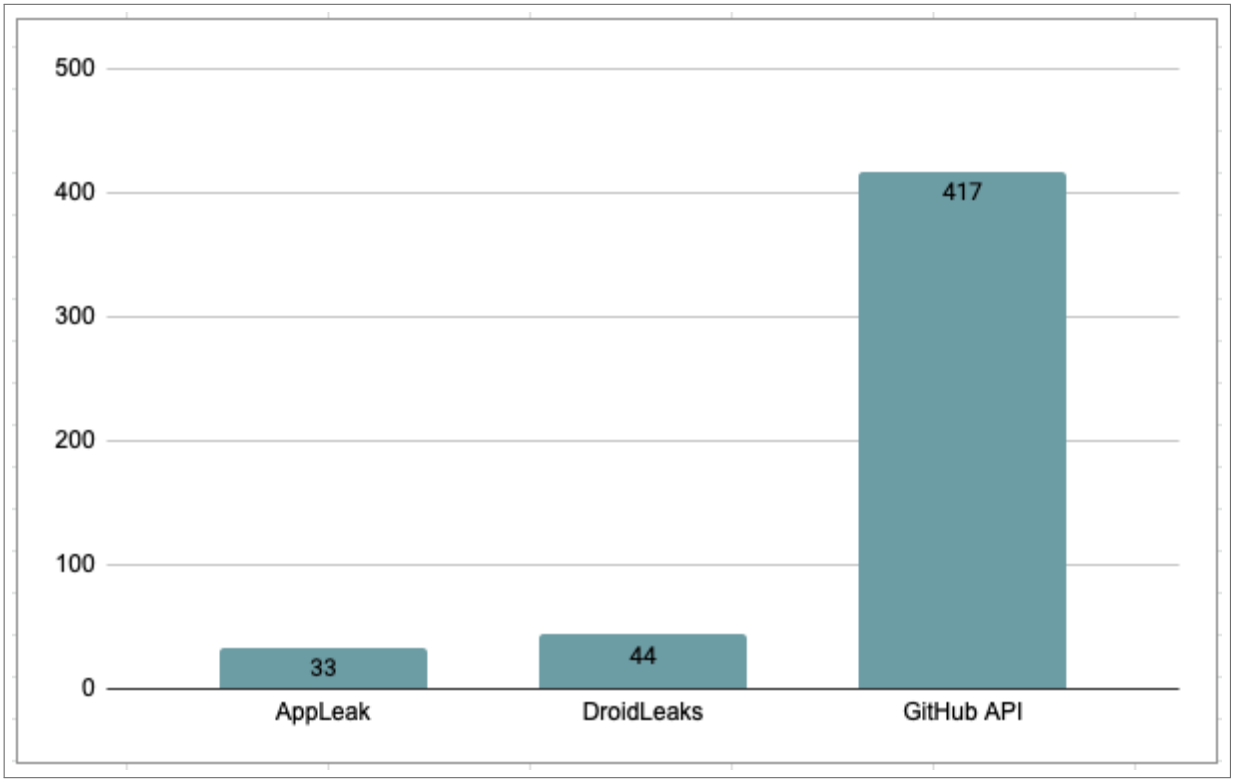
A figura 8 mostra a divisão dos arquivos de acordo com a categoria na base de dados do GitHub. Os arquivos ficaram distribuídos da maneira que 4 arquivos foram voltados para classe do tipo de Camera, 20 arquivos para o tipo MediaPlayer e 393 arquivos para o Cursor,

Figura 6 – Ditribuição de repositórios das bases de dados



Fonte: O autor (2023)

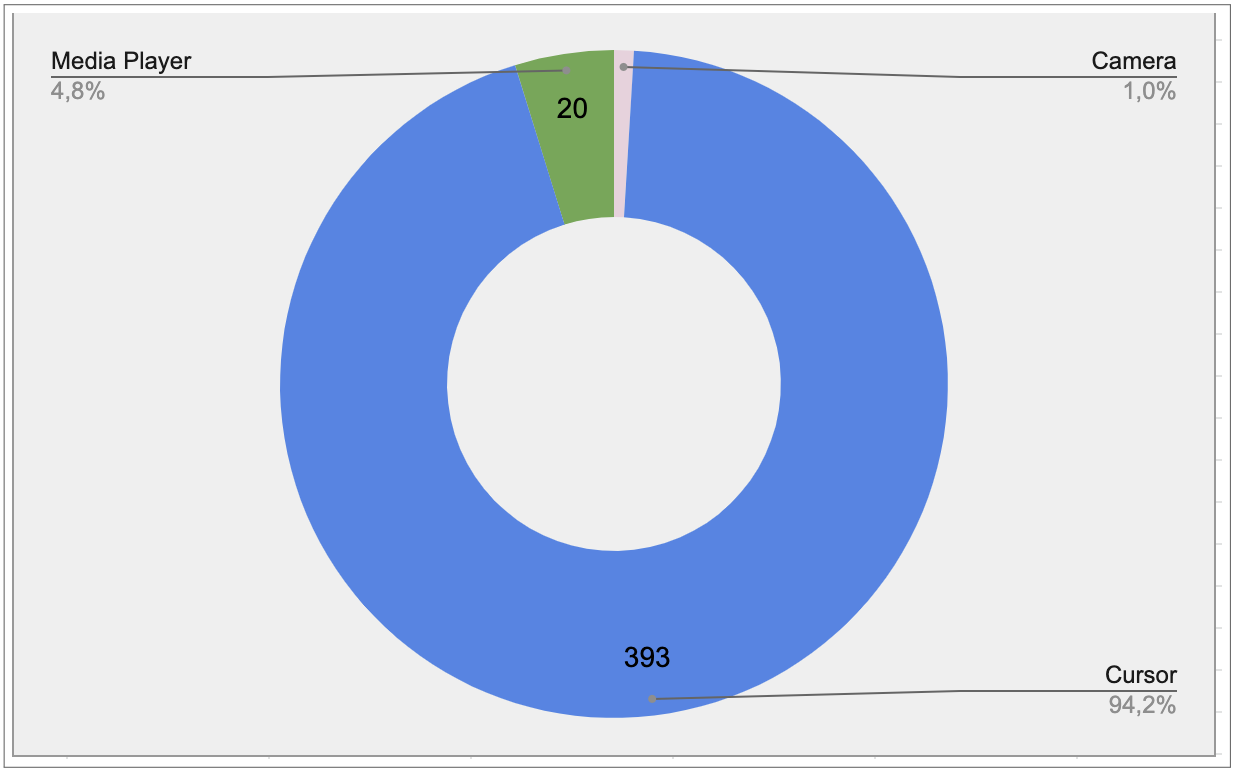
Figura 7 – Total de arquivos detectados



Fonte: O autor (2023)

representando do total dos arquivos detectados em 1%, 4,8% e 94,2%, respectivamente.

Figura 8 – Arquivos detectados no GitHub API

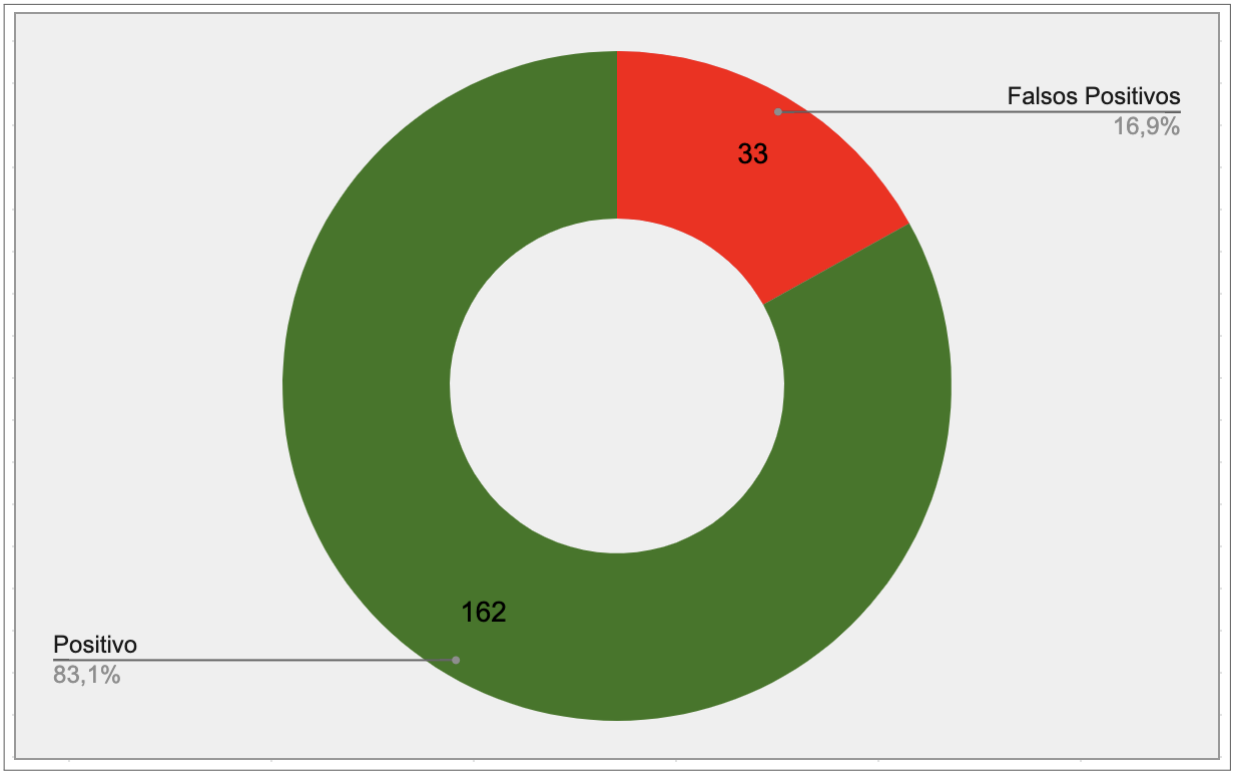


Fonte: O autor (2023)

Após a análise manual realizada nos arquivos divididos em três categorias, chegamos a conclusão que os 4 arquivos com vazamento do tipo Camera e os 20 arquivos com vazamentos do tipo MediaPlayer, tiveram 100% de resultado positivo diante da ferramenta, pois não ocorreu nenhum tipo de falso positivo. Para a terceira categoria voltada para a classe do tipo Cursor que obteve a maior quantidade dos resultados de arquivos detectados com o total de 393 arquivos, onde 195 passaram por uma análise manual. A partir dos 195 arquivos analisados manualmente, 33 tiveram resultados como falsos positivos e 162 arquivos analisados como positivos diante da ferramenta. Dessa forma, a ferramenta apresentou 83,1% com resultados positivos e essa distribuição pode ser analisada na figura 9.

Com uma análise mais detalhada do motivo de ter ocorrido falsos positivos, conseguimos observar que existiam cenários em que a classe possuía uma função que recebia como parâmetro o objeto e realizava o fechamento do recurso, então todos os objetos que estavam sendo criados faziam a chamada dessa função única para realizar o fechamento. Para resolução desses falsos positivos, o FindLeak precisaria ter o controle para detectar uma função central que realiza o fechamento de recursos e que essa função pode ser chamada em vários locais do código, mas

Figura 9 – Vazamento de Cursor dos arquivos do GitHub API



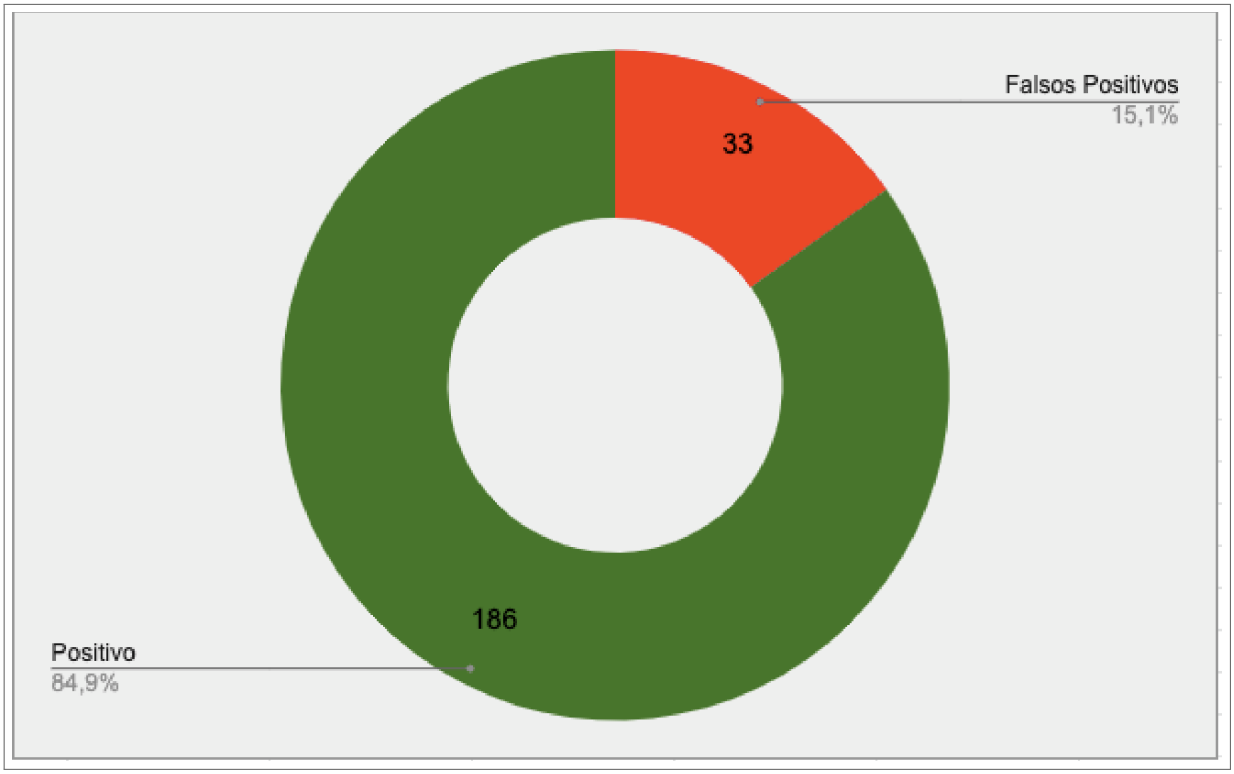
Fonte: O autor (2023)

o FindLeak não possui esse monitoramento, então obteve uma quantidade razoável de falsos positivos.

De modo geral, do total de 219 arquivos que foram analisados manualmente, tivemos um resultado de 186 arquivos com resultados positivos diante da ferramenta. Na figura 10 podemos ver a distribuição do resultado e concluiu que mais de 80% dos arquivos foram analisados positivamente.

Para a base de dados do AppLeak tivemos o aproveitamento de 100% dos arquivos com vazamento do tipo Cursor e dos 33 arquivos analisados manualmente obtivemos um resultado positivo para a ferramenta com 100% dos arquivos que foram detectados, após a análise manual observamos que todos foram com resultados positivos para o FindLeak sem apresentar falsos positivos. E para nosso último grupo do DroidLeaks, tivemos uma distribuição dos três tipos de classes que ficaram repartidos em 1 arquivo com o tipo de Camera, 3 arquivos com o tipo do MediaPlayer e 40 arquivos para o tipo de Cursor, onde podemos ver a distribuição sendo representada na figura 11. Para esse grupo, foi realizada a análise manual para verificar se apresentava algum tipo de falsos positivos, após a análise verificamos que todos os 44 arquivos que foram detectados, apresentaram resultados positivos para a ferramenta e não

Figura 10 – Resultado Análise Manual do GitHub API

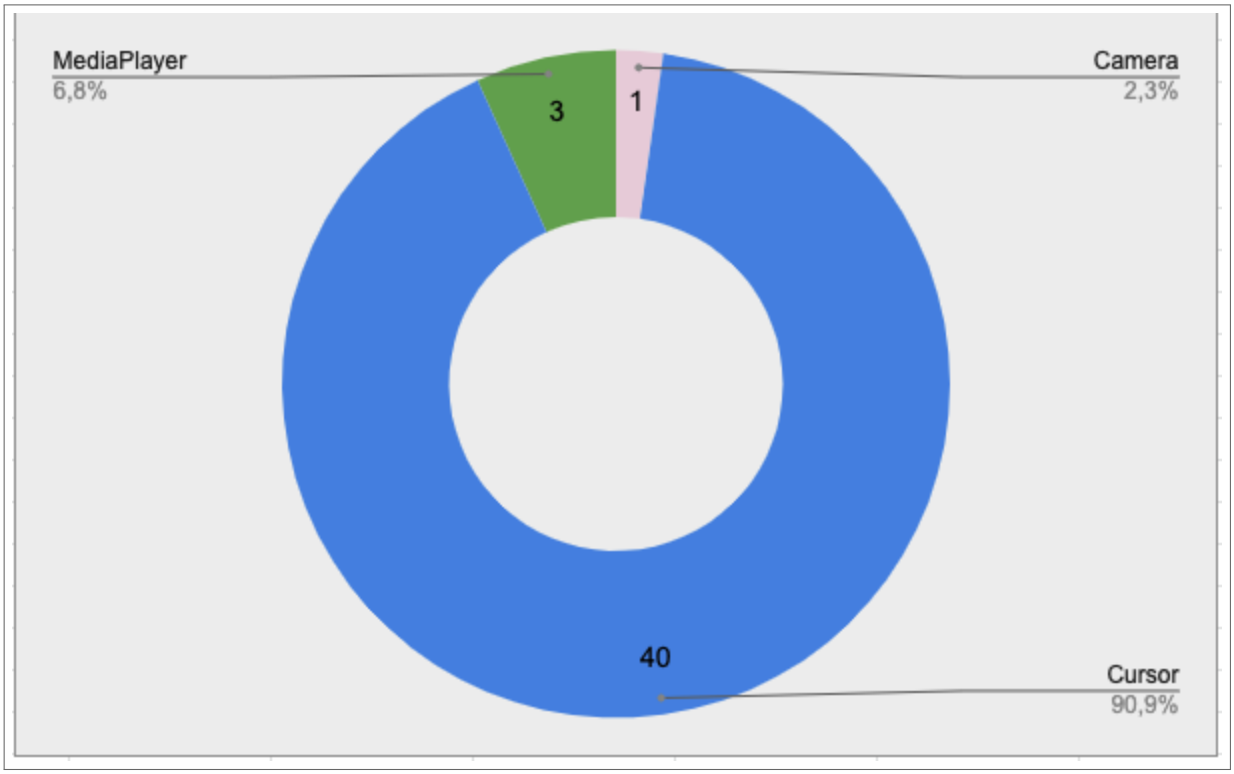


Fonte: O autor (2023)

apresentaram nenhum tipo de falsos positivos.

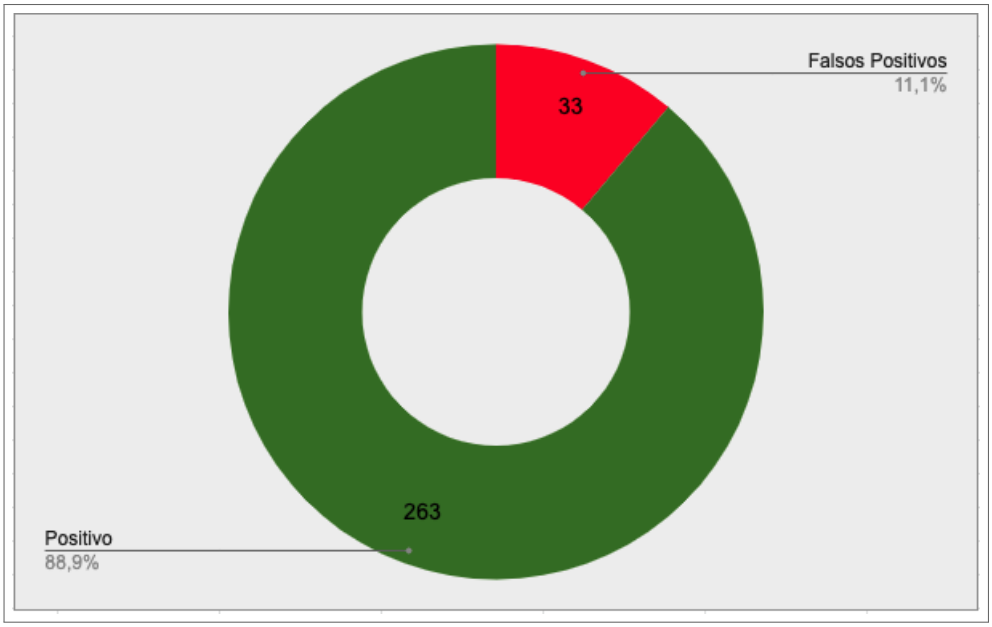
O resultado da ferramenta FindLeak aplicado em 966 projetos de aplicações reais, apresentou um número positivo por obter muitos arquivos detectados de forma positiva e podemos alcançar essa conclusão diante das análises manuais que foram realizadas nos 296 arquivos detectados com possíveis vazamentos de recursos e que deram-se origem das três fonte de base de dados que utilizamos, obtendo representação por 219 arquivos do GitHub API, 33 arquivos do AppLeak e 44 arquivos do DroidLeaks. Desses 296 arquivos, como podemos ver na Figura 12, na análise manual tivemos um resultado de 88,9% que tiveram resultados positivos e apenas 11,1% apresentou como falsos positivos que foram dos arquivos analisados do grupo do GitHub API. Ressaltando que os arquivos detectados como falsos positivos estavam tendo um padrão no qual o FindLeak não dava cobertura, como foi explicado na seção anterior. E para a classificação dos arquivos detectados baseados nas classes de recursos analisadas pelo FindLeak, podemos observar que o tipo Cursor foi o mais destacado em todas as fontes de dados, seguido pelo MediaPlayer e Camera, como podemos ver na Figura 13.

Figura 11 – Arquivos detectados no DroidLeaks



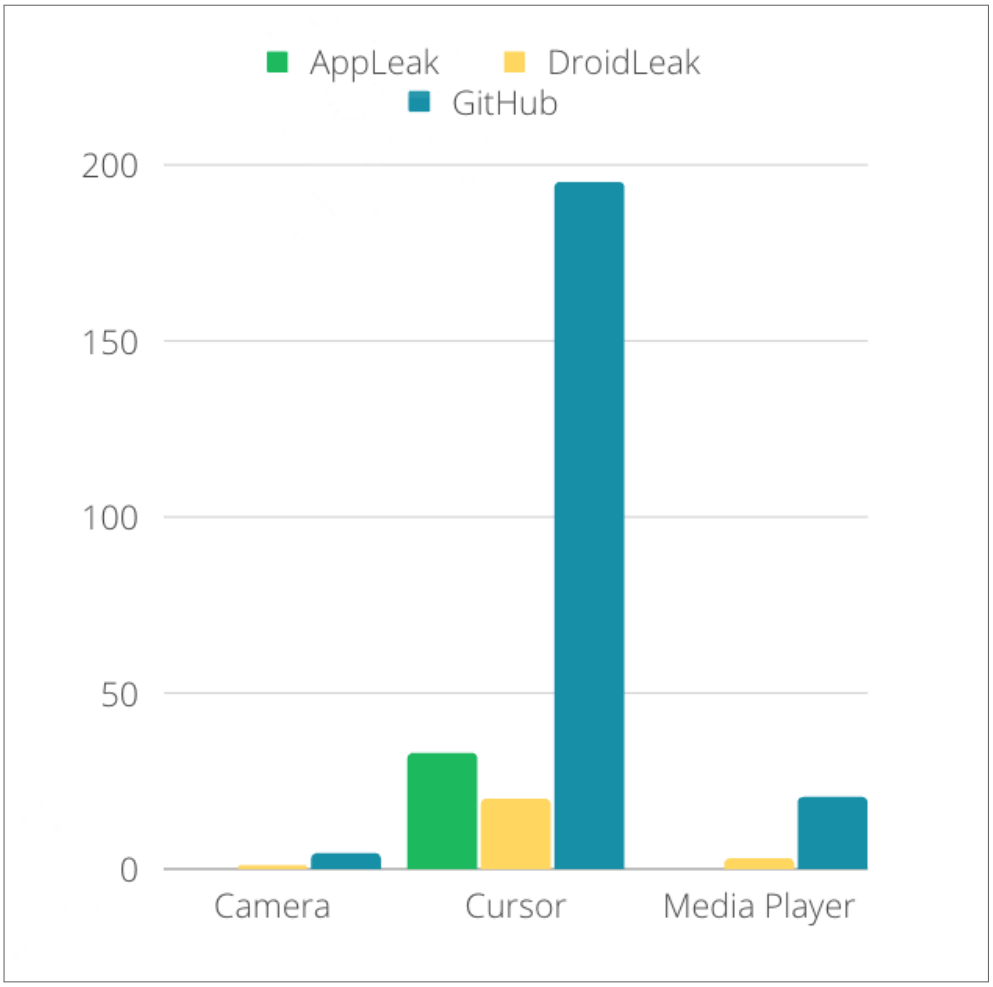
Fonte: O autor (2023)

Figura 12 – Resultado das análises manuais



Fonte: O autor (2023)

Figura 13 – Total das classes de recursos baseadas nas fontes de dados



Fonte: O autor (2023)

5 CONSIDERAÇÕES FINAIS

Neste trabalho realizamos estudos sobre vazamentos de recursos para entender como estavam sendo abordados e tratados pelos desenvolvedores de aplicações Android. Para este fim, iniciamos um estudo em trabalhos que foram relatados com uma análise um pouco mais explanada para entender como estavam sendo propostas as técnicas para ajudar aos desenvolvedores em relação aos vazamentos de recursos. Estudamos algumas pesquisas e analisamos algumas ferramentas para observar os comportamentos, mas não foram bem sucedidos nessa etapa, pois algumas ferramentas citadas não estavam disponíveis e outras não estavam funcionando.

Este estudo objetivou criar uma ferramenta que fosse possível realizar uma análise estática no código de aplicações Android para detectar vazamentos de recursos. O FindLeak é uma ferramenta desenvolvida para detectar vazamentos de recursos em algumas classes, como Camera, Cursor e MediaPlayer. Dado o projeto passado como entrada para ser analisado pelo FindLeak, a ferramenta analisa as classes que são desenvolvidas em Java para realizar a análise estática e detectar se possui vazamentos de recursos, disponibilizando para o desenvolvedor as informações necessárias para identificar onde o vazamento foi detectado.

Com base no experimento realizado em aplicações reais extraídas de três base de dados diferentes e com alguns filtros aplicados para selecionar aplicativos relevantes para o nosso estudo, os arquivos que foram identificados com vazamentos de recursos e alguns que foram analisados manualmente, retrata que o FindLeak pode ser utilizado para facilitar aos desenvolvedores em detectar vazamentos de recursos em suas aplicações e ter informações que facilitem na resolução dos vazamentos para deixar a aplicação com um desempenho melhor.

5.1 LIMITAÇÕES E AMEAÇAS À VALIDADE

Este trabalho apresenta algumas ameaças à validade e elas serão discutidas brevemente a seguir.

- Validade interna: Os tipos de projetos selecionados para avaliação da ferramenta podem ser considerados uma ameaça à validade, pois pode ter ocorrido dos projetos escolhidos serem mais propensos a conter vazamentos de recursos. Para mitigar essa ameaça, seria importante realizar uma seleção dos tipos de projetos com mais complexidades. O número de pesquisadores participantes deste trabalho também é considerado uma

ameaça, sabendo que uma ampliação na quantidade de pesquisadores poderia oferecer uma ampla variedade de opiniões nas etapas de análise e execução do estudo.

- Validade externa: Ao realizar os resultados obtidos neste estudo, podemos considerar uma possível ameaça externa referente aos projetos reais analisados, como não podemos concluir que as amostras representam todas as estratégias de desenvolvimento de aplicações Android e a diversidade de diferentes desenvolvedores e práticas de desenvolvimento, não pode ser generalizada para um conjunto mais amplo. Para abordar essa ameaça de validade externa, seria necessário expandir a amostra de projetos avaliados considerando um amplo perfil de desenvolvedores, pois isso ajudaria a aumentar a generalização dos resultados e garantir que o trabalho seja aplicado a uma gama mais ampla de aplicações Android. Além disso, aumentar o número de colaboração com outros pesquisadores para fortalecer a validade externa do trabalho.

- Validade da construção: Uma ameaça à validade de construção pode ocorrer se as regras utilizadas na análise estática não forem sensíveis o suficientes para detectar alguns falsos positivos ou falsos negativos que ocorram com frequência. Além disso, pode haver casos em que a ferramenta não consiga detectar vazamentos específicos que podem ocorrer em diferentes cenários ou contextos, como foi comentado no decorrer dos resultados obtidos. Essa ameaça já foi reconhecida ao longo do desenvolvimento do trabalho e a medida para realizar uma validação rigorosa é envolver comparação dos resultados obtidos pela ferramenta com um conjunto de casos de vazamentos de recursos conhecidos ou com avaliações manuais realizadas por especialistas da área.

- Validade da conclusão: Para os resultados obtidos neste estudo, uma ameaça à validade está relacionada ao fato de que os resultados se limitam a uma amostra específica de projetos reais e pode não capturar a diversidade de cenários e contextos em que as aplicações Android são desenvolvidas. Dessa forma, o recomendado seria discutir os resultados e reconhecer as limitações das amostras e explorar amostras mais diversas e representativas, incluindo estudos adicionais de projetos de diferentes tamanhos, domínio e características.

5.2 TRABALHOS FUTUROS

Com base nos experimentos e análise da ferramenta, podemos identificar alguns pontos para um trabalho futuro. Este trabalho não explorou muitas classes de recursos existentes em Aplicações Android, apenas três classes foram abordadas, para um trabalho futuro pode adicionar mais opções de classes de recursos que podem causar vazamentos em aplicativos. Outro ponto é que a ferramenta deste trabalho apenas realiza a análise estática em códigos desenvolvidos na linguagem Java, hoje em dia existem aplicações desenvolvidas na linguagem Kotlin, então seria uma possível proposta para incrementar e aumentar o escopo para que mais aplicações tenham o requisitos para serem analisadas. Por último, é a possibilidade de ter uma análise estática mais sofisticada para conseguir identificar quando os vazamentos são fechados a partir da chamada de uma única função criada pelo desenvolvedor para realizar essa ação, na nossa análise manual conseguimos identificar esse ponto que não foi coberto pelo FindLeak.

REFERÊNCIAS

ALMENARA, I. *Cerca de 1 bilhão de dispositivos Android foram ativados em 2021*. 2022. Disponível em: <https://canaltech.com.br/apps/cerca-de-1-bilhao-de-dispositivos-android-foram-ativados-em-2021-216251/>. Acesso em: 01 nov. 2022.

AMALFITANO, D.; RICCIO, V.; TRAMONTANA, P.; FASOLINO, A. R. Do memories haunt you? an automated black box testing approach for detecting memory leaks in android apps. *IEEE Access*, v. 8, p. 12217–12231, 2020.

ANALYTICS, S. *Strategy Analytics: Half the World Owns a Smartphone*. 2021. Disponível em: <https://news.strategyanalytics.com/press-releases/press-release-details/2021/Strategy-Analytics-Half-the-World-Owns-a-Smartphone/default.aspx>. Acesso em: 01 nov. 2022.

ANKIDROID. *Anki-Android*. 2021. Disponível em: <https://github.com/ankidroid/Anki-Android>. Acesso em: 11 mar. 2019.

AYEWAH, N.; PUGH, W.; HOVEMEYER, D.; MORGENTHALER, J. D.; PENIX, J. Using static analysis to find bugs. *IEEE Software*, v. 25, n. 5, p. 22–29, 2008.

BANERJEE, A.; CHONG, L. K.; BALLABRIGA, C.; ROYCHOUDHURY, A. Energypatch: Repairing resource leaks to improve energy-efficiency of android apps. *IEEE Transactions on Software Engineering*, v. 44, n. 5, p. 470–490, 2018.

DEVELOPERS, G. *Guias do desenvolvedor*. 2022. Disponível em: <https://developer.android.com/>. Acesso em: 17 jan. 2023.

GUO, C.; ZHANG, J.; YAN, J.; ZHANG, Z.; ZHANG, Y. Characterizing and detecting resource leaks in android applications. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2013. p. 389–398.

JETBRAINS. *O que é a análise de código estático?* 2000. Disponível em: <https://www.jetbrains.com/pt-br/teamcity/ci-cd-guide/concepts/static-code-analysis/>. Acesso em: 03 nov. 2022.

LIU, Y.; WANG, J.; WEI, L.; XU, C.; CHEUNG, S.-C.; WU, T.; YAN, J.; ZHANG, J. Droidleaks: a comprehensive database of resource leaks in android apps. *Empir Software Eng 24*, p. 3435–3483, 2019. Disponível em: <https://doi.org/10.1007/s10664-019-09715-8>.

PULTSIN, N. *FBReaderJ*. 2017. Disponível em: <https://github.com/geometer/FBReaderJ>. Acesso em: 11 mar. 2019.

QUALTRICS. *Sample size calculator*. 2020. Disponível em: <https://www.qualtrics.com/blog/calculating-sample-size/>. Acesso em: 15 de nov. 2022.

RIGANELLI, O.; MICUCCI, D.; MARIANI, L. From source code to test cases: A comprehensive benchmark for resource leak detection in android apps. *John Wiley and Sons Ltd*, p. 540–548, 2018. Disponível em: <https://doi.org/10.1002/spe.2672>.

TEIXEIRA, E. Ferramenta de análise de código para detecção de vulnerabilidades. 2007. Disponível em: <http://www.di.fc.ul.pt/~nuno/THESIS/EmanuelTeixeira_master07.pdf>.

WU, T.; LIU, J.; DENG, X.; YAN, J.; ZHANG, J. Relda2: An effective static analysis tool for resource leak detection in android apps. In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2016. p. 762–767.

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