



Universidade Federal de Pernambuco
Centro de Informática
Curso de Bacharelado em Engenharia da Computação

# AndroidDriller:

# Uma ferramenta de mineração de repositórios Android

Aluno: Alberto Vital Santos de Sousa
Orientador: Leopoldo Motta Teixeira

Recife, junho de 2018

Universidade Federal de Pernambuco
Centro de Informática
Curso de Bacharelado em Engenharia da Computação

# AndroidDriller:
# Uma ferramenta de mineração de repositórios Android

Monografia apresentada ao Centro de Informática (CIn)
da Universidade Federal de Pernambuco (UFPE), como requisito
parcial para conclusão do Curso de Engenharia da Computação,
orientada pelo professor Leopoldo Motta Teixeira.

Recife, junho de 2018

# Agradecimentos

São muitos nomes para que eu me lembre de todos os que merecem aqui, mas gostaria de agradecer àqueles que de alguma forma contribuíram não só para este trabalho como também para a minha formação como pessoa. Em especial posso citar meus pais Antonio Vital Alves de Sousa e Maria Auxiliadora Santos Vital Alves de Sousa, que sempre me apoiaram e me incentivaram nas minhas decisões e a sempre seguir com meus estudos. Também agradeço ao meu orientador Leopoldo Motta Teixeira, pelos suporte e ensinamentos que me proporcionou durante este período. De uma maneira geral agradeço também a todos os meus colegas de graduação e do laboratório Oki Brasil, que estiveram ao meu lado durante os momentos difíceis da graduação e também nos de descontração do dia a dia.

# Resumo

Utilizando ferramentas capazes de minerar dados sobre repositórios, pesquisadores de engenharia de software têm obtido um melhor conhecimento sobre o processo de desenvolvimento de software, em geral. Este trabalho propõe uma ferramenta capaz de minerar repositórios fazendo uso das características comuns de projetos Android para que se possa realizar uma investigação preliminar de como as mudanças acontecem no desenvolvimento de aplicações para a plataforma. Por fim, são apresentados os resultados da utilização da ferramenta sobre um conjunto de mais de 1100 repositórios Android livres e de código aberto retirados do catálogo do F-Droid.

**Palavras-chave**: Android, Mineração de Repositórios, Sistema de Controle de Versão, Git

# Abstract

Software Engineering researchers have been using tools capable of mining data from repositories to get a better understanding about the software development process. This work proposes a tool for mining repositories that uses the common characteristics of Android Projects so we can perform a preliminary investigation about how changes occur in the platform apps development. In the end, we present the results of running the tool over a set of more than 1100 free and open source Android repositories taken from the F-Droid catalogue.



**Keywords**: Android, Repository Mining, Version Control System, Git

# Sumário

# 1 Introdução

Em engenharia de software, a análise de repositórios tem ajudado pesquisadores a obter um melhor conhecimento sobre o processo de desenvolvimento de um software [1], pois a análise de software consiste em extrair dados para que gerentes e engenheiros de software possam melhorar os processos de tomada de decisão e desenvolvimento de softwares [2]. Dessa forma, podemos definir como mineração de repositórios o processo de extração desses dados. Com isso, é possível predizer bugs e também analisar o padrão de desenvolvimento utilizado pelos colaboradores do projeto. Para tal fim, existem ferramentas capazes de minerar dados sobre repositórios, como por exemplo, o RepoDriller [3].

Ferramentas mais gerais, como o próprio RepoDriller, servem para analisar software em vários domínios. No desenvolvimento de aplicações Android [4], há diversas particularidades envolvidas. Por exemplo, o arquivo de manifesto, chamado AndroidManifest, é obrigatório para qualquer aplicação da plataforma e contém informações sobre os vários componentes do projeto. Assim, qualquer componente não declarado aqui também não fará parte da aplicação final, da mesma forma que permissões (mecanismo utilizado pela plataforma para que as aplicações possam ter acesso a certas funcionalidades do dispositivo) utilizadas no código precisam ser declaradas neste arquivo para que funcionem corretamente. Com isso, as principais alterações feitas no repositório são refletidas em modificações desse arquivo, o que pode ser utilizado para mineração de dados relativos ao histórico de alterações do projeto.

No estudo de repositórios Android, pesquisadores têm usado ferramentas capazes de extrair dados dos arquivos do tipo apk [5, 6, 7]. Para realizar estudos semelhantes, uma outra abordagem seria utilizar uma ferramenta capaz de minerar os repositórios das aplicações e extrair dados específicos levando em consideração não só o arquivo apk mas também as modificações dos arquivos de manifesto, por exemplo. Uma vez que o arquivo apk seria a versão final da aplicação, o histórico do manifesto pode revelar informações sobre o histórico da aplicação durante seu processo de desenvolvimento.

Com o intuito de testar a ferramenta implementada e também gerar dados para uma análise mais profunda, foram analisados 1195 repositórios presentes no F-Droid [8], cujos endereços do código fonte apontavam para o github. Desses 1195, 1138 foram minerados com sucesso.

## 1.1 Objetivos

Este trabalho tem por objetivo realizar uma investigação preliminar de como as mudanças acontecem no desenvolvimento de aplicações Android para criar ferramentas de suporte à evolução de aplicativos, e assim, implementar uma ferramenta capaz de minerar estes repositórios, aproveitando-se das características particulares de projetos Android.

# 2 Conceitos Básicos

Neste capítulo apresentamos conceitos essenciais para a compreensão do trabalho. Na Seção 2.1 apresentamos a plataforma Android e seus componentes. Na Seção 2.2 explicamos o que são sistemas de controle de versão, destacando o Git. Na seção seguinte comentamos sobre mineração de repositórios, e por fim, ainda na Seção 2.3, apresentamos o framework utilizado para desenvolver a ferramenta proposta.

## 2.1 Android

O sistema operacional Android pode ser definido como uma pilha de software de código aberto desenvolvida para sistemas móveis [9]. A partir desta pilha, programadores são capazes de desenvolver aplicações para dispositivos móveis sem se preocupar com detalhes específicos de hardware, pois a própria plataforma trata essas diferenças em sua API. O fato de Android ter seu código aberto, também permite ao desenvolvedor implementar sua própria versão dos módulos da plataforma caso deseje.

A pilha de software citada acima pode ser dividida em camadas, conforme a Figura 1

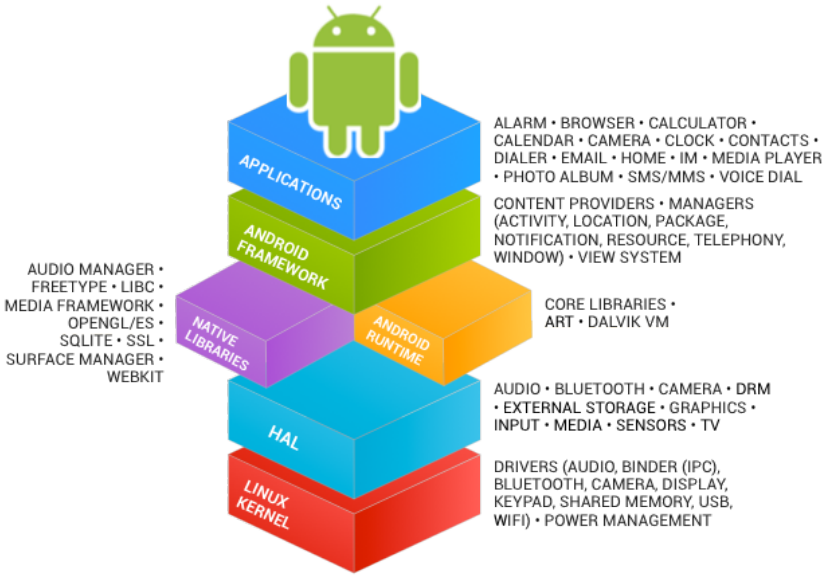


Figura 1: Pilha de software dividida em camadas [9]

A camada mais inferior, *Linux Kernel*, provê uma abstração do hardware para as camadas superiores, com os serviços mais genéricos de um sistema operacional. Na camada *HAL* ou *Hardware Abstraction Layer*, temos uma abstração maior do hardware, realizando o carregamento dos drivers específicos para os periféricos solicitados pelas camadas superiores. Nas camadas *Native Libraries* e *Android Runtime*, encontram-se as bibliotecas nativas e a máquina virtual que roda as aplicações. Na camada *Android Framework*, encontram-se os principais elementos providos pela plataforma para as aplicações, como por exemplo as notificações. Na camada superior, denominada *Applications* é onde estão as aplicações e as principais classes utilizadas pelos desenvolvedores para produzi-las.

Este trabalho propõe uma ferramenta para extrair dados relativos à camada *Applications*. Mais abaixo trataremos alguns dos principais elementos da plataforma e que são abordados pela ferramenta proposta.

### 2.1.1 AndroidManifest

É um arquivo XML obrigatório que contém informações sobre os componentes da aplicação. Precisa ser criado com o nome **`AndroidManifest.xml`**, dessa forma a plataforma é capaz de identificar, dentre outras coisas, qual parte do código implementa cada uma das entidades que compõem a aplicação. Assim, qualquer componente utilizado no código precisa ser declarado neste arquivo para que funcione corretamente. Analogamente, qualquer componente não declarado aqui também não fará parte da aplicação final. Um exemplo desse arquivo pode ser visto na Figura 2.

Nas seções abaixo, descreveremos os elementos da Figura 2 em mais detalhes.

### 2.1.2 Activity

É o componente que representa a interface visual de interação com o usuário. No arquivo de manifesto é representada pela tag **`activity`** e precisa definir o atributo **`name`**, que é utilizado para identificar a classe que a implementa.

Na Figura 2 podemos ver duas activities sendo declaradas, uma delas possui o atributo **`name`** definido como **`br.ufpe.cin.avss.ui.AnotherActivity`**.

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="br.ufpe.cin.avss">

    <permission android:name="br.ufpe.cin.avss.PERMISSAO_PODCAST"/>
    <uses-permission android:name="android.permission.READ_CONTACTS"/>

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">

        <activity android:name="br.ufpe.cin.avss.ui.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name="br.ufpe.cin.avss.ui.AnotherActivity" />

        <provider
            android:name="br.ufpe.cin.avss.db.PodcastProvider"
            android:authorities="br.ufpe.cin.avss.podcast.feed"
            android:permission="br.ufpe.cin.avss.PERMISSAO_PODCAST"
            android:enabled="true"
            android:exported="false" />

        <receiver android:name="br.ufpe.cin.avss.download.BackgroundReceiver">
            <intent-filter>
                <action android:name="android.intent.action.ON_BOOT_COMPLETE"/>
            </intent-filter>
        </receiver>

        <service android:name="br.ufpe.cin.avss.player.PodcastPlayer"/>
    </application>

</manifest>
```

Figura 2: Exemplo de arquivo AndroidManifest.xml

```xml
<activity android:name="br.ufpe.cin.avss.ui.AnotherActivity" />
```

Dessa forma, a plataforma espera encontrar no pacote **`br.ufpe.cin.avss.ui`** a implementação da classe **`AnotherActivity`** que representará uma das telas da aplicação.

### 2.1.3 Broadcast Receiver

A plataforma Android utiliza um sistema de eventos similar ao padrão *publish-subscribe*, onde a plataforma (ou outra aplicação) dispara um evento (*broadcast*) e as aplicações registradas para o evento recebem a mensagem correspondente. O

componente que recebe estes eventos é conhecido como *broadcast receiver*.

Ao iniciar o dispositivo, o sistema Android dispara um evento de *broadcast* chamado de **`ON_BOOT_COMPLETE`**. Para que a aplicação receba esse e outros eventos semelhantes é necessário implementar uma classe do tipo **`BroadcastReceiver`** e registrá-la no evento desejado. Para isso, declara-se no manifesto a tag **`receiver`**, conforme o trecho abaixo, que também pode ser visto na Figura 2 onde foi declarado um componente receiver que espera eventos do tipo citado anteriormente.

```
<receiver android:name="br.ufpe.cin.avss.download.BackgroundReceiver">
  <intent-filter>
    <action android:name="android.intent.action.ON_BOOT_COMPLETE"/>
  </intent-filter>
</receiver>
```

### 2.1.4 Content Provider

Componente responsável por prover uma interface de acesso para outras aplicações aos dados utilizados. Dessa forma, cria-se uma abstração da camada de dados que pode ser utilizada não só por outras aplicações como também pela própria aplicação, conectando os dados em um processo com o código em execução em outro.

No manifesto é representada pela tag **`provider`**. Na Figura 2 temos um exemplo que foi replicado no trecho abaixo, onde a exemplo da tag **`activity`**, o atributo **`name`** define a classe com a implementação correspondente.

```
<provider
    android:name="br.ufpe.cin.avss.db.PodcastProvider"
    android:authorities="br.ufpe.cin.avss.podcast.feed"
    android:permission="br.ufpe.cin.avss.PERMISSAO_PODCAST"
    android:enabled="true"
    android:exported="true" />
```

Ao declarar este elemento é necessário definir o atributo **`authorities`**, que seguindo a convenção de nomes em Java, identifica o provider declarado. Em geral, esse nome é baseado no package name da aplicação, que deve ser único dentre todas as aplicações da plataforma. Dessa forma, os providers declarados pelas aplicações podem ser exportados, pois, a plataforma irá identificá-los a

partir do atributo **authorities**.

O atributo **permission** define que para acessar os dados deste receiver, é necessário possuir a permissão **br.ufpe.cin.avss.PERMISSAO_PODCAST**, um elemento que será discutido mais à frente.

### 2.1.5 Service

Este componente não provê uma interface visual e roda em background. Dessa forma, é utilizado para implementar procedimentos que não necessitam que a aplicação esteja ativa no momento. Em geral, operações mais pesadas como downloads de arquivos e reprodução de áudio são realizadas por estes componentes, uma vez que sua execução não é finalizada mesmo quando o usuário sai da aplicação. No manifesto é declarado com a tag **service**, como pode ser visto no trecho abaixo retirado da Figura 2.

```
<service android:name="br.ufpe.cin.avss.player.PodcastPlayer"/>
```

### 2.1.6 Permission

O sistema de permissões da plataforma Android garante que certas ações só sejam executadas caso o usuário da aplicação conceda o privilégio necessário. As permissões podem ser classificadas em:

- Dangerous
- Normal
- Signature
- Signature or System

As permissões consideradas *dangerous* podem causar algum risco à privacidade do usuário ou ao funcionamento normal do dispositivo, por isso necessitam da validação do usuário. As *normal*, por não trazerem esses riscos, são concedidas automaticamente pelo sistema ao serem declaradas no manifesto.

Desde a versão 6.0 da plataforma Android, a forma como as permissões são solicitadas foi alterada. Nas versões anteriores, qualquer permissão *dangerous* é solicitada antes da instalação, e caso a aplicação precise ser atualizada, as permissões são solicitadas novamente. Caso o usuário não conceda alguma

permissão, a aplicação não é instalada nem atualizada. A partir da versão 6.0, as permissões *dangerous* são solicitadas em tempo de execução. Dessa forma, a aplicação pode ser instalada sem que o usuário tenha que conceder todos os privilégios solicitados. Assim, a permissão associada com a funcionalidade ativada é solicitada sempre que o usuário desejar utilizar a funcionalidade.

Dessa forma, caso uma aplicação deseje ler os contatos do usuário, por exemplo, é necessário declarar a tag **`uses-permission`** no manifesto com o atributo **`name`** de valor igual a **`"android.permission.READ_CONTACTS"`**, conforme pode ser visto na Figura 2.

```
<uses-permission android:name="android.permission.READ_CONTACTS"/>
```

Existe também a possibilidade da própria aplicação definir permissões de acesso ao seus conteúdos. Para isso, utiliza-se a tag **`permission`**. Conforme foi visto na Seção 2.1.4, o atributo **`permission`** pode ser utilizado para definir uma permissão de acesso ao conteúdo provido pelo componente. Dessa forma, pode-se notar que a aplicação definida no arquivo da Figura 2 não só define uma permissão própria, como também protege seu content provider com essa mesma permissão, de acordo com as tags do trecho abaixo.

```
<permission android:name="br.ufpe.cin.avss.PERMISSAO_PODCAST"/>
...
<provider
    android:name="br.ufpe.cin.avss.db.PodcastProvider"
    android:authorities="br.ufpe.cin.avss.podcast.feed"
    android:permission="br.ufpe.cin.avss.PERMISSAO_PODCAST"
    android:enabled="true"
    android:exported="false" />
```

## 2.2 Sistemas de controle de versão

São sistemas que controlam o versionamento de um software durante o estágio de desenvolvimento. Com estes sistemas é possível saber qual o estado exato do código fonte em um determinado período do processo de produção, pois a cada alteração realizada o sistema armazena quais arquivos e quais linhas foram modificadas naquele momento, além de também informar quem realizou as alterações.

Na próxima seção apresentamos o Git, um dos mais populares sistemas de controle de versão da atualidade.

### 2.2.1 Git

É um sistema de controle de versão descentralizado livre e de código aberto [10], onde cada desenvolvedor tem uma cópia completa do histórico do repositório. A cada conjunto de alterações registrada um *snapshot* do projeto denominado *commit* é armazenado. O git utiliza uma estrutura de grafos para organizar os commits em diversas linhas de desenvolvimento, cada uma sendo denominada *branch* [11].

Abaixo estão listados alguns dos principais comandos git, juntamente com uma breve explicação de seu funcionamento.

- `git init` Inicializa um repositório a partir de um diretório local.
- `git clone` Copia um repositório remoto em um diretório local.
- `git pull` Recupera as mudanças remotas para o repositório local.
- `git add` Adiciona as mudanças locais num conjunto para que possam ser incluídas no próximo commit.
- `git commit` Cria um commit com as mudanças selecionadas pelo comando git add.
- `git push` Atualiza o repositório remoto com as mudanças locais.
- `git branch` Cria uma nova linha de desenvolvimento
- `git merge` Unifica duas linhas de desenvolvimento em um novo commit que é denominado *merge commit*

Para exemplificar o funcionamento de um merge commit, suponha o seguinte grafo de commits de um repositório representado pela Figura 3.

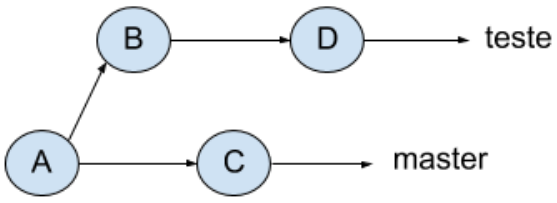


Figura 3: Grafo de commits com duas branches

Estando na branch master e executando o comando **`git merge teste`** cria-se o commit *E* na branch master e que unifica as modificações realizadas nas duas branches, resultando no grafo da Figura 4.

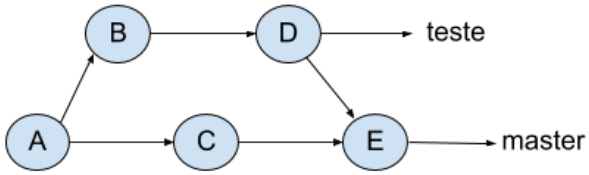


Figura 4: Grafo de commits após o merge commit 'E'.

## 2.3 Mineração de repositórios

A análise de software consiste em extrair dados para que gerentes e engenheiros de software possam melhorar o processo de desenvolvimento de software através do ganho e compartilhamento de ideias provenientes desses dados para uma melhor tomada de decisão [2]. Dessa forma, podemos definir como mineração de repositórios o processo de extração desses dados. Devido ao tamanho e a quantidade de repositórios utilizados durante seus estudos, pesquisadores têm utilizado ferramentas de mineração para automatizar esse processo.

Na próxima seção apresentamos uma ferramenta utilizada para minerar repositórios, e que foi utilizada como base para a ferramenta desenvolvida neste trabalho.

### 2.3.1 RepoDriller

RepoDriller é um framework de mineração de repositórios feito em Java [3]. Com esta ferramenta é possível extrair dados não só dos commits realizados (como por exemplo autor e mensagem) mas também dos arquivos contidos no repositório.

Para escrever um programa que minere um dado repositório, é preciso implementar a interface **`Study`** e no método **`execute`** criar um objeto de estudo

(uma instância da classe **RepositoryMining**) utilizando o padrão builder e informando além do repositório a ser minerado, as implementações da interface **CommitVisitor**. Para informar o repositório, utilizamos o método **in** que recebe uma implementação de **SCMRepository**, podendo ser tanto Git quanto SVN. Neste trabalho foi utilizado a implementação **GitRemoteRepository** que recebe uma URL referenciando o repositório remoto (ver Figura 5).

A interface **CommitVisitor** define o método **process** que é chamado pela API interna do RepoDriller a cada commit visitado. O método **process** recebe como parâmetros **SCMRepository**, **Commit** e **PersistenceMechanism**. O primeiro é um objeto representando o repositório estudado, podendo ser Git ou SVN. O segundo é uma representação do commit sendo visitado. E o terceiro é um objeto para tratar a persistência dos dados minerados ao longo dos vários commits visitados.

Ainda no objeto **RepositoryMining**, definimos o mecanismo de persistência a ser passado para cada visitor. Em geral, escolhe-se um arquivo CSV. A partir da versão 1.4 do RepoDriller, pode-se definir uma configuração de coleta, para evitar que dados desnecessários sejam carregados a cada commit. Para o nosso caso em que vamos focar em informações básicas do commit e nas modificações do arquivo de manifesto, foi acrescentada a chamada ao método **collect** como pode ser visto na Figura 5.

```
new RepositoryMining()
        .in(GitRemoteRepository.singleProject(repositoryPath))
        .through(range)
        .collect(new CollectConfiguration().basicOnly().sourceCode().diffs())
        .process(new ActivityAndroidVisitor(repoName),
                new CSVFile(outputPath + "activityDriller.csv"))
        .process(new ServiceAndroidVisitor(repoName),
                new CSVFile(outputPath + "serviceDriller.csv"))
        .process(new BroadcastReceiverAndroidVisitor(repoName),
                new CSVFile(outputPath + "broadcastReceiverDriller.csv"))
        .process(new ContentProviderAndroidVisitor(repoName),
                new CSVFile(outputPath + "contentProviderDriller.csv"))
        .process(new PermissionAndroidVisitor(repoName),
                new CSVFile(outputPath + "permissionDriller.csv"))
        .process(new UsesPermissionAndroidVisitor(repoName),
                new CSVFile(outputPath + "usesPermissionDriller.csv"))
        .mine();
```

Figura 5: Objeto **RepositoryMining** definido no método **execute** da classe **RepoStudy**.

# 3 AndroidDriller

Neste capítulo apresentamos a ferramenta desenvolvida ao longo deste trabalho. Na Seção 3.1 descrevemos o processo geral do que foi desenvolvido. Na Seção 3.2 detalhamos cada classe implementada. Em seguida, na Seção 3.3 comentamos sobre os repositórios escolhidos para validar e testar a implementação. Por fim, nas seções 3.4 e 3.5 explicamos um experimento onde foram extraídos dados de repositórios Android com o AndroidDriller e os resultados obtidos, respectivamente.

## 3.1 Metodologia

Foi criada uma ferramenta Java que faz uso da API provida pelo RepoDriller para percorrer os commits do repositório. Neste projeto foram implementadas as classes capazes de colher dados sobre os componentes específicos de Android e geração de relatórios em arquivos CSV [1]. Para visualização dos dados, foi implementado um programa na linguagem Python [12] capaz de produzir gráficos a partir dos arquivos CSV gerados pela ferramenta. Este fluxo pode ser descrito pela Figura 6.

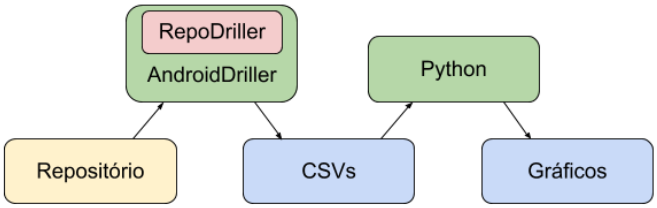


Figura 6: Fluxo dos dados de um repositório até os gráficos gerados.

## 3.2 Implementação

Primeiramente, vamos apresentar o modelo utilizado para representar as entidades da plataforma Android, que foi fortemente baseado no modelo utilizado por Trindade [13]. Como pode ser visto na Figura 7, temos uma classe que representa o arquivo de manifesto e cada elemento do XML foi mapeado em uma classe Java. Para representar os componentes, foi utilizado o padrão de herança entre classes. Dessa forma, a classe `Component` contém os atributos comuns aos componentes

[1] *Comma-separated values*, arquivo onde os valores são separados por vírgulas.

Android, que foram representados cada um como uma classe filha de **`Component`**.

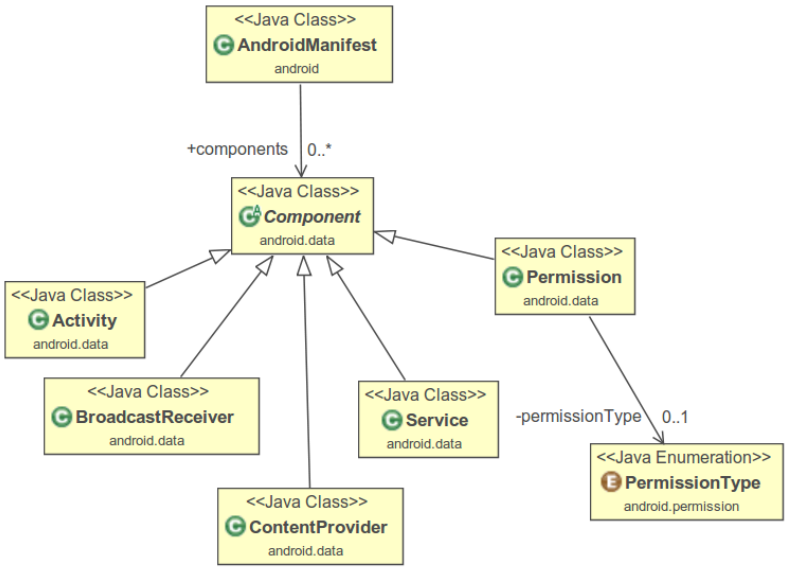


Figura 7: Diagrama das classes de modelo da plataforma Android.

A classe **`Component`** também define o método **`isModificationOf`** que recebe outro componente e verifica se um é apenas uma modificação do outro ou se são componentes totalmente distintos. Para fazer essa verificação foi utilizado o atributo **`name`** como identificador do componente, ou seja, se dois componentes têm o mesmo valor de name mas valores diferentes para outros atributos, eles representam o mesmo componente apenas modificado. Por exemplo, suponha que o componente A tenha o valor de **`name`** definido como **`"NomeA"`** e o atributo **`enabled`** como **`true`**, o componente B tem **`name`** igual a **`"NomeB"`** e **`enabled`** igual a **`true`**, enquanto o componente C tem **`name`** igual a **`"NomeA"`** e **`enabled`** igual a **`false`**. Com isso, a chamada **`A.isModificationOf(B)`** retorna falso enquanto a chamada **`A.isModificationOf(C)`** retorna verdadeiro. Note que, caso exista um elemento D com **`name`** igual a **`"NomeB"`** e **`enabled`** igual a **`true`**, a chamada **`D.isModificationOf(B)`** retorna falso. Dessa forma, cada uma das subclasses de **`Component`** pode sobrescrever este método e acrescentar seus atributos específicos. Este método será útil mais à frente quando comentarmos sobre a classe **`ComponentDiff`**.

Por conveniência, os atributos e métodos foram omitidos dos diagramas aqui apresentados.

Seguindo uma arquitetura semelhante à das classes de modelo, foram implementadas classes de diff, conforme a Figura 8. Essas classes tratam o diff específico de cada componente.

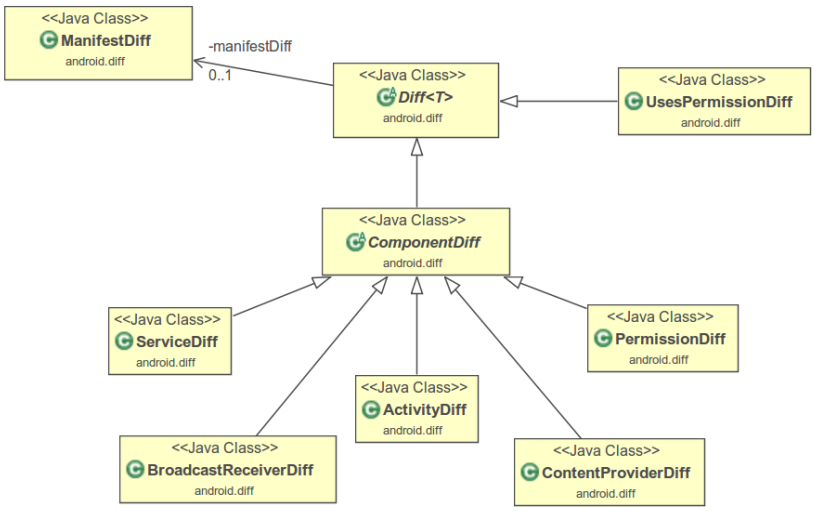


Figura 8: Diagrama das classes que tratam as alterações de cada commit.

A classe **`ManifestDiff`** recebe uma lista de modificações registradas em cada commit, dessa lista, são filtradas as que se referem a qualquer arquivo com a denominação **`AndroidManifest.xml`**. A partir das alterações nos manifestos, são criadas duas listas de versões dos arquivos, uma lista de arquivos anteriores e outra posteriores ao commit. De posse dessas versões utilizamos a classe **`AndroidManifestParser`** para criar duas listas de instâncias da classe **`AndroidManifest`** definida pelo modelo. As instâncias de **`AndroidManifest`** são então passadas para a classe abstrata parametrizada **`Diff<T>`** .

De uma forma geral, em **`Diff`** são criadas duas listas de componentes, uma representando os componentes anteriores ao commit e outra os posteriores. A partir disso, mais três listas são criadas, uma com as adições, outra com as remoções e a terceira com os componentes modificados. Para reusar o código definido nesta classe, o parâmetro **`T`** define qual componente será analisado pela subclasse de **`Diff`**, pois esta define o método abstrato **`getElementsList`** que recebe uma instância de **`AndroidManifest`** e retorna uma lista de objetos do tipo **`T`**. A classe também define outro método abstrato, **`isModification`**, que recebe um elemento **`T`** e verifica se ele foi modificado. Dessa forma, as subclasses só precisam implementar estes dois métodos para que se possa obter informações específicas

de cada componente desejado. Por exemplo, a classe **UsesPermissionDiff** estende **Diff** conforme pode ser visto na Figura 9, onde o parâmetro **T** foi definido como **String**, o método **getElementsList** apenas retorna as permissões utilizadas no manifesto e uma vez que o elemento uses-permission só define um atributo do tipo **String**, o método **isModification** retorna sempre **false**.

```
public class UsesPermissionDiff extends Diff<String>{

    public UsesPermissionDiff(ManifestDiff manifestDiff) {
        super(manifestDiff);
    }

    @Override
    protected List<String> getElementsList(AndroidManifest manifest) {
        return manifest.permissions;
    }

    @Override
    protected boolean isModification(String component) {
        return false;
    }

}
```

Figura 9: Implementação da classe **UsesPermissionDiff**.

Note que na Figura 8 a classe **UsesPermissionDiff** herda diretamente da classe **Diff**, enquanto que os outros componentes herdam da classe **ComponentDiff**. Isto ocorre porque o elemento uses-permission no nosso modelo só define um atributo do tipo **String**, o atributo **name**. Por isso, ele é representado apenas como uma **String** com o valor desse atributo. Por esta mesma razão este componente não está representado no diagrama de modelo da Figura 7. Enquanto isso, a classe abstrata **ComponentDiff** define o parâmetro **T** para ser o tipo **Component**, implementando o método **getElementsList** para retornar uma lista de **Component** que é recuperada pelo método abstrato **getComponents**, e também implementa o método **isModification** para retornar o **isModificationOf** definido pela instância de **Component**, conforme vimos anteriormente. Assim as subclasses podem implementar apenas o método **getComponents** de forma mais específica.

Como visto anteriormente, para percorrer os commits de um dado repositório, é preciso implementar uma interface provida pelo RepoDriller. Para que se pudesse utilizar a mesma implementação para os vários repositórios a serem minerados, foi criada a classe **RepoStudy**, que implementa a interface **Study**, conforme Figura 10.

Essa classe recebe a URL do repositório a ser estudado e a partir disso inicializa o objeto **RepositoryMining** com o repositório a ser minerado e registra os visitors específicos de repositórios Android que serão mais detalhados à frente.

A exemplo do que foi visto na Seção 2.3.1, para que possamos analisar cada commit, foram implementados visitors que extraem dados a respeito de cada componente se-

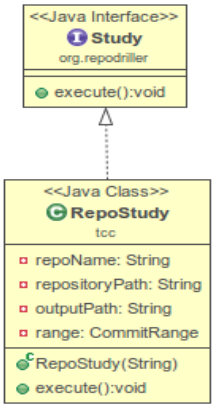


Figura 10: Diagrama de classe do objeto RepoStudy.

paradamente. Esses visitors seguem a estrutura definida na Figura 11, onde um visitor abstrato implementa a interface **`CommitVisitor`** definida pelo RepoDriller, repassando a chamada do método **`process`** (que recebe um **`Commit`**) da interface para o método abstrato **`androidProcess`** (que recebe um **`AndroidCommit`**) conforme Figura 12. Dessa forma, o **`Commit`** provido pelo framework é convertido em uma instância de **`AndroidCommit`**. Assim, a classe **`DiffAndroidVisitor`** implementa o **`método androidProcess`** e define o método abstrato **`getDiff`**, repassando o **`AndroidCommit`** para as subclasses retornarem cada uma a instância da subclasse de **`Diff`** do componente correspondente. Ainda no método **`androidProcess`**, são registrados os dados recuperados pela subclasse de **`Diff`**, retornada pelo método **`getDiff`**, no arquivo CSV definido pelo visitor utilizando a instância de **`PersistenceMechanism`** passada. Essa escrita no CSV só ocorre nos commits em que são identificadas mudanças nos arquivos de manifesto, com isso, caso um repositório possua 100 commits mas apenas 10 alteraram os manifestos, os arquivos CSV gerados terão 10 linhas cada.

A classe **`AndroidCommit`** estende a classe **`Commit`** do RepoDriller, adicionando informações específicas Android (ver Figura 12). No seu construtor, esta classe recebe o **`Commit`** a ser estendido e o repositório a qual ele pertence, parâmetros que são utilizados para recuperar os manifestos do commit e a lista de modificações. A partir disso, é possível criar uma instância de **`ManifestDiff`**, que é utilizada como base para criação das instâncias das subclasses de **`Diff`** que serão recuperadas pelos visitors específicos, como visto anteriormente.

Para que tudo isto funcione, foi implementada a classe **`MyStudy`**, que no seu método main, lê o arquivo de entrada encontrado na pasta **`androidDriller/input`** com o nome **`repoURLs.in`**. Este arquivo deve conter uma lista de URLs de repositórios Android remotos do github. Para cada repositório, a classe **`MyStudy`** cria uma instância de

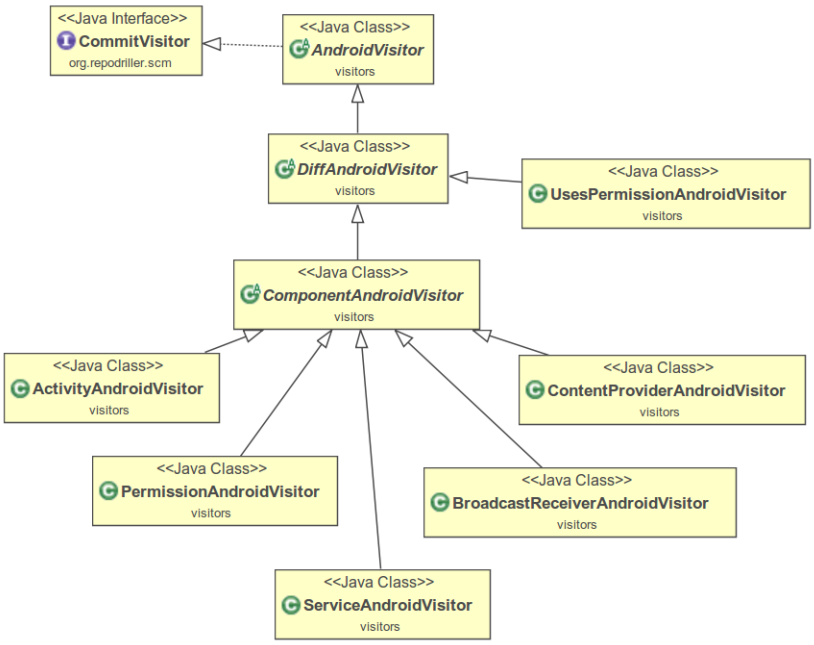


Figura 11: Diagrama das classes que implementam os visitors de cada componente.

**`RepoStudy`** passando a URL do repositório como parâmetro. Na classe **`RepoStudy`**, é criado o diretório de saída, **`androidDriller/output`**, no qual são criadas uma pasta para cada repositório minerado, onde se encontram os arquivos CSV gerados.

## 3.3 Repositórios de Teste

No intuito de gerar um cenário pequeno, com poucos commits, mas que abrangesse os casos mais genéricos de commits que alterassem o manifesto de um aplicação, foi criado um repositório público no github apenas com 2 arquivos **`AndroidManifest.xml`** em diretórios diferentes. Neste repositório foram realizadas alterações nos dois arquivos e em branchs separadas que foram posteriormente agregadas à master e deletadas. Este repositório se encontra na URL `https://github.com/betosousa/fooAndroidManifest.git`.

Para validar a implementação descrita anteriormente, foram escolhidos 6 repositórios de código livre presentes no F-Droid [8], um catálogo de aplicações livre e de código aberto para a plataforma Android. Juntamente com o repositório citado acima, foram listados 7 repositórios para formar uma base de testes. Esses repositórios foram escolhidos aleatoriamente dentro do catálogo, levando em conta apenas a quantidade de commits, onde tentamos escolher repositórios em faixas variadas de quantidade de commits totais. Estes

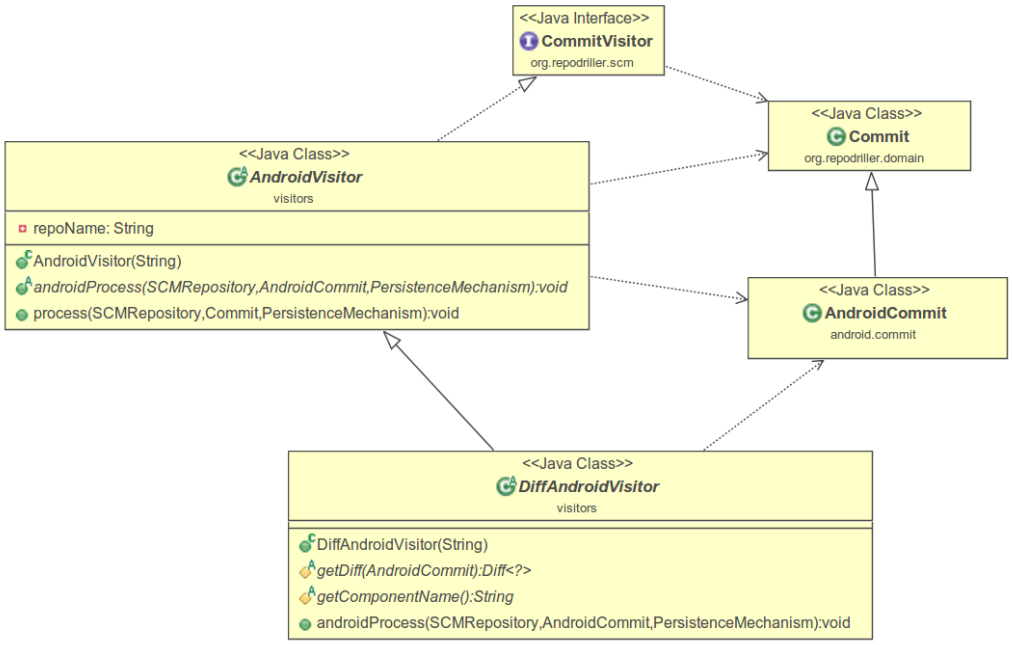


Figura 12: Diagrama do encapsulamento do método process e da classe Commit.

repositórios estão listados na Tabela 1.

| URL | Total | Manifesto |
|---|---|---|
| https://github.com/dozingcat/AsciiCam | 56 | 28 |
| https://github.com/uberspot/2048-android | 70 | 21 |
| https://github.com/naman14/Timber | 597 | 43 |
| https://github.com/Telegram-FOSS-Team/Telegram-FOSS | 704 | 65 |
| https://github.com/jackpal/Android-Terminal-Emulator.git | 1088 | 139 |
| https://github.com/tejado/Authorizer.git | 1304 | 112 |
| https://github.com/betosousa/fooAndroidManifest.git | 29 | 24 |

Tabela 1: Repositórios utilizados para validação, com quantidade total de commits e commits com alteração do manifesto.

Dessa forma foi possível avaliar os resultados obtidos diretamente com o histórico dos repositórios. Assim, foi possível identificar que, em repositórios com commits de merge, as modificações replicadas em mais de uma branch eram contadas de forma duplicada pela ferramenta. Com isso, ao tentar calcular o total de componentes no repositório com a simples diferença entre total somado ao longo do repositório com o total removido, observamos valores diferentes do total de componentes presentes no repositório. Para resolver isso, foi acrescentado o total de componentes por commit, que é calculado na

classe **`Diff`** e representa o total de componentes na lista de elementos posteriores ao commit.

## 3.4 Experimento

Para testar mais a fundo a aplicação implementada e também gerar dados para uma análise mais profunda, foram analisados 1195 repositórios presentes no F-Droid [8], cujos endereços do código fonte apontavam para o github.

Alguns repositórios apresentaram falhas durante o experimento. Foram 51 por não estarem acessíveis (ou por exigirem credenciais para clonar ou por não mais existirem) e 6 por terem arquivos de diff muito grandes, que ainda provocaram um estouro de memória e a interrupção da execução da ferramenta. Todos os 57 foram desconsiderados da análise final.

## 3.5 Resultados

Por fim, foram gerados 6 arquivos CSV para cada um dos 1138 repositórios minerados com sucesso, totalizando 49116 commits registrados com alterações no arquivo de manifesto. Nas tabelas a seguir, vemos um resumo dos dados coletados pela ferramenta, que foram coletados utilizando um script Python que percorreu cada um dos arquivos CSV gerados. Na Tabela 2 temos a coluna *Adicionados*, que representa a quantidade de componentes adicionados ao longo dos históricos dos repositórios; as colunas *Removidos* e *Modificados* que representam a quantidade de removidos e modificados; além do total registrado no último manifesto do repositório, na coluna *Total*.

| Componente | Adicionados | Removidos | Modificados | Total |
|---|---|---|---|---|
| Activity | 20024 | 9591 | 18419 | 6711 |
| Broadcast Receiver | 3252 | 1542 | 1025 | 1112 |
| Content Provider | 868 | 293 | 197 | 349 |
| Service | 3155 | 1491 | 724 | 1095 |
| Permission | 386 | 169 | 9 | 98 |
| Uses Permission | 7253 | 1289 | 0 | 4105 |

Tabela 2: Resumo dos dados coletados sobre os componentes.

Na Tabela 3 temos a quantidade de commits que alteraram o componente, a quantidade de commits que adicionaram pelo menos 1 componente, a quantidade de commits que removeram pelo menos 1 componente e a quantidade de commits que modificaram pelo menos 1 componente, representadas nas colunas *Commits, Adições, Remoções* e *Modificações* respectivamente.

| Componente | Commits | Adições | Remoções | Modificações |
|---|---|---|---|---|
| Activity | 16803 | 9130 | 3914 | 7929 |
| Broadcast Receiver | 2970 | 2091 | 909 | 681 |
| Content Provider | 973 | 751 | 259 | 153 |
| Service | 3019 | 2251 | 1057 | 522 |
| Permission | 287 | 231 | 103 | 6 |
| Uses Permission | 4287 | 3635 | 770 | 0 |

Tabela 3: Resumo dos dados coletados sobre os commits por componente.

Com esses dados, podemos verificar que os componentes mais alterados são as activities, com mais de 16800 commits, quase 4 vezes mais que o segundo componente, o uses permission, com cerca de 4300 commits. Também podemos verificar que, a cada 3 commits que alteram o manifesto, 1 altera activities, enquanto que as permissions só são alteradas a cada 171 commits no manifesto. Note ainda que a quantidade de commits que adicionam pelo menos 1 componente é sempre maior que a quantidade de commits que removem, enquanto que apenas para as activities, a quantidade de commits de modificação é maior que a de remoção. Essas métricas servem para demonstrar que tipo de informações podemos extrair dos repositórios Android utilizando a ferramenta proposta.

Para ilustrar os resultados possíveis de serem obtidos com a ferramenta, o repositório `https://github.com/M66B/NetGuard` foi escolhido aleatoriamente dentre os 1138 minerados. Neste repositório foram registrados 3235 commits dos quais 95 alteraram o arquivo de manifesto.

Na Figura 13 temos o histórico das activities neste repositório. No eixo horizontal temos as datas dos commits que alteraram o manifesto e no eixo vertical temos a quantidade de activities. O total está representado pela linha laranja, a quantidade adicionada no commit está representada nas barras verdes, a removida nas barras vermelhas e as modificadas nas azuis.

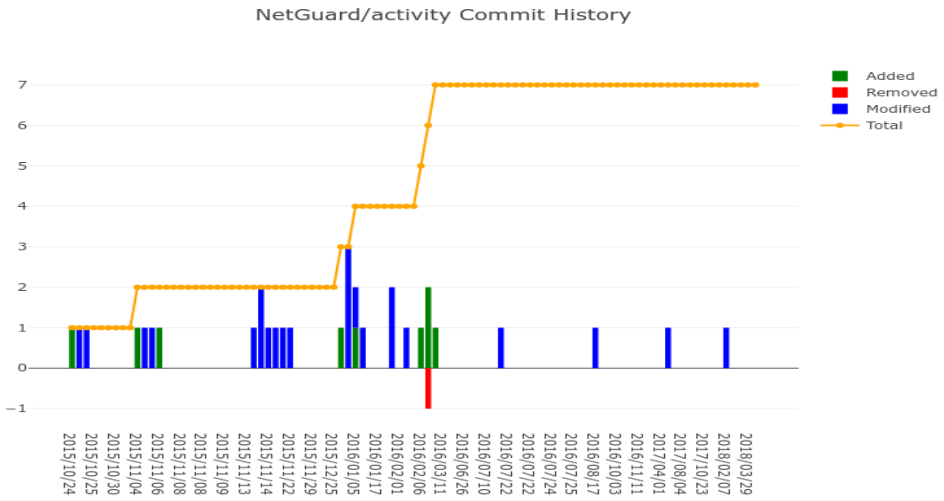


Figura 13: Histórico das activities do repositório NetGuard

Observamos que, até a metade da linha do tempo exibida na Figura 13, há uma quantidade razoável de mudanças em activities, principalmente em modificações. A partir daí, existem mudanças esporádicas apenas modificando activities existentes, o que indica uma possível estabilidade do projeto.

Nas Figuras de 14 a 18 podemos ver os históricos dos outros componentes para o mesmo repositório.

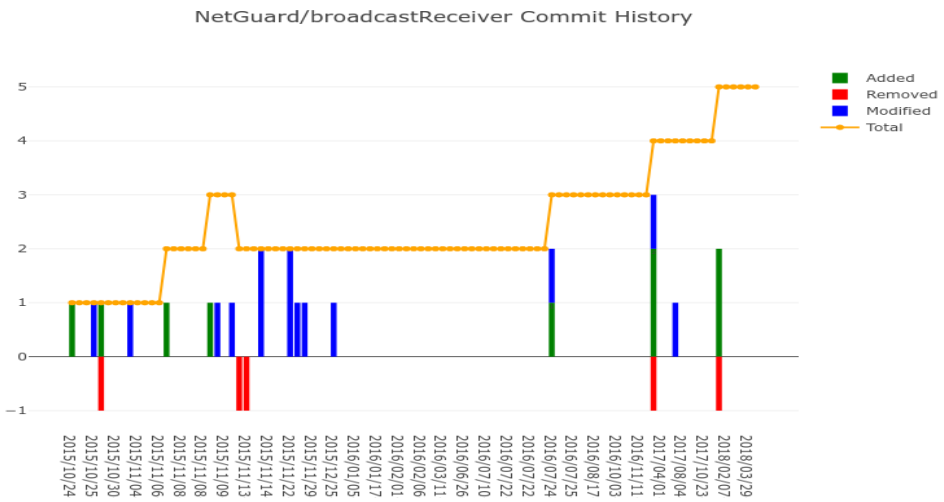


Figura 14: Histórico dos broadcast receivers do repositório NetGuard

Já no gráfico da Figura 14, vemos que há uma distribuição diferente do caso de activities, onde percebemos mais mudanças e remoções associadas, inclusive recentemente. A quantidade maior de mudanças no período recente pode ter se dado por conta de uma possível necessidade de alterar os intent filters dos receivers para a adequação às novas versões Android.

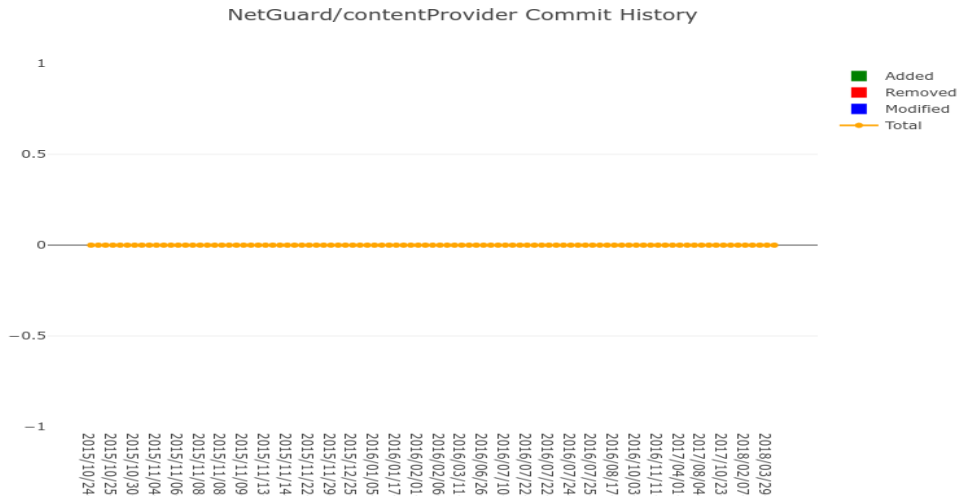


Figura 15: Histórico dos content providers do repositório NetGuard

No gráfico da Figura 15 vemos apenas uma linha horizontal indicando que em todos os commits a quantidade total de providers é zero. Isto ocorre por que em nenhum momento a aplicação fez uso deste tipo de componente, muito provavelmente por não ser necessário neste tipo de aplicação.

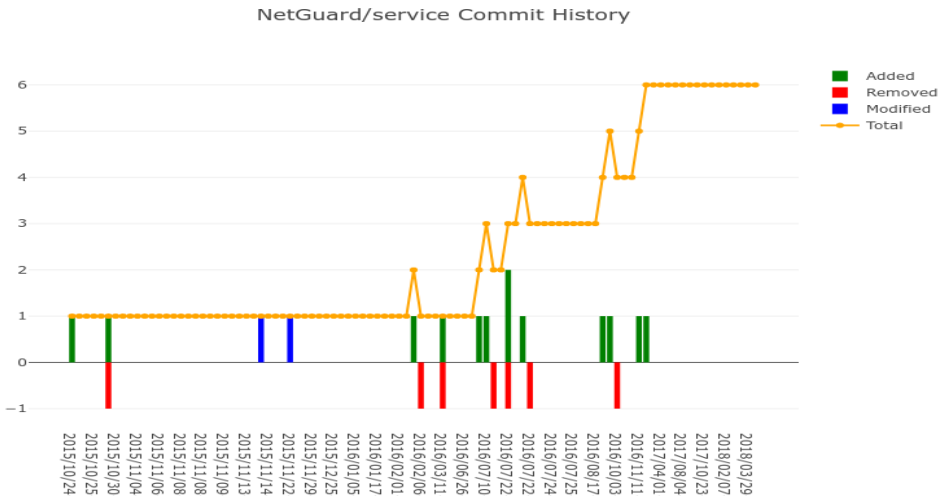


Figura 16: Histórico dos services do repositório NetGuard

No histórico dos services, na Figura 16, percebemos várias adições seguidas de remoções, que podem indicar substituição de um service por outro: primeiro desenvolve um service, depois deixa o que já existia deprecated, e então remove. Ou ainda apenas uma mudança no nome da classe que o implementa. Uma análise mais aprofundada do repositório pode revelar o que houve, a ideia é que a ferramenta proposta no trabalho está justamente permitindo algumas destas visualizações que podem dar insights para compreensão do processo de desenvolvimento do app.

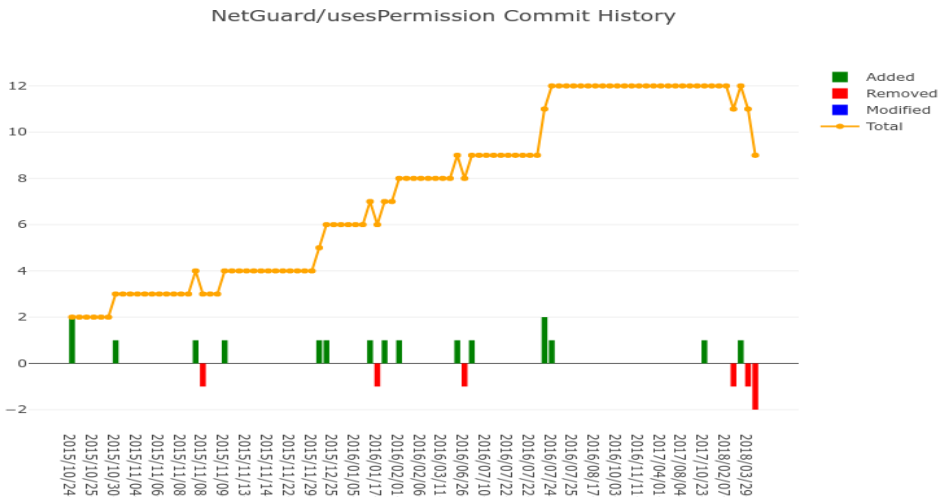


Figura 17: Histórico das permissões utilizadas no repositório NetGuard

Na Figura 17 podemos verificar a tendência de sempre adicionar mais permissões do que remover. Analisando os commits mais recentes, percebemos que algumas permissões foram removidas, isso pode ocorre por consequência de alguma API utilizada pela aplicação que após ser atualizada não necessita mais a requisição de algumas permissões.

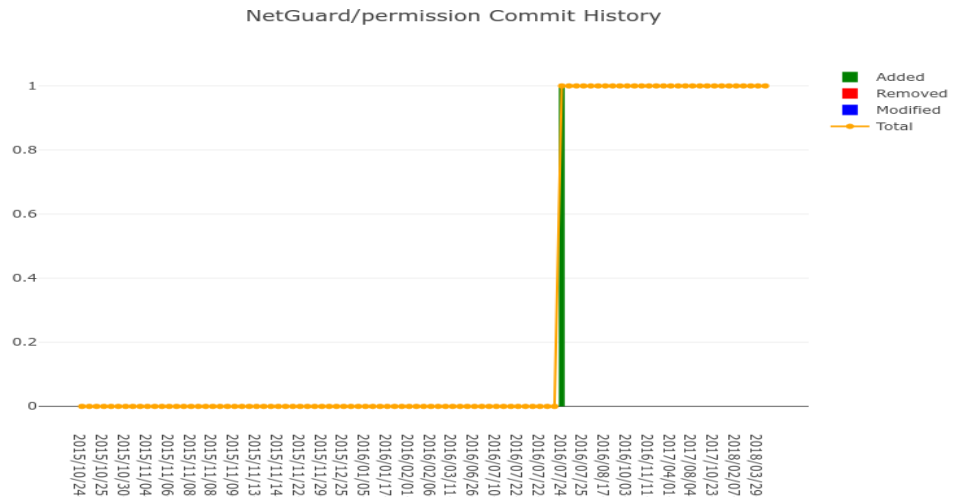


Figura 18: Histórico das permissões criadas pelo repositório NetGuard

Na Figura 18 podemos verificar que apenas uma permissão foi definida pela aplicação. Após análise do commit, vimos que esta permissão foi criada para proteger os widgets criados pela aplicação, impedindo que outras aplicações tenham acesso a eles.

# 4 Conclusão

Após os experimentos realizados, podemos concluir que a implementação da ferramenta proposta no começo do semestre foi bem sucedida. Por mais que alguns repositórios, 57 para ser exato, tenham apresentado problemas durante a mineração, outros 1138 foram processados com sucesso, conforme foi comentado na Seção 3.4.

Na seção a seguir, apresentamos alguns trabalhos que poderiam fazer uso de uma ferramenta deste tipo. E na seção seguinte, sugerimos alguns trabalhos futuros que visam evoluir a ferramenta proposta.

## 4.1 Trabalhos Relacionados

Calciati e Gorla [5] realizaram um estudo sobre como as requisições de permissões evoluíram ao longo das várias releases das aplicações. Eles extraíram essa informação do arquivo APK, a versão final da aplicação que é baixada pelos dispositivos dos usuários. Este trabalho poderia ser enriquecido com informações a respeito do desenvolvimento do código fonte das aplicações com o uso da ferramenta proposta.

Krutz *et al* [6] desenvolveram uma ferramenta semelhante, denominada oSARA, para avaliar quem realizava as alterações nas permissões das aplicações. Em comparação com oSARA, a ferramenta proposta é capaz de coletar informações sobre outros componentes Android além das permissões.

Uma base de dados semelhante, porém muito maior, foi implementada por Geiger *et al* [14]. Chamado de AndroidTimeMachine, o banco de dados baseado em grafos contém dados sobre o histórico completo de mais de 8000 repositórios Android de código aberto encontrados na Google Play Store. Diferentemente da ferramenta proposta, esta base de dados não entra em detalhes das modificações efetuadas nos commits. Ela está mais concentrada apenas em coletar informações do repositório github como um todo e linkar com o app na loja da play store.

## 4.2 Trabalhos Futuros

Hoje a ferramenta extrai dados da aplicação olhando apenas para o seu arquivo de manifesto. Em um trabalho futuro ela pode ser estendida para analisar também o código fonte e validá-lo junto ao manifesto. Dessa forma, além de sabermos em que momento um componente foi acrescentado no arquivo XML, também saberíamos em que momento a classe que o representa foi implementada.

Uma outra opção de melhoria da ferramenta seria utilizá-la para obter dados sobre a implementação de outros elementos Android, como por exemplo, **`AlarmManagers`**, **`AsyncTasks`** e **`Notifications`**. Que apesar de não serem declarados no manifesto, são muito utilizados pelas aplicações para agendar procedimentos, executar tarefas assíncronas e exibir notificações para o usuário, respectivamente.

Na Seção 3.5 apresentamos um resumo dos dados obtidos pela ferramenta que foi gerado por um script Python. Uma abordagem diferente seria utilizar os arquivos CSV para gerar um banco de dados baseado em grafos e a partir disso realizar queries relativas aos dados desejados.

# Referências


[1] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *2009 6th IEEE International Working Conference on Mining Software Repositories*, pp. 1–10, May 2009.

[2] T. Menzies and T. Zimmermann, "Software analytics: So what?," *IEEE Software*, vol. 30, pp. 31–37, July 2013.

[3] "Repodriller." `https://github.com/mauricioaniche/repodriller`.

[4] "Android developers." `http://developer.android.com`.

[5] P. Calciati and A. Gorla, "How do apps evolve in their permission requests?: A preliminary study," in *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, (Piscataway, NJ, USA), pp. 37–41, IEEE Press, 2017.

[6] D. E. Krutz, N. Munaiah, A. Peruma, and M. W. Mkaouer, "Who added that permission to my app?: An analysis of developer permission changes in open source android apps," in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '17, (Piscataway, NJ, USA), pp. 165–169, IEEE Press, 2017.

[7] Y. Lyu, J. Gui, M. Wan, and W. G. J. Halfond, "An empirical study of local database usage in android applications," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 444–455, Sept 2017.

[8] "F-droid." `https://f-droid.org/en/`.

[9] "The android source code." `https://source.android.com/setup/`.

[10] "Git." `https://git-scm.com/`.

[11] "Git handbook." `https://guides.github.com/introduction/git-handbook/`.

[12] "Python." `https://www.python.org/`.

[13] J. P. T. Trindade, "IncR: Ferramenta de detecção incremental de comunicação entre componentes Android." *Universidade Federal de Pernambuco*, Trabalho de Graduação, 2016.

[14] F.-X. Geiger, I. Malavolta, L. Pascarella, F. Palomba, D. D. Nucci, I. Malavolta, and A. Bacchelli, "A graph-based dataset of commit history of real-world android apps," in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR*, (New York, NY), p. to appear, ACM, May 2018.