Rafael Alvarenga de Azevedo

Estudo e implementação de classificadores binários para detecção de *malwares* de Android baseados em *features* estáticas

Formiga - MG

2023

Rafael Alvarenga de Azevedo

Estudo e implementação de classificadores binários para detecção de *malwares* de Android baseados em *features* estáticas

Monografia do trabalho de conclusão de curso apresentado ao Instituto Federal Minas Gerais - Campus Formiga, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Instituto Federal de Educação, Ciência e Tecnologia de Minas Gerais

Campus Formiga

Ciência da Computação

Orientador: Mário Luiz Rodrigues Oliveira

Formiga - MG

2023

Azevedo, Rafael Alvarenga de
A994e Estudo e implementação de classificadores binários para detecção de malwares de Android baseados em features estáticas / Rafael Alvarenga de Azevedo – Formiga : IFMG, 2023.
106p. : il. color.

Orientador: Prof. Me. Mário Luiz Rodrigues Oliveira
Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) Instituto Federal de Educação,Ciência e Tecnologia de Minas Gerais – *Campus* Formiga.

1. Revisão sistemática da literatura. 2. Análise de malware. 3. Análise estática. 4. Classificadores binários. I. Oliveira, Mário Luiz Rodrigues. II. Título.

CDD 004

Ficha catalográfica elaborada pela Bibliotecária Lívia Renata Santos- CRB/6-2561

MINISTÉRIO DA EDUCAÇÃO
SECRETARIA DE EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE MINAS GERAIS
Campus Formiga
Diretoria de Ensino
Docência Área Acadêmica de Computação
Rua São Luiz Gonzaga, s/n - Bairro São Luiz - CEP 35570-000 - Formiga - MG
- www.ifmg.edu.br

RAFAEL ALVARENGA DE AZEVEDO

Estudo e implementação de classificadores binários para detecção de malwares de Android baseados em features estáticas

Trabalho de Conclusão de Curso apresentado ao Instituto Federal de Minas Gerais - Campus Formiga, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

APROVADO em: 21 de novembro de 2023.

BANCA EXAMINADORA

Prof. Mário Luiz Rodrigues Oliveira(orientador, IFMG)

Prof. Everthon Valadão dos Santos (IFMG)

Prof.º Walace de Almeida Rodrigues (IFMG)

Documento assinado eletronicamente por **Mario Luiz Rodrigues Oliveira**, **Professor**, em 21/11/2023, às 20:05, conforme Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Everthon Valadao dos Santos**, **Professor**, em 21/11/2023, às 20:05, conforme Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Walace de Almeida Rodrigues**, **Professor**, em 21/11/2023, às 20:33, conforme Decreto nº 10.543, de 13 de novembro de 2020.

Dedico este trabalho aos meus pais e aos meus avós, que sempre me incentivaram a estudar. Em reconhecimento à dedicação e sacrifícios que fizeram para me ajudar a alcançar meus objetivos.

Agradecimentos

Agradeço aos meus pais e aos meus avós por me proporcionarem a oportunidade de estudar o que eu gosto. Também, sou grato por tudo aquilo que aprendi com meus professores, dentro e fora da sala de aula. Por fim, gostaria de destacar a minha profunda admiração pelos educadores que sempre estiveram dispostos a dialogar diante das minhas provocações.

"Criar o que não existe ainda deve ser a pretensão de todo sujeito que está vivo."
(Paulo Freire)

Resumo

Este trabalho oferece uma visão abrangente sobre a crescente dependência da sociedade contemporânea em relação à tecnologia, em particular aos dispositivos móveis . Diante disso, destaca-se que aplicativos maliciosos de Android podem ser usados para uma ampla gama de atividades ilícitas que afetam a segurança e a privacidade dos usuários. Logo, é necessário uma intervenção por parte de pesquisadores para responder e mitigar esse tipo de ameaça cibernética. Para contribuir com essa premissa, conduziu-se uma revisão sistemática da literatura, de estudos entre os anos de 2015 a 2021, para identificar técnicas e ferramentas que tratam problemas de Análise de *Malware* em dispositivos Android. Consequentemente, leram-se 60 artigos na íntegra para compilar e classificar técnicas e ferramentas, tipos de *malwares* e técnicas anti-análise de *malware*, mais recorrentes. Por conseguinte, os resultados deste trabalho permitem concluir que a análise de *malwares* em dispositivos Android está em constante evolução, pois existem técnicas tanto para analisar amostras, como para impedir esta análise. Ainda, identificaram-se 118 técnicas, de classes distintas, para tratar os problemas dessa área de estudo; 357 ferramentas, classificadas como as técnicas; 9 técnicas anti-análises; e 14 tipos de *malwares*. Adicionalmente, por meio da montagem de um *dataset* com 10000 aplicativos, legítimos e maliciosos, com o emprego das técnicas *Permission based Analysis*, *Call Graph Analysis* e *Taint Analysis*, concluiu-se que é possível caracterizar aplicativos Android, na medida em que são identificados métodos alcançáveis no *call graph* de um app e métodos que vazam informações sensíveis do usuário. Com o *dataset* pronto, implementou-se e avaliou-se os resultados dos seguintes classificadores binários: *Decision Tree*, *Random Forest*, *Adaboost*, *Naive Bayes* e SVM (RBF).

Palavras-chave: Revisão Sistemática da Literatura, Análise de *Malware*, Análise Estática, Classificadores Binários.

Abstract

This work provides a comprehensive overview of the growing dependence of contemporary society on technology, particularly mobile devices. In light of this, it is worth noting that malicious Android applications can be used for a wide range of illicit activities that impact the security and privacy of users. Therefore, intervention by researchers is necessary to address and mitigate this type of cyber threat. To contribute to this premise, a systematic literature review was conducted, covering studies from 2015 to 2021, to identify techniques and tools that address malware analysis issues on Android devices. As a result, 60 articles were read in full to compile and categorize the most recurring techniques and tools, malware types, and anti-malware analysis techniques. Consequently, the results of this work lead to the conclusion that malware analysis on Android devices is continually evolving, with techniques available both for analyzing samples and for preventing such analysis. Furthermore, 118 techniques from different classes were identified to address the problems in this area of study, along with 357 tools categorized according to the techniques, 9 anti-analysis techniques, and 14 types of malware. Additionally, by assembling a dataset of 10,000 applications, both legitimate and malicious, and using Permission-based Analysis, Call Graph Analysis, and Taint Analysis techniques, it is possible to characterize Android applications by identifying reachable methods in an app's call graph and methods that leak sensitive user information. With the dataset ready, the following binary classifiers were implemented and evaluated: Decision Tree, Random Forest, Adaboost, Naive Bayes, and SVM (RBF).

Keywords: Systematic Literature Review, Malware Analysis, Static Analysis, Binary Classifiers.

Lista de ilustrações

Lista de tabelas

Lista de Algoritmos

Lista de abreviaturas e siglas

AD	Análise Dinâmica
AE	Análise Estática
AMD	Android Malware Dataset
ANN	Artificial Neural Network
API	Application Programming Interface
APK	Android Package
APP	Aplicativo
CART	Classification and Regression Trees
CNN	Convolutional Neural Network
DAE	Deep Autoencoder
DBN	Deep Belief Network
DT	Decision Tree
DNN	Deep Neural Network
EML	Extreme Machine Learning
FC	Fully Connected Neural Network
GKMC	Global K-means clustering
IccRE	Inter Component Communication Repository
KMC	K-means clustering
KNN	K-Nearest Neighbor
LRLR	Label Regularized Logistic Regression
LR	Logistic Regression
LSSVM	Least Squares SVM
LSTM	Long Short-Term Memory

MAIL	Malware Analysis Intermediate Language
MC	Markov Chain
MM	Markov Model
MLP	Multi-layer Perceptron
MIL	Multiple Instance Learning
NB	Naive Bayes
PART	Partial Decision Tree
RF	Random Forest
ROT	Rotation Forest
RNN	Recurrent Neural Network
SVM	Support Vector Machine
SCNN	Serial Convolutional Neural Network
SL	Simple Logistic
SLR	Systematic Literature Review
SMO	Sequential Minimal Optimization
VMI	Virtual Machine Introspection
WEKA	Waikato Environment for Knowledge Analysis
XGBoost	Gradient Boost

Sumário

1 Introdução

A sociedade contemporânea mostra-se cada vez mais dependente da tecnologia. Mais especificamente, dos serviços mantidos pela rede mundial de computadores. Enquanto a informação chega a todos, usuários mal-intencionados também usufruem das novas tecnologias. Eles dispõem da ferramenta que toda a sociedade utiliza: o *smartphone*. Dessa forma, aqueles com conhecimento técnico avançado podem abusar de seu funcionamento por meio da criação de programas maliciosos (*malwares*), que viabilizam atividades ilícitas, como: extorsão, espionagem, difamação ou geração de renda extra. Portanto, é necessário que pesquisadores busquem formas de responder a incidentes cibernéticos decorrentes do abuso de aplicativos de dispositivos móveis.

Aparelhos *Smartphones*, após adquiridos pelo usuário final, vêm com um sistema operacional instalado. Além desse sistema, é comum haver uma loja de aplicativos (APPs) que disponibiliza ferramentas, pagas ou gratuitas, que auxiliam o usuário em seu cotidiano, como: calendários, aplicativos de *streaming*, redes sociais, câmera, entre outros. No sistema operacional *Android*, a loja de aplicativos é chamada *Play Store*. É nela que desenvolvedores disponibilizam suas aplicações para outros usuários tirarem proveito. No entanto, também é possível que desenvolvedores mal-intencionados publiquem aplicações gratuitas que realizem atividade maliciosa, sem o consentimento das vítimas.

Ademais, aplicações maliciosas são empregadas em contextos específicos. Com elas, atacantes podem: extorquir vítimas com o uso de *ransomwares*; espionar um alvo com *spywares*; enviar anúncios como spam mediante *adwares*; disseminar um programa malicioso chamado *worm*; colocar mais *malwares* no dispositivo da vítima com o uso de *trojans*, entre outros. Em resposta a essas situações, pesquisadores dispõem de técnicas e ferramentas para analisar *malwares*. Logo, busca-se entender quais técnicas e ferramentas são usadas para analisar programas maliciosos de *Android*.

Considerando os aspectos mencionados, este trabalho de conclusão de curso (TCC) tem por objetivo principal implementar uma abordagem de análise estática de aplicativos Android para extrair *features* e treinar classificadores binários com modelos de aprendizagem de máquina supervisionada. Sendo assim, busca-se o desenvolvimento de um classificador de *malwares* de *Android*.

Na sequência, apresenta-se a justificativa para a realização deste TCC, bem como os objetivos gerais e específicos. A fundamentação teórica deste estudo é detalhada no capítulo 2. A metodologia adotada é descrita no capítulo 3. Sintetizou-se os resultados no capítulo 4. Concluiu-se o estudo no capítulo 5 e tratou-se dos trabalhos futuros no capítulo 6.

1.1 Justificativa

Num levantamento realizado em 2019, constatou-se que há 230 milhões de *smartphones* ativos no Brasil (ESTADÃO, 2019). Além disso, segundo Taylor (2023), o sistema operacional Android detém mais de 70% de participação de mercado, mundialmente, quando comparado ao iOS e outros. Sendo assim, tais dispositivos são alvo de ataques sofisticados que operam programas maliciosos. Exemplo disso, o *malware* brasileiro denominado *BrasDex*. Segundo Sampaio (2023), o *software* intercepta transações bancárias Pix e altera o destinatário. Os alvos são aplicativos Android de bancos conhecidos, como: Banco do Brasil, Nubank, PicPay, Inter, entre outros.

Vale ressaltar, contudo, que a comunidade científica tem produzido trabalhos com foco em análise de *malwares*. Grupos de pesquisa estão reagindo à demanda do mercado em segurança da informação, e produzindo novos conhecimentos sobre o tema para avançar o estado da arte. As produções científicas de Galante et al. (2018), Botacin, Grégio e Geus (2019) e Pontte, Dominico e Alves (2023) são exemplos de publicações recentes.

1.2 Objetivos

Nesta seção, detalha-se o objetivo geral deste trabalho, também, enumeram-se os objetivos específicos que direcionam seu desenvolvimento.

1.2.1 Objetivo Geral

O objetivo principal deste trabalho foi realizar uma SLR de técnicas e ferramentas usadas na análise de *malwares* de dispositivos Android. Além disso, implementaram-se abordagens de análise estática, *Permission based Analysis*, *Call Graph Analysis* e *Taint Analysis*, para extrair *features*, montar um *dataset* e treinar classificadores binários com modelos de aprendizagem de máquina supervisionada.

1.2.2 Objetivos Específicos

1. Identificar e categorizar técnicas e ferramentas utilizadas em análise de *malwares* de Android;
2. Identificar os principais tipos de *malwares* desenvolvidos por usuários maliciosos do Android;
3. Identificar técnicas anti-análise de *malwares*, e
4. Aplicar e comparar técnicas e ferramentas utilizadas em análise de *malwares* de Android.

2 Fundamentação Teórica

Nesta seção, serão apresentados os fundamentos teóricos essenciais para a realização e compreensão das tarefas propostas. Os tópicos a seguir serão detalhados: Análise de Fluxo de Dados, Análise de *Malware*, Revisão Sistemática da Literatura, Fundamentos de Aplicativos Android e Aprendizado de Máquina Supervisionado.

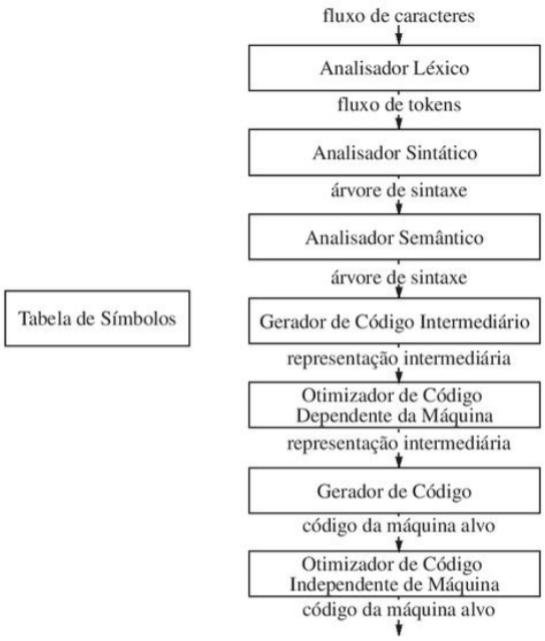
2.1 Análise de Fluxo de Dados

No estudo de Compiladores, existem algoritmos para traduzir um código em representação intermediária para um código equivalente descrito conforme o conjunto de instruções de uma arquitetura específica. Como mostra a figura 1, esse processo faz parte do *backend* de um compilador e começa pela fase *Otimizador de Código Dependente de Máquina* (AHO et al., 2007).

Antes de gerar o código de montagem, ou código da máquina alvo, realizam-se análises a fim de melhorar o desempenho ou reduzir o tamanho do código gerado. Entre elas, destaca-se a Análise de Fluxo de Dados, que assume o programa como um grafo, para tentar determinar comportamentos dinâmicos — ou de tempo de execução — de um programa, em tempo de compilação (NIELSON; NIELSON; HANKIN, 2010).

O problema de determinar o comportamento de um programa sem executá-lo é indecidível. Segundo Cohen (1987), é impossível criar um algoritmo que identifique todos os tipos possíveis de vírus, definindo, assim, a *Impossibilidade de Cohen*. Formalmente, enuncia-se que para todo algoritmo A, que recebe um código de teste para identificar se é vírus, existe um programa `p(pgm): if A(pgm) then sair; else infectar`, que, quando $pgm = p$, p infecta — ou comporta-se como vírus — se e somente se $A(p)$ é negado, ou "A não detecta p como vírus". Logo, por contradição, é impossível que A detecte todos os tipos de vírus, uma vez que, sempre haverá um programa que escapa a detecção.

Figura 1 – Fases de um compilador.



Fonte: (AHO et al., 2007).

Uma representação intermediária, segundo Appel e Palsberg (2003), é uma linguagem que não depende da linguagem fonte e consegue expressar as operações de uma arquitetura alvo sem considerar detalhes específicos de máquina. Então, é crucial a escolha de uma boa representação intermediária para realizar otimizações.

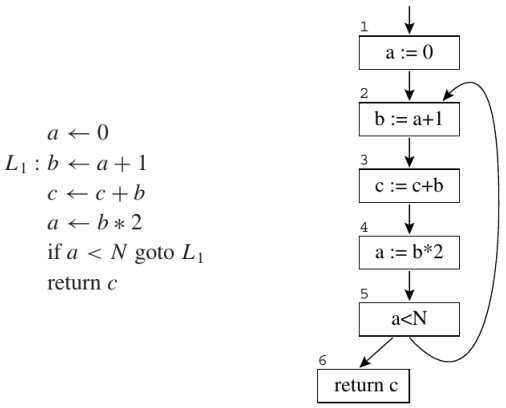
Para efetuar a Análise de Fluxo de Dados, constrói-se um Grafo de Fluxo de Controle (do inglês *Control Flow Graph*, ou CFG). Esse grafo é definido por nós que representam blocos básicos de um programa, e arestas que descrevem caminhos no fluxo de execução do código. A figura 2a mostra um exemplo desse tipo de grafo: à esquerda, um código em representação intermediária; à direita, o CFG equivalente. Ressalta-se que alguns autores, como Aho et al. (2007), escolhem representar os blocos básicos com mais de uma instrução até chegar em um *goto*, ou desvio condicional, como ilustrado na figura 2b.

Uma otimização usualmente feita por um compilador é eliminação de código morto, ou seja, o compilador tenta eliminar instruções inseridas no código que não influenciam seu fluxo de execução. Para completar essa tarefa, solucionam-se as equações de fluxo de dados descritas no processo de análise de variáveis vivas (ou *Liveness Analysis*) e eliminam-se aquelas instruções que contém variáveis que não serão usadas no futuro (APPEL; PALSBERG, 2003).

As equações de fluxo de dados são definidas pelos seguintes conjuntos de fluxo de dados: in, out, use, def e $succ$. O conjunto in é formado por variáveis que estão vivas na entrada de um bloco básico. O conjunto out é denotado pelas variáveis que estão vivas na

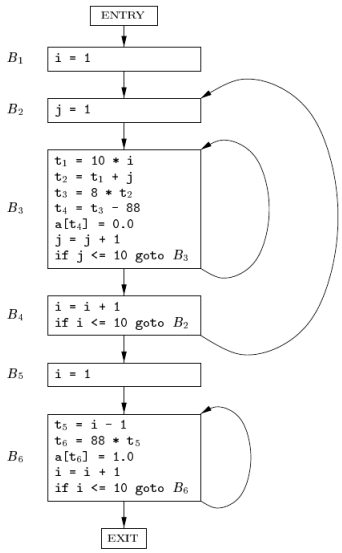
Figura 2 – Dois exemplos de CFG.

(a) Primeiro exemplo.



Fonte: (APPEL; PALSBERG, 2003).

(b) Segundo exemplo.



Fonte: (AHO et al., 2007).

saída de um nó do CFG. O conjunto *succ* é definido por nós do CFG que são sucessores de um nó atual. O conjunto *use* é composto por variáveis usadas em um nó do CFG, enquanto o conjunto *def* por variáveis definidas em dado nó. Sendo assim, diz-se que uma variável está viva em uma aresta do grafo se tem um caminho direto dessa aresta até um nó que use a variável e não passe por alguma definição dela (APPEL; PALSBERG, 2003).

Além disso, define-se a direção de propagação do fluxo de dados e como os fluxos são transformados para rastrear as informações. A propagação pode ser *forward* ou *backward*. Enquanto isso, os fluxos são transformados com base na forma como os dados rastreados são tratados nos caminhos do CFG. No caso da *Liveness Analysis*, segundo Appel e Palsberg (2003), propagam-se os fluxos em modo *forward* e assume-se que uma variável pode estar viva em *qualquer* um dos caminhos no CFG, dado uma bifurcação. Portanto, ambos aspectos são definidos pelo conjunto *succ* e pelo operador $\bigcup$, respectivamente, na equação 2.2. Finalmente, determina-se se os conjuntos *in* e *out* são inicializados vazios ou cheios, para serem operados até que não haja mudança. Logo, no caso de análise de variáveis vivas, inicializam-se tais conjuntos vazios.

As equações 2.1 e 2.2, quando computadas, montam todos os conjuntos. Nelas, assume-se n como sendo o número do nó no CFG, como disposto na figura 2a. Uma implementação de solução das equações, sugerida por Appel e Palsberg (2003), é demonstrada no pseudo-código 1.

$$in[n] = use[n] \cup (out[n] - def[n]) \tag{2.1}$$

$$out[n] = \bigcup_{s \in succ[n]} in[s] \tag{2.2}$$

Algorithm 1 Solução das equações de fluxo de dados

for all $n \in CFG$ **do**
  $in[n] \leftarrow \{\}$
  $out[n] \leftarrow \{\}$
end for
repeat
  for all $n \in CFG$ **do**
    $in'[n] \leftarrow in[n]$
    $out'[n] \leftarrow out[n]$
    $in[n] \leftarrow use[n] \cup (out[n] - def[n])$
    $out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$
  end for
until $in'[n] = in[n] \wedge out'[n] = out[n]$ para todo n

Portanto, como enunciam Appel e Palsberg (2003), se existe uma quádrupla (instrução) $s : a \leftarrow b \bigoplus c$ ou $s : a \leftarrow M[x]$, tal que a não está no conjunto *out* do nó s, então essa quádrupla pode ser eliminada.

Por fim, ressalta-se que as análises apresentadas são do tipo *intraprocedural*. Este tipo de análise é caracterizado por considerar apenas instruções em um único procedimento. Todavia, é possível realizar outro tipo de análise, intitulado *interprocedural*, em que são consideradas mais de um procedimento na análise (APPEL; PALSBERG, 2003), como é o caso da *Taint Analysis* apresentada na seção 3.2.2.3.

2.2 Análise de *Malware*

Segundo Sikorski e Honig (2012 apud MONTEIRO et al., 2018, p. 350), "qualquer *software* que realize qualquer ação que cause danos ao usuário, computador ou rede pode ser considerado um *malware*, incluindo vírus, trojans, *worms*, *rootkits*, *scareware* e *spyware*.". Tendo isso em vista, a análise de *malware* tem por objetivo garantir que o profissional saiba exatamente o que ocorreu com os dispositivos infectados por códigos maliciosos. Mais especificamente, a intenção do analista é descobrir como um executável se comporta no computador da vítima (SIKORSKI; HONIG, 2012).

Em linhas gerais, o analista de *malware* dispõe de três técnicas para tentar classificar programas maliciosos, conforme os comportamentos analisados. A primeira técnica chama-se Análise Dinâmica. Ela é usada, quando o profissional deseja entender o que acontece ao passo que um dado programa nocivo executa. O segundo esquema é denominado Análise Estática. Nesse caso, o analista utiliza ferramentas específicas para obter informações comportamentais de um programa, sem executá-lo. Uma vez com esses dados em mãos, o perito pode usufruir deles, numa possível heurística dinâmica. A última estratégia

é conhecida como Análise Híbrida, cujo diferencial é juntar a Análise Estática com a Dinâmica (SIKORSKI; HONIG, 2012).

Para a execução eficiente desses artifícios, o profissional precisa ter conhecimento aplicado em Ciência da Computação, pois, para conseguir compreender a maneira conforme um programa é executado num computador, o experto deve utilizar-se da análise crítica desenvolvida no estudo de disciplinas como Sistemas Operacionais, Arquitetura e Organização de Computadores, Redes, Compiladores, Estrutura de Dados, entre outras. Ainda assim, existem diversas ferramentas que auxiliam o técnico a completar o exame de *malwares*. Afinal, os diferentes níveis de abstrações, oferecidos pelos utensílios, podem acelerar este processo na totalidade.

Conforme Monteiro et al. (2018) categorizaram, existem algumas ferramentas usualmente utilizadas na análise de *malwares* desenvolvidos para *Windows*, para cada técnica associada. Na análise estática, é comum operar programas como *IDA Pro*. Na análise dinâmica, são empregados depuradores como *OllyDBG*, *WinDBG* e *x64dbg*. Todos eles retornam valores em linguagem de máquina que podem ajudar na compreensão do comportamento de amostras investigadas, sem conhecer seu código-fonte. Ainda, pode-se isolar a execução de um programa malicioso com virtualizadores de máquinas como *VMWare* e *Oracle VirtualBox*. Caso o analista deseje usar ambas as técnicas, é possível manusear a suíte *SysInternals* em conjunto com as demais equipagens para coletar informações mais detalhadas sobre programas e processos (MONTEIRO et al., 2018).

2.3 Revisão sistemática da literatura

A revisão sistemática da literatura (SLR, do inglês *systematic literature review*) é um meio para identificar, avaliar e interpretar todas as pesquisas recentes relacionadas a um assunto em particular. Pesquisas individuais que contribuem para uma revisão sistemática são chamadas de primárias, e a revisão em si, secundária (KITCHENHAM; CHARTERS, 2007).

Com essa técnica de estudo, é possível sintetizar trabalhos existentes de forma justa. Para colocá-la em prática, é necessário seguir uma estratégia de pesquisa. Esta, deve permitir que a totalidade da pesquisa seja avaliada. Pesquisadores em processo de construção de uma revisão sistemática devem procurar pesquisas que não suportam sua hipótese inicial, bem como estudos que suportam (KITCHENHAM; CHARTERS, 2007).

Em consonância com Kitchenham e Charters (2007), uma revisão sistemática possui uma metodologia bem definida e é capaz de mostrar resultados não tendenciosos. Contudo esse estudo sistemático não consegue proteger-se contra o viés da publicação nos estudos primários. Além disso, é capaz de fornecer informações sobre fenômenos diante de um intervalo vasto de configurações e métodos empíricos. Ademais, consegue aumentar a

chance de detectar efeitos reais no final do processo de revisão, enquanto que estudos individuais não conseguem fazê-lo. Porém, essa forma de pesquisa requer muito esforço quando comparado a revisões de literatura tradicionais.

A realização de uma revisão sistemática pode ser separada em três etapas: planejamento, condução e relatório. Na primeira, são feitas a identificação da necessidade da pesquisa, a especificação da pergunta da pesquisa, o desenvolvimento de um protocolo para revisão e a avaliação deste protocolo. Em segunda instância, é feita a extração de todos os estudos primários; bem como a avaliação, monitoramento e síntese destes. No último momento, o relatório é formatado e avaliado (KITCHENHAM; CHARTERS, 2007).

2.4 Fundamentos de Aplicativos Android

Antes de realizar qualquer análise, procurou-se entender como aplicativos Android são organizados e executados no sistema. Com auxílio da documentação oficial do Android, resumiram-se os conceitos fundamentais para compreender como programas são executados em dispositivos Android. Ressalta-se que quando um analista de malware obtém uma amostra, ele dispõe da técnica de análise caixa-preta (*black-box analysis*), ou seja, ele reconhece o formato da amostra e qual sistema o executa, mas não sabe o quê ela realmente faz quando o usuário interage com ela. Sendo assim, é necessário conhecer o ambiente de execução do programa corrente para instrumentalizá-lo e, assim, compreender seu comportamento.

A seguir, apresenta-se uma visão geral de como aplicativos são executados, como eles especificam os recursos que utilizarão, quais são seus componentes, uma introdução sobre as versões da API do Android e seu sistema de permissões.

2.4.1 Visão Geral

Aplicativos Android podem ser desenvolvidos em uma das seguintes linguagens: Java, Kotlin ou C++. Mas, ressalta-se que as ferramentas usadas neste trabalho analisam programas escritos na linguagem Java. Os aplicativos são distribuídos em lojas como a Play Store (loja oficial da Google) e são empacotados no formato APK (ou *Android Package*). Para isso, utilizam-se as ferramentas providas pelo Android SDK.

Uma vez instalado, o app executa em um processo independente identificado pelo ID de usuário criado pelo sistema operacional baseado em Linux. Dessa forma, cada app corresponde a um usuário de sistema diferente e fica isolado das demais aplicações. Além disso, cada processo tem sua própria máquina virtual para executar seus *bytecodes*.

O Android implementa o princípio de menor privilégio, isto é, um aplicativo pode acessar apenas os componentes necessários para sua execução e não mais que isso.

Consequentemente, tem-se um ambiente de execução seguro, no qual apps não conseguem acessar recursos do sistema para os quais não têm permissão. No entanto, existem formas de aplicativos compartilharem recursos entre si, como *Intents*, por exemplo. Portanto, para acessar recursos protegidos pelo sistema, cada aplicativo deve especificar as permissões que precisa para operar, e deve apresentá-las ao usuário, para que ele decida se autorizará ou não o uso de tais recursos (ANDROID, 2023c).

Além disso, um aplicativo é distribuído com recursos audiovisuais usado em sua apresentação, como imagens, vídeos, arquivos de áudio, tabelas de *strings*, ou qualquer tipo de arquivo correlato (ANDROID, 2023c).

2.4.2 Arquivo de Manifesto

Todo aplicativo Android dispõe de um arquivo de manifesto, identificado como *AndroidManifest.xml* no pacote de distribuição (APK). Neste arquivo são declarados os componentes da aplicação, as permissões que ela requer para executar, o número mínimo e o número alvo da API do Android que ela é compatível, *hardwares* específicos que serão usados e bibliotecas externas a API do Android. Tal arquivo é especificado por meio da linguagem XML. Logo, todos os requisitos para executar o aplicativo são descritos em *tags XML* (ANDROID, 2023b).

As principais tags que compõem um arquivo de manifesto são:

- `<manifest>`: a tag raiz do arquivo, que contém metadados gerais sobre o aplicativo, como nome do pacote e versão;
- `<application>`: contém informações sobre o aplicativo em si, como atividades, serviços, provedores de conteúdo e permissões. É também onde a maioria das configurações globais do aplicativo são definidas;
- `<activity>`: descreve uma atividade (tela) do aplicativo. Contém informações como o nome da classe da atividade, método *main* do app e *intent-filters*;
- `<service>`: define um serviço em segundo plano no aplicativo, usado para realizar tarefas em execução contínua sem uma interface de usuário;
- `<receiver>`: declara um receptor de transmissões, permitindo que o aplicativo responda a eventos do sistema ou personalizados;
- `<provider>`: define um provedor de conteúdo, permitindo que aplicativos compartilhem dados com outros aplicativos;
- `<uses-permission>`: indica as permissões necessárias para o aplicativo acessar recursos protegidos do sistema, como câmera, localização, internet, entre outros;

- `<intent-filter>`: usado para definir os tipos de intenções (intents) que uma atividade, serviço ou receptor pode lidar. Isso determina como o componente pode ser acionado por outras partes do sistema ou por outros aplicativos;
- `<meta-data>`: permite adicionar metadados extras a componentes do aplicativo, que podem ser usados para configurações específicas;
- `<activity-alias>`: cria um apelido para uma atividade, permitindo que ela seja acessada por meio de diferentes identificadores.

2.4.3 Componentes

Um aplicativo Android é composto, sumariamente, por atividades, serviços, receptores de conteúdo, provedores de conteúdo e *intents*. Atividades são classes de interface com o usuário. Serviços são classes que serão executadas sem a intervenção do usuário. Receptores de transmissão possibilitam aplicativos responderem a eventos de sistema. Provedores de conteúdo são interfaces entre aplicativos para manipulação de arquivos ou base de dados. Os *Intents* são mensagens assíncronas que realizam comunicação inter-processo no Android. Na sequência detalha-se cada um desses conceitos.

2.4.3.1 Atividades

Uma atividade (ou *Activity*) representa uma tela ou uma interface do usuário em um aplicativo Android. Ela gerencia a interação com o usuário, exibindo informações e permitindo a entrada de dados. Cada tela visível em um aplicativo é geralmente implementada como uma atividade. As atividades podem ser iniciadas, pausadas, retomadas e finalizadas conforme a interação do usuário.

2.4.3.1.1 Ciclo de vida

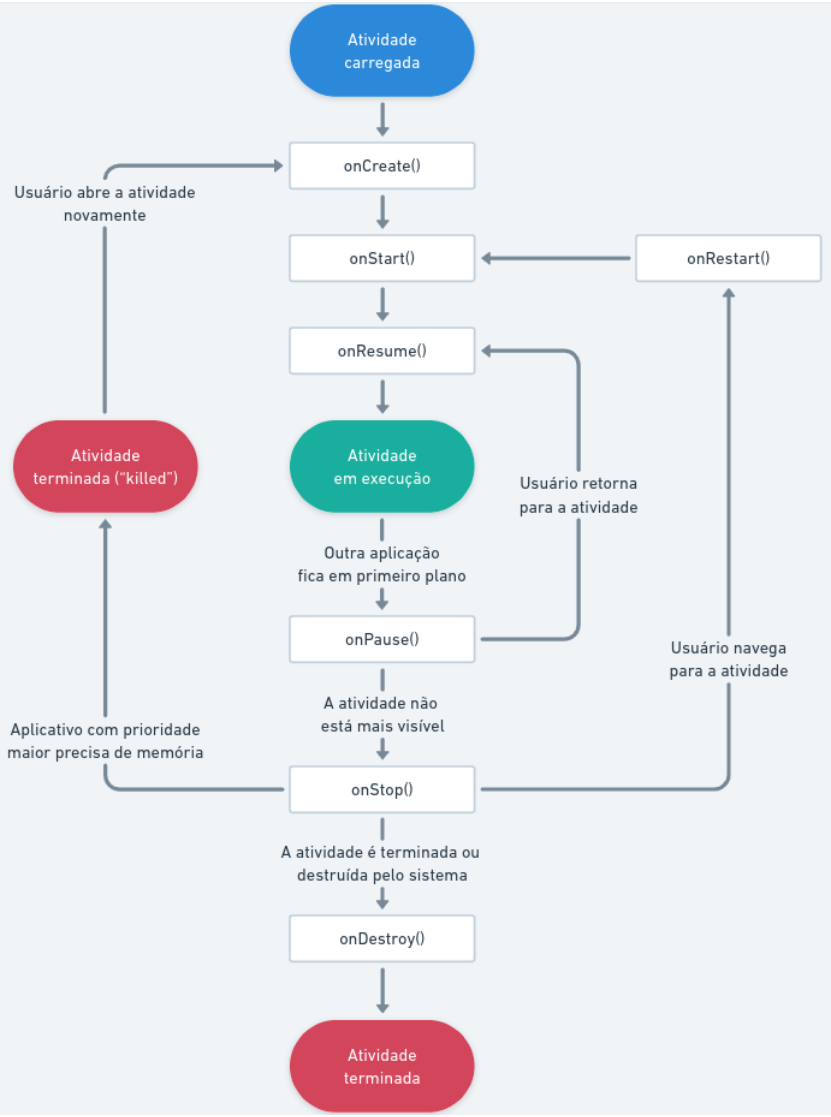
Segundo Android (2023a), o ciclo de vida de uma atividade em um aplicativo Android descreve as várias fases pelas quais a atividade passa, desde sua criação até sua finalização. A lista a seguir elenca os métodos que compõem uma atividade:

- `onCreate()`: a atividade é criada. Este é o primeiro método chamado e é onde a inicialização básica ocorre, como carregar o layout da interface do usuário e configurar recursos.
- `onStart()`: a atividade se torna visível para o usuário. Neste ponto, a atividade está prestes a entrar em primeiro plano, mas ainda não está interagindo ativamente com o usuário.

- `onResume()`: a atividade entra no estado ativo e está pronta para interagir com o usuário. Neste ponto, a atividade está em primeiro plano e pode receber entradas do usuário.
- `onPause()`: a atividade perde o foco, mas ainda é visível para o usuário. Isso geralmente acontece quando outra atividade é iniciada ou quando um diálogo é exibido.
- `onStop()`: a atividade não é mais visível para o usuário. Isso ocorre quando a atividade é interrompida ou uma nova atividade está ocupando toda a tela.
- `onDestroy()`: a atividade está sendo destruída. Isso pode acontecer quando o sistema precisa liberar recursos, quando o usuário fecha a atividade ou quando há uma mudança de configuração (como girar o dispositivo).

Além dessas etapas principais, existem também os métodos `onRestart()` (chamado quando a atividade volta a ser reiniciada após ter sido parcialmente interrompida) e `onSaveInstanceState()` (usado para salvar informações importantes antes que a atividade seja destruída, permitindo restaurar o estado quando a atividade for recriada). A figura 3 mostra um fluxograma do funcionamento do ciclo de vida de uma atividade.

Figura 3 – Ciclo de vida de uma atividade.



Fonte: (ANDROID, 2023a).

2.4.3.2 Serviços

Um serviço (ou *Service*) é um componente que executa operações em segundo plano, sem uma interface do usuário visível. Os serviços são usados para executar tarefas de longa duração, como baixar arquivos, reproduzir música em segundo plano ou sincronizar dados. Eles operam independentemente das atividades e podem continuar a funcionar mesmo quando o aplicativo não está ativamente em primeiro plano.

2.4.3.3 Receptor de transmissões

Um receptor de transmissão (ou *Broadcast Receiver*) permite que um aplicativo responda a eventos do sistema ou personalizados. Eles "escutam" transmissões enviadas pelo sistema ou outros aplicativos e executam ações correspondentes. Isso permite que

os aplicativos respondam a eventos como mudanças de conectividade de rede, níveis de bateria e outras ações do sistema.

2.4.3.4 Provedores de Conteúdo

Um provedor de conteúdo (ou *Content Provider*) permite que aplicativos compartilhem e acessem dados entre si. Ele age como uma interface para armazenamentos de dados, como bancos de dados, arquivos e recursos, permitindo que outros aplicativos consultem e, em alguns casos, modifiquem esses dados de forma segura e controlada.

2.4.3.5 *Intents*

Intents são mensagens assíncronas que permitem que componentes diferentes (como atividades, serviços e receptores) solicitem ações específicas ou comuniquem-se entre si. Eles podem ser usados para iniciar atividades, iniciar serviços, transmitir informações e realizar ações dentro de ou entre aplicativos. Os *Intents* podem ser explícitos (direcionados a um componente específico) ou implícitos (indicando uma ação e deixando o sistema decidir qual componente lidará com ela).

2.4.4 Versões da API do Android

As versões, ou níveis, da API do Android se referem aos diferentes conjuntos de funcionalidades e recursos disponíveis para os desenvolvedores, ao criar aplicativos para o sistema. Cada versão de API representa uma iteração do sistema operacional Android e traz consigo melhorias, novos recursos e correções de bugs.

2.4.5 Sistema de permissões

O sistema de permissões no Android é uma parte crucial do sistema operacional que controla o acesso de aplicativos a recursos sensíveis e informações do dispositivo. A partir da versão Android 6.0 (nível de API 23), introduziu-se um modelo de permissões dinâmicas. Neste modelo, aplicativos solicitam permissões somente quando necessário, isto é, em tempo de execução. As permissões podem ser classificadas como “perigosas” ou “não perigosas”, com permissões perigosas exigindo aprovação explícita dos usuários. Isso proporciona aos usuários maior controle sobre as permissões concedidas e segue o princípio do menor privilégio, garantindo que os aplicativos tenham acesso mínimo necessário aos recursos. Além disso, os usuários podem revogar permissões a qualquer momento nas configurações do aplicativo, proporcionando um nível adicional de segurança e privacidade (ANDROID, 2023d).

2.5 Aprendizado de Máquina Supervisionado

De acordo com Coppin (2004), a área do conhecimento denominada Inteligência Artificial é responsável pelo estudo de qualquer sistema que se baseie na inteligência humana, ou de outros animais, para resolver problemas complexos. Mais especificamente, o campo de estudo intitulado Aprendizado de Máquina, é responsável por agregar princípios, técnicas e ferramentas que permitem computadores aprenderem a partir de exemplos. Assim que o computador aprende com um conjunto de entradas, gera-se um conjunto de valores de saída, que será finito ou representado por um número (RUSSELL; NORVIG, 2021). Logo, existem dois tipos de problemas de aprendizado, respectivamente: problemas de classificação e problemas de regressão. Além disso, conforme um agente aprende, ele pode ser afetado por viés (do termo em inglês *bias*) ao não aprender da forma correta, quando o conjunto de entrada — ou de treinamento — cresce.

Como definem Russell e Norvig (2021), um agente é qualquer entidade que consiga perceber e interagir com seu ambiente por meio de sensores. Quando esse agente é um computador, ele observará os dados de entrada e construirá um modelo baseado neles. Em seguida, tal modelo será usado tanto para criar uma hipótese sobre o mundo, como um programa que resolve problemas (RUSSELL; NORVIG, 2021). A depender do tipo de *feedback* que o agente possui, determinam-se três tipos de aprendizado: aprendizado supervisionado, aprendizado não supervisionado e aprendizado por reforço.

O aprendizado supervisionado é caracterizado pela presença de pares de entrada e saída que permitem ao agente aprender por meio de uma função que determina o valor de saída, dado uma entrada. O valor de saída pode ser um rótulo, ou um número. Quando rótulo, trata-se de uma tarefa de classificação, e, quando número, o problema tratado é de regressão. Já no aprendizado não supervisionado, o agente recebe entradas sem alguma forma explícita de *feedback*, como um rótulo. A tarefa mais comum tratada por esse tipo de aprendizagem é *clustering*. Por último, no aprendizado por reforço, o agente aprende por meio de recompensas e penalidades, de modo que suas ações sejam tomadas com base em experiências anteriores, bem sucedidas, que alcançam seu objetivo final (RUSSELL; NORVIG, 2021).

Ademais, quando um agente aprende com um conjunto de treinamento de dados, pode ocorrer viés. Conforme esse conjunto cresce, a função de aprendizagem pode não conseguir identificar padrões nos dados de entrada, levando ao fenômeno conhecido como *underfitting*. Por outro lado, a função de aprendizagem também pode ser excessivamente ajustada aos dados de treinamento, tornando-se incapaz de generalizar para amostras não vistas anteriormente, o que é denominado *overfitting* (RUSSELL; NORVIG, 2021).

3 Materiais e Métodos

Na presente seção, apresentam-se, detalhadamente, os materiais e métodos empregados no desenvolvimento deste trabalho. Quanto aos materiais usados, merece destaque a colaboração do IFMG — Campus Formiga, que forneceu um ambiente virtualizado por meio de um contêiner, para a implementação e teste da aplicação criada *SFEDroid*. Ainda, seguiram-se, rigorosamente, os procedimentos propostos por Senocak (2019) para condução da revisão sistemática da literatura acerca de Análise *Malware* em dispositivos Android. Por fim, detalhou-se quais técnicas, de Análise Estática, foram usadas para identificar *features* estáticas, também, quais técnicas foram empregadas para fazer o pré-processamento de *features*, antes de treinar os classificadores binários escolhidos.

3.1 Materiais

Compôs-se uma coleção de arquivos APKs, tanto de aplicativos maliciosos como legítimos, para construir um *dataset*. E, diversas ferramentas foram usadas para desenvolver os códigos deste trabalho. Para realizar Análise de Fluxo de Dados, instrumentalizou-se a biblioteca *FlowDroid*. Já para testar a implementação, configurou-se um ambiente de execução Java em um contêiner do campus. Enquanto isso, para validar alguns resultados no processo de desconstrução de *bytecodes* do Android, compararam-se os códigos gerados com os da ferramenta *jadx*. Também, adaptou-se um pacote *npm* para fazer *crawling* de aplicativos no site *ApkPure*. Por fim, implementou-se um *notebook* com algoritmos de Aprendizado de Máquina Supervisionado no *Kaggle*.

3.1.1 Coleção de APKs

Para montar o *dataset* de treinamento dos classificadores binários, obteve-se uma coleção de aplicativos. Os aplicativos maliciosos foram adquiridos no site, amplamente conhecido pela comunidade de analistas de *malware*, vx-underground. Enquanto os aplicativos legítimos foram baixados da loja de aplicativos ApkPure.

A coleção de aplicativos maliciosos disponibilizada no *vx-underground* é uma cópia do repositório de amostras Android Malware Dataset (AMD). Esse conjunto de dados não está mais disponível *online*. Então, buscou-se adquirir os arquivos mediante outras fontes. Neste caso, a comunidade internacional de analistas de *malware* citada tinha os arquivos buscados. Porém, como arquivos APK são binários no formato ZIP, testou-se cada arquivo APK do *dataset* antes de alimentá-lo ao *SFEDroid*. Por meio do programa *7-zip* e a linha de comando abaixo, eliminou-se do *dataset* APKs que apresentavam erros

em seu cabeçalho:

```
$ find . -type f -iname '*.apk' -print0 |\
xargs -0 -I{} sh -c '7z t {} | grep -i error && echo {} >> res.txt'
```

Logo, entendeu-se que tais arquivos estavam corrompidos e não eram suscetíveis a análise estática. Identificou-se, por uma linha de comando análoga a anterior, que o arquivo baixado *Argus.Dowgin.7z* estava totalmente corrompido e não foi usado. Os arquivos encontram-se no seguinte caminho de diretório do site: *root* > *Samples* > *Argus Collection*.

Concomitantemente, buscava-se coletar aplicativos legítimos. Contudo, como a coleção de *malwares* é do ano de 2016, procurou-se adquirir amostras legítimas da mesma época. Tentou-se obter acesso ao repositório *AndroZoo* mediante e-mail de solicitação, conforme exigido no site oficial da Universidade de Luxemburgo. Todavia, não se obteve nenhuma resposta. Mas, para dar continuidade no trabalho, contornou-se esse problema por meio da utilização do índice disponibilizado, publicamente, no site do repositório, para filtrar amostras legítimas (segundo informação do escanamento do site *VirusTotal*, contido no índice) até 2017.

Seguindo orientações do próprio site da universidade, baixou-se o arquivo *latest.csv.gz* e filtrou-se o índice das amostras por 50.000 aplicativos que continham os seguintes metadados: publicados na *Play Store*; escaneados pelo *VirusTotal* antes de 2017 (assumiu-se que o app não poderia ter sido lançado depois dessa data); nenhum antivírus do *VirusTotal* acusou a amostra como maliciosa naquela data; e o tamanho do arquivo não passava de 1.000.000 *bytes*. Limitou-se o tamanho dos aplicativos em 1MB devido a restrições de *hardware*, mas reconhece-se que aplicativos maiores podem conter mais dados e isso pode melhorar a caracterização das amostras de um *dataset*.

O comando abaixo realiza essa tarefa, e ainda grava em arquivo de texto os metadados *SHA-256*, *pkgName* e *versionCode*:

```
$ zcat latest.csv.gz |\
grep -v ',snaggamea' |\
awk -F, '{if ($11 ~ /play\.google\.com/) {print} }' |\
awk -F, '{if ($9 < 2017-01-01) {print}}' |\
awk -F, '{if ($8 == 0) {print}}' |\
awk -F, '{if ($5 < 1000000) {print}}' |\
awk -F, '{if (NR <= 50000) {print $1 "," $6 "," $7}}' >\
50K_1MB_apks_legit.txt
```

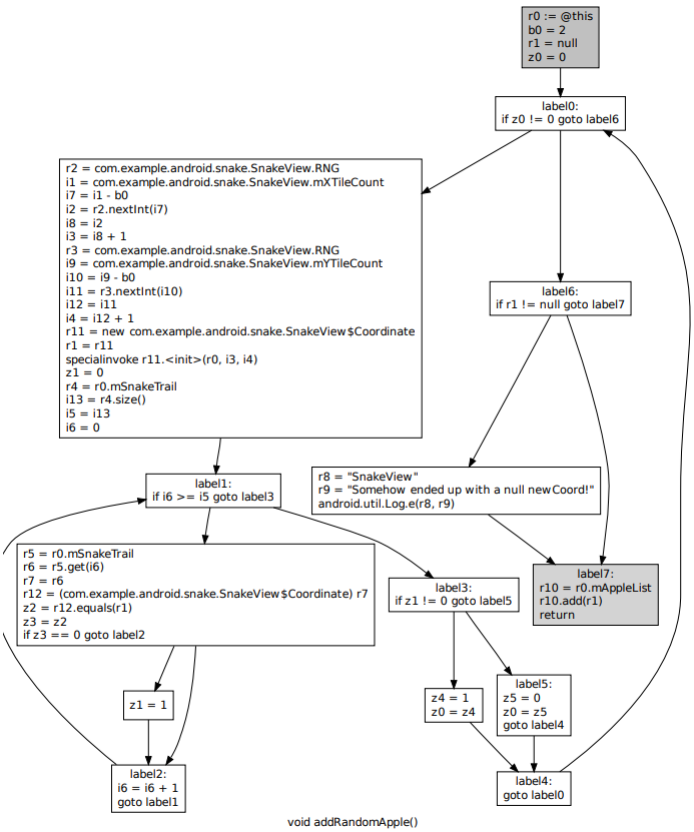
Deste modo, com os metadados salvos em arquivo de texto, alimentou-se o *script* descrito na seção 3.1.5, para tentar baixar os mesmos arquivos por um site diferente. Ao final, adquiriu-se apenas 5.000 aplicativos da lista, pois apenas estes estavam acessíveis no site *ApkPure* no momento do *download*.

3.1.2 Análise de Fluxo de Dados de Java

A ferramenta *SFEDroid* foi desenvolvida como uma extensão da ferramenta *FlowDroid*, visando facilitar o processo de Análise de Fluxo de Dados de aplicativos Android. A Análise de Fluxo de Dados empregada é conhecida como *Taint Analysis*, e ela permite rastrear como dados sensíveis do usuário fluem nos componentes do aplicativo (ARZT, 2017).

FlowDroid, por sua vez, é uma ferramenta baseada no framework de otimização de código Java chamado *Soot*. Ambas ferramentas são conhecidas pela comunidade científica e vislumbradas nos estudos primários da SLR conduzida. Como representação intermediária para realizar as análises, o framework gera código em *Jimple*. A figura 4 mostra um exemplo de CFG montado com blocos básicos codificados nessa representação intermediária.

Figura 4 – Exemplo de CFG com código Jimple.



Fonte: (BARTEL et al., 2012).

Em resumo, a *SFEDroid* coleta os resultados da *Taint Analysis* implementada

pela *FlowDroid*, para caracterizar aplicativos Android. O processo envolve a conversão de *bytecodes* do aplicativo Android (arquivo *classes.dex*), para a representação intermediária em *Jimple*, com o uso da ferramenta *dexpler* (BARTEL et al., 2012). A *FlowDroid* também analisa o Arquivo de Manifesto do app e gera dados que a *SFEDroid* coleta para formar o conjunto de dados final.

3.1.3 Ambiente de testes

O IFMG — Campus Formiga forneceu acesso remoto a um contêiner inserido na rede interna do campus. O ambiente virtualizado contava com os seguintes recursos de *hardware*:

- **CPU:** Intel(R) Core(TM) i7-6900K CPU @ 3.20GHz, com 8 núcleos e 16 *threads*;
- **RAM:** 24 GB;
- **HD:** 80 GB;
- **Sistema Operacional:** Ubuntu 22.04.3 LTS;

A escolha de um *hardware* com essas especificações foi motivada pela exigência de manter um computador dedicado operando, ininterruptamente, para analisar mais de 15.000 aplicativos, maliciosos e legítimos. Para analisar tais *apps*, o programa implementado, *SFEDroid*, executou ao longo de 2 semanas, aproximadamente, com múltiplas *threads*, para gerar o *dataset composto*.

3.1.4 Validação

A ferramenta *SFEDroid*, como explicitado na seção 4.6 deste trabalho, também exporta *Call Graphs* no formato DOT para facilitar a validação da representação das classes do app gerada. Como comparação, analisou-se o código descompilado gerado pela ferramenta *jadx*, para conferir se os métodos e classes gerados estavam realmente corretos.

No contexto de engenharia reversa de aplicativos Android, *jadx* é uma ferramenta de código aberto, projetada para auxiliar engenheiros reversos a descompilar e analisar o código-fonte de aplicativos Android compilados (QAISAR; LI, 2021).

3.1.5 *Crawler* em Node.js

Necessitava-se adquirir aplicativos legítimos lançados antes do ano de 2017, logo, buscou-se baixar aplicativos no site *ApkPure*. Esse site é conhecido por disponibilizar várias versões de aplicativos lançados ao longo dos anos na *Play Store*. No entanto, o site

não facilitava a aquisição de arquivos em lote. Dessa forma, programou-se um *script* em *Node.js* para realizar o *crawling* de aplicativos do site.

O pacote usado está disponível em https://is.gd/MUJfKY. O *crawler* desenvolvido baixa do site a partir de uma lista de arquivos que contém o Hash SHA-256, o nome do pacote APK e o código de versão. Para baixar cada aplicativo, o programa instancia abas de navegador sem interface gráfica (chamadas abas *headless*). Porém, o pacote original instanciava uma aba para cada arquivo, mas não as fechava após o término do *download*. Logo, corrigiu-se o problema para possibilitar o download de vários arquivos de uma só vez, sem consumir a memória RAM toda do computador. O código pode ser acessado em https://is.gd/zPj3FE.

3.1.6 Notebook no *Kaggle*

Empregou-se um *notebook*, em Python, hospedado na plataforma *Kaggle*, com as bibliotecas *scikit-learn*, *seaborn*, *matplotlib*, *pandas* e *numpy*, para executar e analisar os resultados de classificadores binários alimentados com o *dataset* gerado pela *SFEDroid*. A escolha por esta plataforma se justificou devido à sua capacidade de oferecer recursos computacionais de alto desempenho e uma variedade de ferramentas de análise de dados.

Em resumo, esse ambiente proporcionou uma infraestrutura eficiente e acessível para computar, analisar e publicar os resultados dos classificadores binários implementados. Além disso, permitiu a utilização de ferramentas de visualização de dados para gerar os gráficos que compõem os resultados deste trabalho.

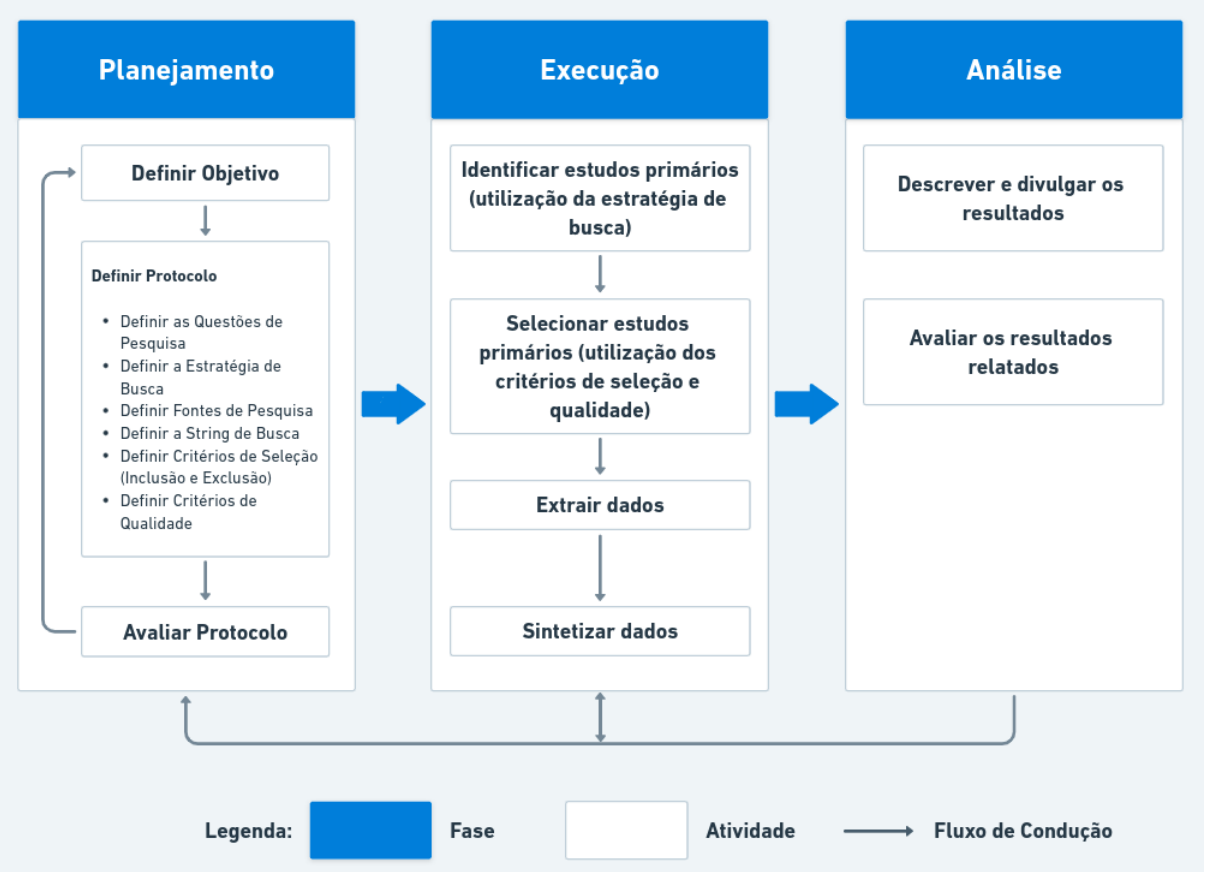
3.2 Métodos

Primeiramente, estabeleceu-se um protocolo para SLR, e conduziu-se, sistematicamente, os passos Formulação de Perguntas, Seleção de Fontes, Seleção de Estudos e Extração de Informações. Seguidamente, implementou-se um sistema para analisar permissões, *call graphs* e vazamentos de dados sensíveis do usuário de aplicativos Android. Também, apresenta-se a *Entropia de Shannon*, para cálculo de entropia de arquivos. Com o arquivo *dataset* pronto, selecionou-se as melhores *features* para treinamento dos classificadores binários escolhidos. Além disso, empregou-se uma técnica de redução de dimensionalidade para ver o *dataset* em um gráfico. Finalmente, analisou-se o desempenho dos modelos implementados por meio de parâmetros de avaliação.

3.2.1 Protocolo SLR

Para realização da SLR, adaptou-se o procedimento proposto por Senocak (2019), e ilustrado na figura 5.

Figura 5 – Etapas da SLR.



Fonte: (SENOCAK, 2019).

3.2.1.1 Formulação de Perguntas

Inicialmente, questionava-se a existência das técnicas e ferramentas para tratar problemas da área de Análise de *Malware* em dispositivos Android. Dessa forma, esperava-se compreender quais técnicas e ferramentas são mais adequadas para tratar *malwares* de Android.

Entretanto, não há, atualmente, um compilado com as técnicas e ferramentas do estado da arte do período de 2015 a 2021. Logo, propuseram-se as seguintes questões:

- "Quais as técnicas existentes para lidar com *malwares* de Android?";
- "Quais as ferramentas existentes para lidar com *malwares* de Android?";
- "Quais os tipos de *malwares* existentes no Android?";

Para responder a tais perguntas, identificaram-se como palavras-chave e sinônimos os seguintes:

- Palavras-chave: Android; *Malware Analysis*;

- Sinônimos de *Malware Analysis*: *malicious code analysis*; *malicious software analysis*;

Por meio da busca pelas técnicas e ferramentas para analisar *malwares* de Android, pretendia-se a categorização e compilação como resultados esperados. Por fim, a população do estudo é composta por publicações conduzidas por pesquisadores ou profissionais que realizam análise de *malwares* de Android.

3.2.1.2 Seleção de Fontes

Selecionaram-se fontes com publicações acerca de análise de *Malware* em dispositivos Android. Optou-se pela língua inglesa como língua padrão, visto que há alta disponibilidade de estudos publicados, abertamente, pela comunidade internacional. Para buscar tais estudos, recorreu-se a uma *string* de busca e às seguintes bases de dados: *Google Scholar*, *ACM*, *ScienceDirect* e Periódicos CAPES (CAFe).

Em primeira instância, utilizou-se o Google Scholar para realizar um teste de sanidade com a primeira proposta de consulta. Em todas as ferramentas de buscas, limitou-se o escopo de publicações entre os anos de 2015 a 2021. Foram excluídas as citações nos filtros de busca do *Google Scholar*. A tabela 1 lista a consulta e o número de resultados retornados pela ferramenta. No entanto, tais publicações não foram usadas na seleção de estudos.

Em segunda instância, parametrizou-se a consulta inicial para cada uma das bases de dados restantes. Na base de dados *ACM*, considerou-se a opção *Research Article* marcada. No ambiente da *ScienceDirect*, escolheu-se o filtro da área de Ciência da Computação, e as opções *Research Article* e *Open Access & Open Archive*. Por meio do site Periódicos CAPES (CAFe), restringiu-se a consulta com os seguintes filtros adicionais: filtro *Computer Science* marcado, com os termos da consulta contidos em qualquer campo. A tabela 1 lista as fontes, as consultas parametrizadas e o número de resultados retornados por cada fonte.

3.2.1.3 Seleção de Estudos

Posto que as bases de dados escolhidas forneciam acesso a artigos de língua inglesa, realizou-se um procedimento sistemático para seleção dos estudos primários da SLR.

Primeiramente, realizou-se a leitura do título e do resumo. Em seguida, leu-se a conclusão. Por conseguinte, leram-se a introdução e a metodologia. Por fim, leu-se o artigo por completo, para extrair as informações.

Vale ressaltar que em cada etapa do procedimento de seleção, manteve-se o estudo corrente para próxima classificação quando havia dúvida se o texto responderia a alguma questão da pesquisa ou não. Ao final da seleção, 60 artigos foram compilados na tabela 2, com as informações extraídas de cada um.

Tabela 1 – Consultas parametrizadas

Fonte	Consulta	Nº de resultados
Google Scholar	"android"AND (("malware"OR ("malicious"AND ("code"OR "software"))) AND "analysis") AND "tools"AND "techniques"	17.400
ACM	[All: "android"] AND [[All: "malware"] OR [[All: "malicious"] AND [[All: "code"] OR [All: "software"]]]] AND [All: "analysis"] AND [All: "tools"] AND [All: "techniques"] AND [Publication Date: (01/01/2015 TO 12/31/2021)]	239
ScienceDirect	"android"AND (("malware"OR ("malicious"AND ("code"OR "software"))) AND "analysis") AND "tools"AND "techniques"	136
Periódicos CAPES (CAFe)	"android"AND (("malware"OR ("malicious"AND ("code"OR "software"))) AND "analysis") AND "tools"AND "techniques"	56

Fonte: o próprio autor.

3.2.1.4 Extração de Informações

Nesta etapa da SLR, buscou-se pelas técnicas e ferramentas utilizadas ou produzidas nos estudos primários. Sendo assim, enumerou-se quais técnicas e ferramentas foram usadas, criadas ou mencionadas no estudo; seu título; seus autores e seu ano de publicação.

3.2.2 Análises Estáticas

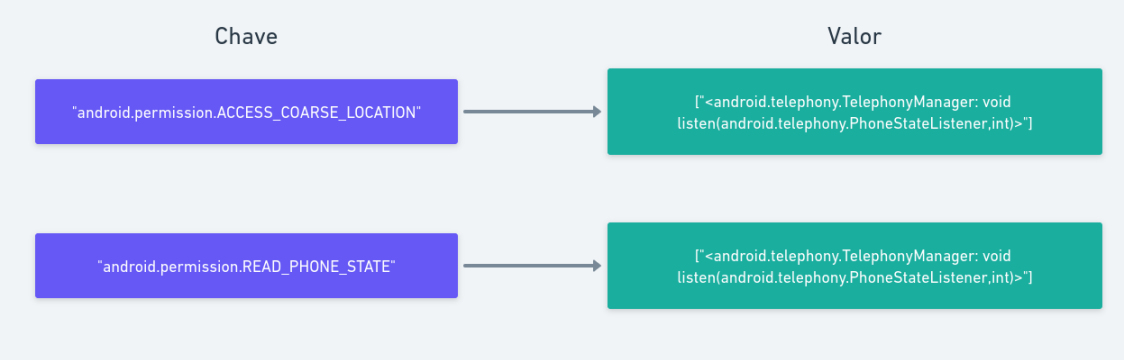
Para analisar, estaticamente, aplicativos Android, empregaram-se as técnicas: *Permission Based Analysis*, *Call Graph Analysis* e *Taint Analysis*.

3.2.2.1 *Permission based Analysis*

O sistema *SFEDroid* recebe um APK como entrada e o desempacota, para analisar seu arquivo de manifesto. Durante esse processo, destacam-se as permissões. Após identificá-las, elas são utilizadas como chaves para consultar quais métodos o aplicativo pode empregar em suas atividades, com base no mapeamento permissão-métodos fornecido por Backes et al. (2016).

A cada execução do programa *SFEDroid*, é carregada uma estrutura de dados baseada em pares chave-valor. As chaves correspondem às assinaturas de permissões, e os valores consistem em listas de assinaturas de métodos, conforme exemplificado na figura 6. No entanto, as assinaturas dos métodos listadas no repositório Axplorer possuem uma sintaxe diferente em relação à representação interna da ferramenta *FlowDroid*, que utiliza a sintaxe de *SootMethod*. Para abordar essa discrepância, realiza-se a tradução das assinaturas *Axplorer* para o formato *SootMethod* por meio de expressões regulares, conforme ilustrado no diagrama da figura 7.

Figura 6 – Exemplo de pares chave-valor.



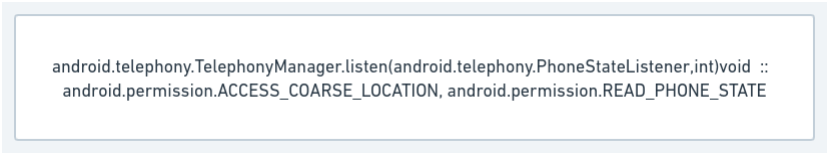
Fonte: o próprio autor.

Figura 7 – Tradução entre sintaxes de assinaturas de métodos.

Fonte: o próprio autor.

As figuras 6 e 7 são exemplos da representação interna da ferramenta *SFEDroid*, quando é fornecido como entrada o exemplo de mapeamento *Axplorer* da figura 8.

Figura 8 – Exemplo de mapeamento *Axplorer*.



Fonte: o próprio autor.

Em suma, o repositório *Axplorer* separa os conjuntos de mapeamentos permissão-métodos, em pastas intituladas com o número da versão da API do Android. Porém, a ferramenta *SFEDroid* não distingue mapeamentos idênticos que podem ter sido listados em mais de uma versão da API, isto é, mapeamentos que tenham uma mesma chave na estrutura de dados final. Logo, agrega-se todas as permissões e métodos possíveis da versão 16 a 25 da API do Android, que correspondem às versões 4.1 e 7.1 do sistema, respectivamente.

3.2.2.2 *Call Graph Analysis*

Empregaram-se capacidades da ferramenta *FlowDroid* para analisar o *Call Graph* dos aplicativos, a fim de encontrar métodos alcançáveis partindo de um *Entry Point* deles. Diferentemente dos CFGs, definidos na seção 2.1, o grafo de chamadas é composto por nós que representam métodos em um bloco básico e arestas direcionadas que indicam qual método é chamado no escopo do outro. Dessa forma, é possível encontrar caminhos entre métodos que eventualmente o usuário pode executar sem consentimento, por exemplo.

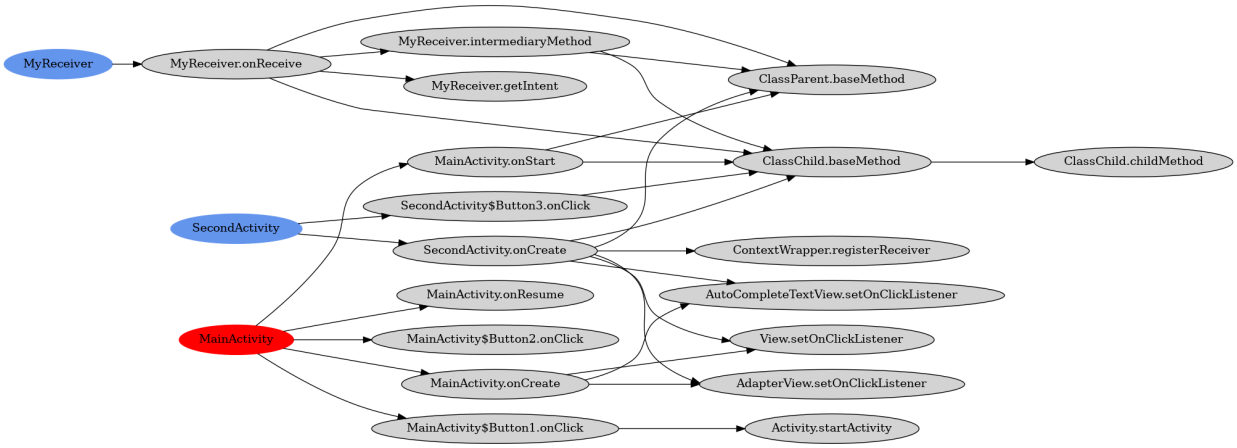
A ferramenta *FlowDroid* fornece uma interface para criadores de *Entry Points*. Um criador de *Entry Point* gera um método principal fictício (também chamado de *dummy method*), que serve como um ponto de entrada artificial para a construção de um grafo de chamadas. Este método pode instanciar as classes alvo corretas a serem analisadas e chamar os métodos de instância desejados na ordem correta. O método principal fictício toma o lugar do método *main*() de classes Java, com o intuito de sempre fornecer um ponto de entrada para a análise de atividades (ARZT, 2017).

A análise de *call graph* realizada emprega o algoritmo CHA (*Class Hierarchy Algorithm*), para descobrir quais métodos são chamados a partir de um método inicial de uma classe. Esse algoritmo é rápido e, por ser conservativo, não deixa de criar nós entre métodos chamados explicitamente (DEAN; GROVE; CHAMBERS, 1995).

O *framework FlowDroid* implementa outros métodos para analisar o *Call Graph* de um programa, mas escolheu-se o CHA, pois ele apresentava mais métodos alcançáveis

pelos *Entry Points* dos aplicativos. A figura 9 ilustra um exemplo de *call graph* impresso, o nó vermelho representa a atividade principal do app, os nós azuis são outras atividades do app e os nós cinzas são os métodos de classes chamados a partir de cada atividade.

Figura 9 – Exemplo de *call graph* gerado pela *SFEDroid*.



Fonte: o próprio autor.

Portanto, dado um *Entry Point* da aplicação, é possível identificar métodos da API do Android, alcançáveis em atividades, que foram concedidos acesso por alguma permissão.

3.2.2.3 *Taint Analysis*

A análise estática baseada em Análise de Fluxo de Dados usada é a *Taint Analysis*. Essa forma de análise assume que existem métodos que são fontes (ou *sources*) e métodos que são sumidouros (ou *sinks*). A ideia é que dados sensíveis do usuário são coletados no aparelho pelos *sources* e vazados pelos métodos *sinks*. A técnica é implementada pela *FlowDroid* por meio do cálculo de caminhos de acesso (ou *access paths*) entre os *sources* e *sinks* (ARZT, 2017). Para indicar quais métodos são de cada tipo, emprega-se um mapeamento em arquivo de texto, em que cada tipo é anotado para rotular os métodos.

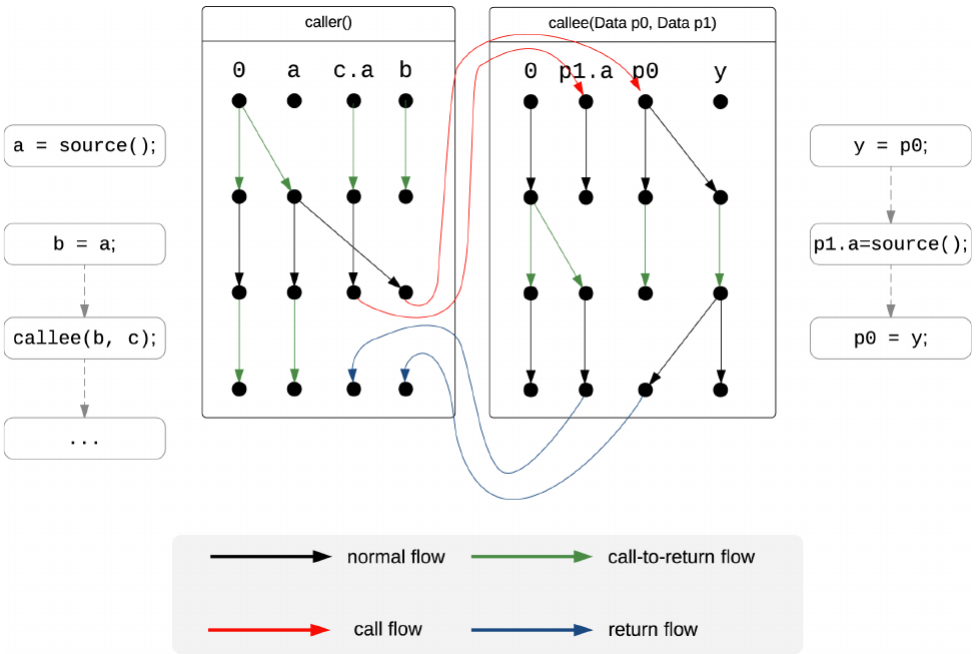
Assim como a *Liveness Analysis*, explicada na seção 2.1, conjuntos de fluxo de dados são definidos para solucionar o problema de identificar vazamento de dados sensíveis do usuário em aplicativos Android. Entretanto, os conjuntos use e def são chamados de gen e $kill$, respectivamente. Isso posto, de acordo com Arzt (2017), quando o fluxo de dados se propaga em caminhos distintos no CFG, deve-se uní-los. Os conjuntos in e out são inicializados vazios e a direção da propagação do fluxo é *forward*.

Apesar da definição da direção de propagação do fluxo de dados, Arzt (2017) argumenta que tanto a direção *forward* como *backward* são usadas pela sua implementação, a depender de qual função de fluxo é usada. Consequentemente, quatro funções de fluxo são definidas para tratar tipos distintos de conexões entre declarações no código. Em conformidade com Arzt (2017), tem-se:

- **Função de Fluxo Normal (*normal flow*)**: aplicada em declarações que não são chamadas nem retornos, comuns em atribuições e condicionais;
- **Função de Fluxo de Chamada (*call flow*)**: modela chamadas de método, mapeando argumentos de chamada para parâmetros do método no *callee*;
- **Função de Fluxo de Retorno (*return flow*)**: aplicada na saída de métodos, mapeando parâmetros e valores de retorno do *callee* de volta para o *caller*;
- **Função de Fluxo de Chamada para Retorno (*call-to-return flow*)**: usada em chamadas para preservar informações sobre variáveis fora do escopo do *callee*.

Para ilustrar os tipos de funções de fluxo, adaptou-se o diagrama de Arzt (2017), na figura 10. Nela, a análise começa na função *caller()*. Assume-se zero como ponto de partida do fluxo e modela-se a variável a a partir da função de fluxo de chamada para retorno, dado o método *source()*. Em seguida, o valor de a é copiado para b, pela função de fluxo normal. Na linha seguinte do código, invoca-se o método *callee()* e os valores de a e b são propagados por meio da função de fluxo de chamada. Quando o fluxo de execução encontra-se no escopo do método *callee()*, $p1.a$ e $p0$ são inicializados com os valores de $c.a$ e b, respectivamente. Na sequência, $p0$ é copiado para y; $p1.a$ é sobrescrito pelo valor de retorno do método *source*(); e $p0$ é sobrescrito pelo valor de y. Por fim, os dados de $p1.a$ e $p0$ são propagados, respectivamente, para $c.a$ e b pela função de fluxo de retorno. A expressão $c.a$ é colocada na figura para ilustrar um caso em que tal variável não recebe propagação de fluxo, enquanto o *callee* não propaga o valor de $p1.a$.

Figura 10 – Exemplo de funções de fluxo.



Fonte: (ARZT, 2017).

A figura 11 exemplifica como um dado pode ser vazado no método *onCreate()*, de uma atividade de um aplicativo. Assume-se que o método *getDeviceId()* é um *source* e o método *sendTextMessage()* é um *sink*. Neste exemplo, portanto, é vazado o identificador do dispositivo, como IMEI, por SMS.

Figura 11 – Exemplo de vazamento de dados sensíveis do usuário.

```
1 void onCreate() {
2     // Get the data
3     TelephonyManager mgr = (TelephonyManager) this.getSystemService(TELEPHONY_SERVICE);
4     String deviceId = mgr.getDeviceId();
5
6     // Leak the data
7     SmsManager sms = SmsManager.getDefault();
8     sms.sendTextMessage("+49 1234", null, deviceId, null, null);
9 }
```

Fonte: (ARZT, 2017).

Outrossim, os chamados caminhos de acesso são abstrações que representam acessos a um mesmo objeto na *heap* de um programa Java. Essa tarefa é modelada pela *FlowDroid* por meio da técnica intitulada *Alias Analysis*. Basicamente, a técnica consiste em assumir um caminho de acesso e em seguida enumerar todos os outros caminhos de acesso que podem apontar para o mesmo objeto na memória (ARZT, 2017).

O código da figura 12 demonstra diversas formas de acessar um mesmo campo de dados, de uma estrutura de dados do tipo lista duplamente ligada. Posto que podem existir diversos caminhos de acesso para uma mesma variável, é estabelecido um tamanho máximo k para computá-los, na ferramenta *FlowDroid*.

Figura 12 – Exemplo de caminhos de acesso em lista duplamente ligada.

```
1  void onCreate() {
2      A a = new A();
3      a.next = new A();
4      a.next.prev = a;
5      a.data = source();
6      sink(a.data); // leak
7      sink(a.next.prev.data); //leak
8      sink(a.next.prev.next.prev.data); // leak
9      sink(a.next.prev.foo);  // no leak
10 }
```

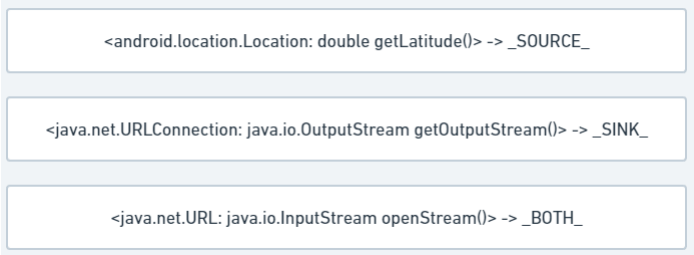
Fonte: (ARZT, 2017).

Em suma, a *Taint Analysis* utilizada, analisa o fluxo de dados entre funções para detectar vazamentos de dados sensíveis de usuário. Faz isso por meio de funções de fluxo e caminhos de acesso. Porém, para realizar esses cálculos, é necessário identificar, *a priori*, quais métodos da API do Android são *sources*, *sinks* ou ambos, simultaneamente. Existem algumas abordagens automáticas para cumprir essa tarefa, mas fogem do escopo

deste trabalho. Por último, ressalta-se que a listagem usada na implementação deste trabalho é aquela fornecida pela ferramenta *FlowDroid*, pois seus métodos representam fontes e sumidouros que vazam dados sensíveis de usuário, mas, essas definições podem ser especializadas em outros contextos para detectar propagação de dados no código de aplicações.

A imagem 13 mostra um exemplo de mapeamento na sintaxe usada no documento de definições de *sources*, *sinks* e *both*.

Figura 13 – Exemplos de mapeamentos *source*, *sink* e *both* no arquivo de texto.



Fonte: o próprio autor.

3.2.3 Cálculo de Entropia

Uma das *features* escolhidas para compor o *dataset* montado, é a entropia do arquivo *classes.dex* de um app. Como elucidado, esse arquivo contém os *bytecodes* de todos os códigos de um aplicativo. Em concordância com Hartman (2013), a entropia de Shannon é uma medida fundamental na teoria da informação que quantifica a incerteza ou a imprevisibilidade de uma fonte de informação. No contexto de arquivos, a entropia de Shannon é usada para calcular a quantidade de informação ou desordem presente em um conjunto de dados. Quanto maior a entropia, mais imprevisível e mais informação é contida nos dados.

No contexto de compressão de dados, um arquivo é comprimido substituindo padrões de bits por padrões mais curtos de bits. Portanto, quanto maior a entropia no arquivo de dados, menos ele pode ser comprimido mais ainda. Determinar a entropia de um arquivo também é útil para detectar se ele provavelmente está criptografado (HARTMAN, 2013).

Implementou-se o pseudo-código 2 para calcular a entropia de Shannon do arquivo *classes.dex* de aplicativos Android.

Em conclusão, a entropia de um arquivo indica o grau de imprevisibilidade de seus dados e auxilia na identificação de criptografia e compressão em uma amostra. Essa é uma heurística usada comumente por analistas de *malware* identificarem tais tipos de amostras quando o valor da entropia ultrapassa sete (KIM; KWAK; RYOU, 2015).

Algorithm 2 Cálculo da Entropia de Shannon

```
1: function SHANNON_ENTROPY(file_path)
2:     Inicialize um dicionário vazio bytes_counts
3:     Inicialize total_bytes como zero
4:     abrir o arquivo file_path para leitura
5:     while houver bytes não lidos no arquivo do
6:         byte ← próximo byte do arquivo
7:         bytes_counts[byte] ← bytes_counts[byte] + 1
8:         total_bytes ← total_bytes + 1
9:     end while
10:    Inicialize uma lista vazia probabilities
11:    for cada byte c em byte_counts do
12:        prob ← byte_counts[c]/total_bytes
13:        if prob > 0 then
14:            Adicione prob à lista probabilities
15:        end if
16:    end for
17:    Inicialize entropy como zero
18:    for cada probabilidade p em probabilities do
19:        entropy ← entropy − (p · log2(p))
20:    end for
21:    retornar entropy
22: end function
```

3.2.4 Seleção de *features*

Antes de treinar os classificadores binários escolhidos, selecionaram-se *features* por meio do cálculo de variância e pelo ganho de informação. Ambas técnicas foram implementadas por meio da biblioteca *sklearn* em Python.

3.2.4.1 *Features* de zero-variância

Inicialmente, havia 5.174 *features*, no entanto, buscou-se remover aquelas com variância zero. Como resultado, as colunas do *dataset* que mantinham o mesmo valor em todas as linhas foram excluídas, resultando em um total de 515 *features*.

3.2.4.2 *Information Gain*

O objetivo era criar um *dataset* com menos de 515 *features* a fim de comparar o desempenho dos classificadores. Portanto, as 100 melhores *features* dentre as 515 foram selecionadas utilizando o cálculo do *Information Gain*.

A seleção de *features* com *Information Gain* é um método que avalia a relevância das características em um conjunto de dados para tarefas de classificação ou previsão. Esse processo se baseia na teoria da entropia, que quantifica a impureza nos dados, e no *Information Gain*, que mede a redução na entropia ao utilizar uma determinada *feature*

para dividir os dados. As *Features* com maior *Information Gain* são consideradas mais importantes, ao contribuírem significativamente para a diminuição da incerteza nos dados. Portanto, a seleção de *features* com *Information Gain* envolve calcular o *Information Gain* de cada *feature* em relação às classes alvo, classificá-las com base nessa métrica e, em seguida, selecionar as *features* mais informativas para a construção de modelos de aprendizado de máquina. Isso ajuda a simplificar e otimizar o processo de treinamento do modelo, mantendo a relevância das características mais significativas (RUSSELL; NORVIG, 2021).

Como resultado, conduziu-se um segundo treinamento com um *dataset* composto por 100 *features* e 10.000 amostras.

3.2.5 Classificadores binários

A seguir, o conceito de cada classificador binário escolhido para treinamento é resumido. Ressalta-se que a escolha dos classificadores *Decision Tree* (DT), *Random Forest* (RF), *AdaBoost*, *Naive Bayes* (NB) e SVM, deve-se à sua ampla utilização pelos autores dos estudos primários da SLR conduzida. Dos 60 artigos lidos, 14 empregaram o modelo DT, 19 usaram o RF, 13 treinaram o classificador NB e 29 utilizaram o SVM (com funções de kernel variadas). Além disso, para avaliar o desempenho de um algoritmo baseado em *boosting*, optou-se por treinar o classificador *AdaBoost*.

3.2.5.1 *Decision Tree*

Uma Árvore de Decisão é um modelo de classificação que opera semelhantemente a um fluxograma. Ela divide o espaço de atributos em subconjuntos menores por meio de decisões baseadas em critérios, buscando classificar as amostras em classes distintas. O processo de divisão é guiado por características específicas e seus valores. As Árvores de Decisão são facilmente interpretáveis e podem ser profundas (com muitas divisões) ou rasas (com poucas divisões), dependendo da complexidade do problema (RUSSELL; NORVIG, 2021).

3.2.5.2 *Random Forest*

Random Forest é uma extensão das Árvores de Decisão que combina múltiplas árvores para reduzir a variância e aumentar a precisão. Ela opera por meio do processo de *bootstrapping*, criando várias amostras aleatórias com reposição a partir dos dados originais e, em seguida, construindo várias Árvores de Decisão. Os resultados das árvores individuais são agregados para obter uma decisão final, tornando o modelo mais robusto e preciso (RUSSELL; NORVIG, 2021).

3.2.5.3 *Adaboost*

O *Adaboost* (*Adaptive Boosting*) é um algoritmo de *boosting* que combina múltiplos classificadores fracos para criar um classificador forte. Ele atribui pesos às amostras incorretamente classificadas, incentivando o modelo a focar nas amostras mais difíceis. A cada iteração, *Adaboost* ajusta os pesos e constrói um novo classificador fraco. Os classificadores fracos são ponderados e combinados para formar o classificador final (FREUND; SCHAPIRE, 1995).

3.2.5.4 *Naive Bayes*

O *Naive Bayes* é um classificador probabilístico que se baseia no Teorema de Bayes. Ele assume independência condicional entre as características, daí o termo *naive* (ingênuo). Mesmo com essa simplificação, o *Naive Bayes* é eficaz em muitos cenários de classificação, especialmente em tarefas de processamento de linguagem natural e classificação de documentos. Ele calcula a probabilidade de uma amostra pertencer a uma classe específica com base nas probabilidades condicionais das características (RUSSELL; NORVIG, 2021).

3.2.5.5 SVM (RBF - *Radial Basis Function*)

As Máquinas de Vetores de Suporte (SVM) são classificadores que buscam encontrar a melhor separação entre duas classes, criando uma margem máxima entre os pontos de dados. Por meio da utilização da função de kernel RBF, os dados são mapeados para um espaço de características de alta dimensão e uma fronteira de decisão não linear é criada. O RBF é eficaz em problemas onde a relação entre as características e as classes não é linear e pode capturar relações complexas (RUSSELL; NORVIG, 2021).

3.2.6 Visualização de dados

Reduziu-se a dimensão do *dataset* de 10.000 amostras e 515 *features* por meio de PCA, para visualizar os pontos de dados em um gráfico de dispersão.

O PCA (do inglês *Principal Component Analysis*), é uma técnica para simplificar conjuntos de dados de alta dimensionalidade. Isso é alcançado encontrando os componentes principais, os quais são vetores ortogonais que representam as direções de máxima variação nos dados. Esses componentes são ordenados por importância, com o primeiro componente capturando a maioria da variação. O PCA utiliza os conceitos de autovalores e autovetores da matriz de covariância dos dados para realizar essa transformação. Ao projetar os dados originais nos componentes principais, cria-se um novo conjunto de dados de menor dimensionalidade, preservando a maioria da variância original. Isso é fundamental em aplicações de IA, incluindo redução de dimensionalidade, análise de padrões e compressão

de dados, permitindo uma melhor compreensão e processamento de informações complexas (RUSSELL; NORVIG, 2021).

3.2.7 Parâmetros de Avaliação

Para avaliar o desempenho dos classificadores implementados, tabulou-se os resultados das métricas: Acurácia, *F-measure*, MCC, curva AUC e 10-fold *cross-validation*. A lista a seguir define, matematicamente, cada uma dessas medidas.

- **Acurácia**: é a proporção de instâncias verdadeiras-positivas (VP) e verdadeiras-negativas (VN), dividida pelo número total de instâncias, incluindo falsos-positivos (FP) e falsos-negativos (FN). É calculada como mostrado na equação 3.1;

$$Acurácia = \frac{VP + VN}{VP + FP + FN + VN} \tag{3.1}$$

- ***F-measure***: representa a média harmônica de recall e precisão. É calculado como mostrado na equação 3.4;

$$Precision = \frac{VP}{VP + FP} \tag{3.2}$$

$$Recall = \frac{VP}{VP + FN} \tag{3.3}$$

$$F - measure = \frac{2 \times (Precision \times Recall)}{(Precision + Recall)} \tag{3.4}$$

- **MCC**: usado para medir a qualidade de algoritmos de classificação binária. Seu valor varia de -1 a +1. O valor -1 significa uma previsão inversa, enquanto +1 significa uma previsão perfeita. É calculado como mostrado na equação 3.5;

$$MCC = \frac{VP \times VN - FP \times FN}{\sqrt{(VP + FN)(VP + FP)(VN + FP)(VN + FN)}} \tag{3.5}$$

- **Curva AUC**: representa a medida da separabilidade. Um modelo excelente tem uma AUC próxima de 1, o que significa que possui uma boa medida de separabilidade. Um modelo fraco tem uma AUC próxima de 0, o que significa que possui a pior medida de separabilidade;

- **10-fold *cross-validation***: métrica que separa o conjunto de dados em conjuntos menores, chamados *folds*, e rotaciona-se o treinamento e a validação entre eles. O número de *folds* escolhido é 10. Sendo assim, calcula-se a média das acurácias dos 10 *folds*, bem como seu desvio padrão.

4 Resultados e Discussões

Após a conclusão da fase de Extração de Informações da SLR, os resultados foram sintetizados em tabelas e análises quantitativas realizadas. Assim, obteve-se o número total de estudos primários por ano de publicação, além de tabulações dos estudos primários lidos na íntegra. Essas tabulações incluem classificações das técnicas e ferramentas utilizadas para analisar *malwares* em dispositivos Android. Adicionalmente, foram detalhados os tipos de *malwares* abordados nos estudos primários, bem como as técnicas *anti-análise de malware* mais recorrentes nesses estudos.

Além disso, detalhou-se a implementação do sistema criado, visando elucidar como algumas técnicas e ferramentas podem ser empregadas para tentar resolver o problema de classificação binária de aplicativos maliciosos. Ressalta-se que alguns estudos primários lidaram com o problema de classificar *malwares* de acordo com sua família, como realizaram Dhalaria e Gandotra (2020), isto é, uma classe de códigos maliciosos com um comportamento muito similar. No entanto, este é um problema de classificação multi-classe e não é tratado neste trabalho.

4.1 Estudos Primários

O gráfico da figura 14 mostra o total de estudos primários publicados em cada ano, entre 2015 e 2021. Ainda, tais estudos foram compilados na tabela 2.

Figura 14 – Total de estudos primários por ano.

Fonte: o próprio autor.

Tabela 2 – Estudos primários.

Técnica(s)	**Ferramenta(s)**	**Título (Referência)**
Taint Analysis, Dataflow Analysis, Entry Point Analysis; Instrumentation; System call tracing; Debugging; Code Emulation; Mandatory Access Control; Discretionary Access Control; Role Based Access Control; Context Based Access Control; Attribute Based Access Control; Targeted analysis; Dynamic Taint tracking;	IccTA; apktool; dex2jar; Amandroid; FlowDroid; Epicc; DidFail; Soot; ComDroid; IntelliDroid; APKParser; Dare; Z3 constraint solver; TaintDroid; IntentDroid; WALA; DIALDroid; APKCombiner; FlaskDroid; XmanDroid; dedexer; Ded; Jimple (IR); Smali (IR);	Android inter-app communication threats and detection techniques (BHANDARI et al., 2017)
Manifest based analysis, Permission based analysis; code emulation; sandboxing;	ScanMe Mobile; DroidBox; apktool; Androguard; jd-cmd; dex2jar; Matlab; Datasets (Android Malware Genome Project);	ScanMe mobile: A local and cloud hybrid service for analyzing apks (COLE et al., 2015)
Análise Estática; SVM; LR; RF; KNN; NB; DT; RNN;	Datasets (CICAndMal2017, DREBIN, CICInvesAndMal2019, AndroidBotnet2015, CIC-MalDroid2020); Androguard;	Redroiddet: Android malware detection based on recurrent neural network (ALMAHMOUD; ALZU'BI; YASEEN, 2021)
Análise Estática; J48; BayesNet; RF; LogisticModelTrees; JRip; DecisionTable; KNN; CNN; Model Checking;	SigMal; WEKA;	Android collusion detection by means of audio signal analysis with machine learning techniques (CASOLARE et al., 2021)
System call Interposition	MOSES; AirBag; MockDroid; FireDroid; NJAS; MBOX; Cells; Boxify; AppGuard; I-ARM-Droid; RetroSkeleton; Dr. Android; Aurasium;	NJAS: Sandboxing unmodified applications in non-rooted devices running stock android (BIANCHI et al., 2015)
Similarity based analysis	smali (IR); AndroGuard; VirusTotal	Android Applications Repackaging Detection Techniques for Smartphone Devices (RASTOGI; BHUSHAN; GUPTA, 2016)

Continua na próxima página

Tabela 2 – continuação da tabela

Técnica(s)	**Ferramenta(s)**	**Título (Referência)**
Manifest based analysis; SVM; KMC; MM;	Mobile Sandbox; TargetDroid; ANANAS; Marvin; Draco; SCREDENT; Droidbox; Dataset (Contagio mobile malware minidump, DREBIN);	SCREDENT: Scalable Real-time Anomalies Detection and Notification of Targeted Malware in Mobile Devices (MCNEIL et al., 2016)
Dataflow analysis; String analysis; Signature-based Analysis; Code dependency Analysis; String-taint analysis; Dynamic string-taint analysis; Dynamic symbolic execution;	NetworkProfiler; Autograph; EarlyBird; PolyGraph; Hamsa; Stowaway; Kirin; TaintDroid; Apex; TISSA; PiOS;	NTApps: A Network Trafic Analyzer of Android Applications (RODRIGUEZ; MOSTAFA; WANG, 2017)
Signature-based detection; Anomaly-Based detection; SL, NB, BayesNet; SMO; IBK; J48; RF; KNN; SVM; PART; ROT; Random Committee; C4.5; Information flow analysis;	Kirin; DroidMat; PUMA; k-map; SIGPID; MalPat; Andro-AutoPsy; Mal-warehouse; MIET; ScanMe mobile; DroidNative; MAIL;	A Survey on Mobile Malware Detection Techniques (KOULIARIDIS et al., 2020)
System call tracing; Taint analysis; CNN; RNN;	Datasets (Drebin, Genoma Project, Contagio); ADB (monkey tool); strace; CopperDroid;	Evaluating Convolutional Neural Network for Effective Mobile Malware Detection (MARTINELLI; MARULLI; MERCALDO, 2017)
Signature-based analysis; fuzzy hashing; N-grams; Similarity based analysis;	ssdeep; mvHash-B; dcfldd; sdhash; mrshv; APK-DNA; ROAR; Datasets (Android Malware Genome; Drebin); xxd; Android SDK (dexdump; aapt); DroidMOSS; Juxtapp; ANDRUBIS; DroidSIFT; DroidAPIMiner	Fingerprinting Android packaging: Generating DNAs for malware detection (KARBAB; DEBBABI; MOUHEB, 2016)
Análise dinâmica; HoeffdingTree; SVM; KNN;	Datasets (Drebin, HelDroid); adb; Android Emulator (do Android SDK); Monkey tool (do adb); tcpstat;	Energy consumption metrics for mobile device dynamic malware detection (FASANO et al., 2019)
Análise estática; Análise dinâmica; Behaviour based analysis; API hooking; Taint Analysis; Similarity based Analysis;	Honeyspot; Smartpot; Paranoid Android; NMAP; Apktool; Comdroid; Dex2Jar; TaintBochs; TaintDroid; DroidBox; DroidMOSS; logcat;	A study of android malware detection techniques in virtual environment (JUNG; KIM; CHO, 2016)

Continua na próxima página

Tabela 2 – continuação da tabela

Técnica(s)	**Ferramenta(s)**	**Título (Referência)**
Signature-based analysis; event-based triggering; time-based triggering; permission-based analysis; C4.5; NB; RF; SMO; Expectation-maximisation clustering;	Datasets (Malgenome; Drebin); apktool; Monkey tool; IpVoid; PScout; MobiSec Lab; ApkScan; DroidBot; DroidBox; SAFEDroid; AppGuard; Crowdroid; TaintDroid; MADAM; ComDroid; RiskRanker; Andrubis; Grab'n Run; StaDyna; StaDART; ArtDroid;	Analysis of dynamic code updating in Android with security perspective (AYSAN; SAKIZ; SEN, 2019)
Análise estática; RF; SVM; LR; KNN; Association Rule Analysis; permission-based analysis	Datasets (DREBIN, CICInvesAndMal2019); DroidAPIMiner; DroidDelver; DaDiDroid; Apktool; PScout; MaMaDroid	Android Malware Detection based on Vulnerable Feature Aggregation (ROY et al., 2020)
decompiling; debugging;	dex2jar; JAD; IDA Pro; Tracedroid; Volatility; LiME (Linux Memory Extractor); AndroGuard; LogCat; JEB decompiler;	DWroidDump: Executable Code Extraction from Android Applications for Malware Analysis (KIM; KWAK; RYOU, 2015)
Análise dinâmica; LSTM; MIL;	VizMal; Amandroid; FlowDroid; Epicc; AndroDialysis; BRIDEMAID; Andromaly; Jackdaw; TaintDroid; CopperDroid; Datasets (Genoma Project, Contagio, Drebin); Kprobes; monkey tool;	Visualizing the outcome of dynamic analysis of Android malware with VizMal (LORENZO et al., 2019)
Sandboxing; Fuzzy hashing; Dynamic taint tracking; Binary Instrumentation;	DroidScope; Andrubis; CopperDroid; VetDroid; Condroid; Anbox; Cells; QEMU; KVM; strace; CrowDroid; DroidTrace; ptrace; Frida; MockDroid; I-ARM-Droid; RiskRanker; Xposed Framework; DroidMOSS; PiggyApp; HARVERSTER; Ether; Panorama; LXC; Docker; rkt; LXD; ADB; logcat; monkey tool;	Dynamic analysis with Android container: Challenges and opportunities (CHAU; JUNG, 2018)

Continua na próxima página

Tabela 2 – continuação da tabela

Técnica(s)	**Ferramenta(s)**	**Título (Referência)**
Análise estática; Análise Dinâmica; API Hooking;	A3E; AppsPlayground; SMV-Hunter; ACTEve; IntelliDroid; ADB (Monkey); Robotium; DynoDroid; PUMA; Brahmastra; Soot; FlowDroid; EdgeMiner; Z3; Xposed Framework;	SMARTGEN: Exposing Server URLs of Mobile Apps With Selective Symbolic Execution (ZUO; LIN, 2017)
Análise estática; LR; DNN; SVM; LRLR; RF; NB; DT; KNN; MC;	axplorer; PScout; PROTEUS; DroidMiner; Droid Analyst Combo; StormDroid; DroidSieve; MAMADROID; RiskInDroid; DroidCat; dex2jar; Tensorflow;	Android Fragmentation in Malware Detection (NGUYEN-VU; AHN; JUNG, 2019)
Análise estática; Análise Dinâmica; Signature-based analysis; Behaviour based analysis; N-gram; Active learning; RF; SVM; KNN; NB; Affinity propagation;	RobotDroid; DroidSieve; apktool; WEKA;	TinyDroid: A lightweight and efficient model for android malware detection and classification (CHEN et al., 2018)
Análise estática	dex2jar; jd-gui; apktool	Spyware Detection in Android Using Hybridization of Description Analysis, Permission Mapping and Interface Analysis (KAUR; SHARMA, 2015)
Análise estática; NB; (Linear) SVM; DT; KNN; RF; MIL; Online Learning Algorithm; PART; J48; MLP; SL; DAE; CNN; DBN; k-star; GKMC;	Cuckoo sandbox; Strace; IDAPro; Captone; Datasets (Drebin);	Android malware detection method based on bytecode image (DING et al., 2020)
Análise Estática; Análise Dinâmica; MLP; (Linear) SVM; SVM RBF; NB; SL; PART; RF; J48; DT;	DroidBox; ApkScan; DL-Droid; Android Virtual Device (AVD); Genymotion; Monkey tool; Dynodroid; ACTEve; A3E; GUIRipper; SwiftHand; PUMA; DroidBot; Droid-Sec; DroidDetector; Deep4MalDroid; Dataset (Intel Security [McAfee Lab]); H20 flow; WEKA; AutoDroid;	DL-Droid: Deep learning based android malware detection using real devices (ALZAYLAEE; YERIMA; SEZER, 2019)

Continua na próxima página

Tabela 2 – continuação da tabela

Técnica(s)	**Ferramenta(s)**	**Título (Referência)**
Análise estática; Análise Dinâmica; CART; RF; Extremely Randomized Trees; KNN; SVM; LR; XGBoost; N-grams;	StormDroid; MalDy; Datasets (MalGenome; Drebin; Maldozer; AndroZoo; PlayDrone); DroidBox; ThreatAnalyzer; Monkey tool;	MalDy: Portable, data-driven malware detection using natural language processing and machine learning techniques on behavioral analysis reports (KARBAB; DEBBABI, 2019)
Análise Dinâmica; DT; RF; SVM; N-gram; VMI based Analysis;	QEMU; strace; ptrace; Monkey tool; WEKA; Datasets (Genome Project; Inter Component Communication Repository (IccRE); Additional Malware Families 2015; Contagio Minidump); VirusTotal; MamaDroid; DroidSieve; CrowDroid; DroidScope; CopperDroid; DroidScribe; ANDect; SWORD: FireDroid; RiskRanker;	SWORD: Semantic aWare andrOid malwaRe Detector (BHANDARI et al., 2018)
Análise Estática; KNN; KMC; SVM; DT; RF; AdaBoost;	strings; QEMU; Smali (apktool); Alligator; DroidLysis; RobotDroid; PUMA; MADAM; Mast; Andromaly; AndRadar; Andrubis; PlayDrone; DroidRanger; Datasets (Drebin); AppsPlayground;	SherlockDroid: a research assistant to spot unknown malware in Android marketplaces (APVRILLE; APVRILLE, 2015)
Análise dinâmica; LSTM; MIL;	CANDYMAN; DroidBox; ADB (monkey tool); Datasets (Drebin); Crowdroid; DroidScope; MaMaDroid; DroidDetector; DroidDelver; Deep4MalDroid; logcat; Cuckoo Sandbox; ProcMon; MALHEUR; Androguard; scikit-learn library; Keras Framework; TensorFlow; Theano;	CANDYMAN: Classifying Android malware families by modelling dynamic traces with Markov chains (MARTíN; RODRíGUEZ-FERNáNDEZ; CAMACHO, 2018)
Análise estática; Análise Dinâmica; SVM; ANN; DT; NB; KNN; MLP;	DroidDolphin; LIBSVM; Datasets (Android Genome Project); WEKA; Androguard; Scikit-learn framework; Apktool;	Framework for malware analysis in Android (URCUQUI-LóPEZ; CADAVID, 2016)

Continua na próxima página

Tabela 2 – continuação da tabela

Técnica(s)	**Ferramenta(s)**	**Título (Referência)**
Control flow analysis; Dataflow analysis; Points-to analysis; Symbolic execution; Taint analysis; Análise Dinâmica;	JITANA; EPICC; IC3; SOOT; SIFTA; APKCOMBINER; EMMA; Amandroid; Monkey tool (adb); SPARK; Apktool; ICCTA; FlowDroid; DIDFAIL; SEALANT;	Jitana: A modern hybrid program analysis framework for android platforms (TSUTANO et al., 2018)
Análise estática	RAPID; Apktool; Dex2jar; Ded; Dare; Androguard; IDA Pro; JEB; AIS: Dedexer; Soot; Jad; JD-GUI; Dataset (Android Malware Genome Project);	Rapid Android Parser for Investigating DEX files (RAPID) (ZHANG; BREITINGER; BAGGILI, 2016)
Análise estática; ANN; word embedding;	Datasets (Malgenome; Drebin; MalDozer dataset; virusshare; Contagio Minidump); dexdump; TensorFlow; Playdrone;	MalDozer: Automatic framework for android malware detection using deep learning (KARBAB et al., 2018)
Análise estática; Análise dinâmica; Rule-based Analysis;	Soot; Dare; Dataset (VirusShare, Drebin); SCanDroid; DroidAPIMiner; AndroidLeaks; SCANDAL; RiskRanker; DroidSIFT; Andromaly; Crowdroid; DroidRanger;	Profiling user-trigger dependence for Android malware detection (ELISH et al., 2015)
Análise dinâmica; ANN; DT;	PowerTutor	Seeing the Unseen: Revealing Mobile Malware Hidden Communications via Energy Consumption and Artificial Intelligence (CAVIGLIONE et al., 2015)
Control flow analysis; Dataflow analysis; rule-based detection; KMC; Bayesian Model;	Soot (e FlowDroid); JADX; Datasets (Malware Genome Project; Drebin; AndroZoo); CopperDroid; SmartDroid; CrowDroid; PUMA; MAMA; Andromaly; DroidRanger; Droid Detective; MONET; AV-class; Euphony; Apposcopy; DroidAnalytics; DroidAPIMiner; MADAM;	Self-hiding behavior in Android apps: detection and characterization (SHAN; NEAMTIU; SAMUEL, 2018)

Continua na próxima página

Tabela 2 – continuação da tabela

Técnica(s)	**Ferramenta(s)**	**Título (Referência)**
Control flow analysis, Symbolic Data Flow Analysis; Bytecode Instrumentation;	DroidAntiRM; IntelliDroid; Dynodroid; Condroid; PScout; SuSi; Dare; Apktool; WALA; FlowDroid; EdgeMiner; Epicc; Soot; Datasets (VirusTotal, Drebin, Contagio, Malware Genome); Droidsafe; TriggerScope; TaintDroid; CopperDroid; DroidScope; BareDroid; Malton; Ninja; Harvester;	Droid-AntiRM: Taming Control Flow Anti-analysis to Support Automated Dynamic Analysis of Android Malware (WANG et al., 2017)
Análise estática	MASCAT; IDA Pro; hooper; radare;	MASCAT: Preventing Microarchitectural Attacks Before Distribution (IRAZOQUI; EISENBARTH; SUNAR, 2018)
Análise estática; KNN; SVM; ANN; RF;	Datasets (Drebin, DroidAPIMiner, Malware Genome Project; VirusShare Project); LoopMC; Androguard; AppContext; RevealDroid; FlowDroid; MAMADROID; Kirin; DroidSafe; RiskRanker; DroidRanger;	Using Loops For Malware Classification Resilient to Feature-unaware Perturbations (MACHIRY et al., 2018)
Análise estática; SVM; Natural Language Processing; Similarity Based Analysis;	MAIL; DroidNative; DroidBox; dex2jar;	Applying Natural Language Processing for detecting malicious patterns in Android applications (ALAM, 2021)
Análise Dinâmica; RF; LR; C4.5;	logcat; KidLogger; Datasets (Drebin); SCSDroid; TaintDroid; tcpdump; ps; netstat; WEKA; netem; FireEye IDS; Crowdroid; AppIntent; SmartDroid; Webprophet; Rippler; WebWitness; NetworkProfiler;	Causality-based Sensemaking of Network Traffic for Android Application Security (ZHANG; YAO; RAMAKRISHNAN, 2016)
Análise estática; Deep CNN; EML; SVM; XGBoost; MC; ROT; N-grams; Similarity based Analysis;	Malytics; Datasets (Androzoo, Virusshare, VirusTotal, DexShare, Drebin); Keras API; scikit-learn; AV-CLASS tool; MAMADROID; Apache Spark;	Malytics: A Malware Detection Scheme (YOUSEFI-AZAR et al., 2018)

Continua na próxima página

Tabela 2 – continuação da tabela

Técnica(s)	**Ferramenta(s)**	**Título (Referência)**
Análise estática; C4.5; KNN; SVM; ANN; NB; MM;	Datasets (Drebin); RevealDroid; DroidSIFT; Dendroid; DroidMiner; DroidAPIMiner; VILO; DroidLegacy; MAST; WEKA; Androguard; FlowDroid; RevealDroid*; Kirin; DroidRanger; DroidSafe; ComDroid; Epicc; Amandroid; DidFail; Apposcopy; AppContext; TriggerScope;	Picking on the family: Disrupting android malware triage by forcing misclassification (CALLEJA et al., 2017)
Análise estática; Análise dinâmica; Análise híbrida; ROT; DT; SVM; RF; LR; NB; KNN; Bagging; Stacking; Boosting; CNN; MLP; C4.5, PART; EML;	Datasets (OmniDroid, Drebin, Malgenome, Dataset 1 e 2 dos autores [disponíveis no Github e Kaggle]); CuckooDroid; Cuckoo sandbox; SigPID; DroidFusion; DroidCat; EnDroid; TaintDroid; DroidTrace; DroidDetector; AASandbox; strings; AXMLPrinter2; Xposed Framework; sklearn;	A Hybrid Approach for Android Malware Detection and Family Classification (DHALARIA; GANDOTRA, 2020)
Análise estática; Análise Dinâmica; Signature-based Analysis; Anomaly-based Analysis; SVM; CART; ANN; LSSVM; KNN; DT; Formal Concept Analysis; API hooking; Sandboxing;	DroidBox; Androguard; LIBSVM; Datasets (Contagio Blogger Web Sites, Dr. Web); Concept Explorer from Protégé; Clam-AV; VirusTotal;	Malware behavioural detection and vaccine development by using a support vector model classifier (WANG; WANG, 2015)
Análise estática;	JADX tool; RevealDroid; Droid-Sec; DroidSieve; Andronio; VirusTotal;	An in-Depth Study of the Jisut Family of Android Ransomware (MARTIN; HERNANDEZ-CASTRO; CAMACHO, 2018)
Análise estática; Cased based reasoning; Multimodal Approach; RF; SVM; DT; Decision fusion; Multi-view adversarial network; KMC;	dex2jar; Risk Ranker; Datasets(Drebin, Android Proguard, Kharon Dataset, Androzoo, CICAAGM); FAMD; WEKA;	Multimodal information fusion for android malware detection using lazy learning (QAISAR; LI, 2021)

Continua na próxima página

Tabela 2 – continuação da tabela

Técnica(s)	**Ferramenta(s)**	**Título (Referência)**
Análise estática; Análise Dinâmica; Alias analysis; Call graph analysis;	SmartDroid; MAST; Apposcopy; DroidAnalytics; DroidAPIMiner; AXMLPrinter; uiautomator; Apkanalyzer; grep; Datasets (Offensive Computing, Android Malware Dataset, Malware Genome Project, AndroMalShare, Kharon, DroidCat, Drebin, AppsApk, AndroZoo); Soot (Jimple IR); FlowDroid/IccTA; Smali;	Device Administrator Use and Abuse in Android: Detection and Characterization (SHAN; SAMUEL; NEAMTIU, 2019)
Análise estática; Análise dinâmica; Análise híbrida; API hooking; Dynamic instrumentation; Taint Analysis; Inline hooking; Points-to Analysis; String Analysis; Reachability analysis;	DroidRA; Ripple; TamiFlex; AppsPlayground; DroidScope; TaintDroid; DroidAlarm; monkey tool; Datasets (Drebin); ArtDroid; logcat; PScout; AndroGuard; CopperDroid; UnDroid; SandDroid; MobiSec Lab; AndroTotal; VirusTotal; FlowDroid; IccTa; Amandroid; SCandroid; SAAF; StaDART; DroidBot;	StaDART: Addressing the problem of dynamic code updates in the security analysis of android applications (AHMAD et al., 2019)
Análise estática; Análise dinâmica; Análise híbrida; Event-based triggering; Dynamic instrumentation; Hooking;	MEGDroid; DroidMon; Xposed framework; Genymotion emulator; Jadx; Andrubis sandbox; Dynalog sandbox; DroidBox; DroidBot; Curiousdroid; AppsPlayground; uiautomator; GroddDroid; Adamant; EHBDroid; Dynodroid; adb; Monkey; Datasets (AMD Argus);	MEGDroid: A model-driven event generation framework for dynamic android malware analysis (HASAN; LADANI; ZAMANI, 2021)
Análise estática; Similarity based analysis;	Datasets (Malware Genome Project; AndroZoo); Androguard; Soot (Jimple, Dexpler); PScout; IccTa; FSquaDRA; AppSpear; DexHunter; DNADroid; AnDarwin; DroidMOSS; ResDroid; SMIT; BitShred; PiggyApp;	Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting (LI et al., 2017)

Continua na próxima página

Tabela 2 – continuação da tabela

Técnica(s)	**Ferramenta(s)**	**Título (Referência)**
Análise estática; Sensitivity analysis; Risk analysis; Multimodal Approach;	Datasets (AndroZoo, Drebin); Sensdroid; AndroDialysis; PIndroid; ICCDetector; MaMaDroid; VAnDroid; DroidDet; DroidSieve; VxStream Sandbox; Alde; RNPDroid;	SensDroid: Analysis for Malicious Activity Risk of Android Application (SHRIVASTAVA; KUMAR, 2019)
Análise estática; signature-based; NB; LR; SVM; RF; SMOTE; KNN; J48; Voted Perceptron; CNN; LSTM; C4.5;	Datasets (Drebin, AMD argus); SIGPID; DroidFusion; RevealDroid; MADAM; DroidAPIMiner; DroidSieve; 7zip; apk-decompiler;	A new machine learning-based method for android malware detection on imbalanced dataset (DEHKORDY; RASOOLZADEGAN, 2021)
Manifest based Analysis; NB; SVM; C4.5; Maximum Entropy; text categorization; bag of words; n-gram;	Datasets (Drebin, AndroTracker M0DROID); apktool; WEKA; OpenNLP; ScanDal; DroidMat; MAMA; DroidAPIMiner; AndroSimilar; PUMA; ANASTASIA; ADROIT; TaintDroid; CrowDroid; Andromaly; DroidScope; DroidScribe; DroidCat; CSCdroid; Monet; Andrubis; Mobile-Sandbox; DroidNative; MAIL; OmniDroid; Dendroid; TextDroid; Apkreader;	Adapting Text Categorization for Manifest based Android Malware Detection (COBAN; OZEL, 2019)
Signature-based; permission-based; bytecode-based; Anomaly-based analysis; Taint Analysis; code emulation; Análise Híbrida; fuzzy hashing;	SDHash; VirusTotal; YARA; Limon Sandbox; SSDEEP; Dataset (AMD arguslab); DroidMOSS: ScanDal; Uranine; Andromaly; Crowdroid; Taintdroid; DroidScope; Andrubis; AndroSimilar; APK Auditor; Genymotion emulator;	LimonDroid: a system coupling three signature-based schemes for profiling Android malware (TCHAKOUNTé et al., 2020)
Análise estática; FC; CNN; DAE; DBN; RNN; LSTM; Bayesian Machine; NB; SVM; DT; VMI based analysis;	Kirin; TaintDroid; FlowDroid; DroidScope; Datasets (Android Malware Genome, Drebin, VirusShare); BayesDroid; Python libraries (h5py, numpy, pandas, keras, TensorFlow, theano, pymc3, matplotlib, seaborn); RevealDroid;	Deep neural architectures for large scale android malware analysis (NAUMAN et al., 2017)

Continua na próxima página

Tabela 2 – continuação da tabela

Técnica(s)	**Ferramenta(s)**	**Título (Referência)**
Análise estática; Similarity based analysis;	PiggyApp; DroidMOSS; DNADroid; AnDarwin; ViewDroid;	App in the Middle: Demystify Application Virtualization in Android and its Security Threats (ZHANG et al., 2019)
Análise estática; Análise dinâmica; Signature-based analysis; Permission based analysis; SVM, CART; Dataflow analysis; Taint analysis, Dynamic taint analysis; Virtualization;	RevealDroid; Adagio; MUDFLOW; Dendroid; Datasets (Android Malware Genome, AndroZoo, VirusShare, Drebin, VirusTotal); Soot (and Dexpler); Android ABI toolchain; Scikit-learn; AV-Class; DroidSIFT; FlowDroid; DroidMat; AppContext; ViewDroid; MassVet; DroidAnalytics; AsDroid; HARVESTER; Apposcopy; DroidScribe;	Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware (GARCIA; HAMMAD; MALEK, 2018)
Análise estática; Control Flow Analysis; opcode-based analysis; DT; Model-checking; Value set analysis;	Datasets (Drebin, Contagiominidump); DroidNative; MAIL; IDA Pro; DroidSift; DroidLegacy; AndroSimilar; SDHash; REIL; VINE; CFGO-IL; WIRE; AndroTracker;	DroidNative : Automating and Optimizing Detection of Android Native Code Malware Variants (ALAM et al., 2016)
Análise estática; Análise Dinâmica; RF; KNN; SVM; MC; Information flow analysis; Dataflow analysis; Random Fuzzing; Concolic testing;	Datasets (PlayDrone, Drebin, VirusShare, VirusTotal, McAfee, Genome); DroidAPIMiner; MaMaDroid; Soot; FlowDroid; SuSi; Androguard; TriFlow; AppContext; Dare; Kirin; RiskRanker; CHEX; DroidScope; TaintDroid; ParanoidAndroid; IntelliDroid; ASTROID; NetworkProfiler; StormDroid; MADAM; ScanMe Mobile; AuntieDroid; Kuafudet; Transcend; MalDozer; HinDroid; DroidMat; DroidMiner; MAST; Crowdroid; RevealDroid;	MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models (Extended Version) (ONWUZURIKE et al., 2019)

Continua na próxima página

Tabela 2 – continuação da tabela

Técnica(s)	**Ferramenta(s)**	**Título (Referência)**
Análise estática; análise dinâmica; LSSVM; LSTM; CNN; DAE; SCNN; SVM; DNN; KNN; DT; J48; NB; MLP; RF; Bayesian Classification; BayesNet; Adaboost;	AASandbox; Pegasus; Mobile Application Security Triage (MAST); Kirin; Saint; Woodpecker; Datasets (Drebin, VirusTotal, Microsoft Windows Defender, Android Malware Genome, AndroMalShare); PUMA; RiskRanker; TaintDroid; MADAM; DroidScope; Crowdroid; DeepDroid; PerbDroid; PermPair; DL-Droid; SmartDroid; AppGuard; Andromaly; Aurasium; TstructDroid; AppsPlayground; AppProfiler; Andrubis; Androguard; CopperDroid; HinDroid; DroidCat; MalDozer; DroidDet; GAdroid; DroidMOSS; AndroSimilar; MATLAB;	FSDroid:- A feature selection technique to detect malware from Android using Machine Learning Techniques (MAHINDRU; SANGAL, 2021)

Fonte: o próprio autor.

4.2 Técnicas

Categorizaram-se as técnicas encontradas nos estudos primários nas seguintes classes: Análise Estática, Análise Dinâmica, Análise baseada em políticas de segurança, Análise Híbrida, Outros, *Machine/Deep/Imbalanced Learning* e Processamento de Linguagem Natural. Em seguida, resumiram-se os princípios que fundamentam cada técnica. Por fim, destacaram-se algumas observações e heurísticas, identificadas nos estudos primários, que podem ajudar pesquisadores no futuro.

Tabela 3 – Categorização de técnicas.

Título	**Técnica(s)**
Análise Estática	Alias Analysis; Anomaly-Based Analysis; Bytecode-based Analysis; Call Graph Analysis; Code dependency Analysis; Control Flow Analysis; Dataflow Analysis; Decompiling; Entry Point Analysis; Fuzzy Hashing; Information Flow Analysis; Manifest based Analysis; Opcode-based Analysis; Permission based Analysis; Points-to Analysis; Reachability Analysis; Selective Symbolic Execution; Signature-based Analysis; Similarity based Analysis; String Analysis; String-taint Analysis; Symbolic execution; Taint Analysis; Value Set Analysis; Model Checking; Rule-based Analysis;
Análise Dinâmica	(API, Inline) Hooking; Anomaly-Based Analysis; Behaviour based Analysis; Binary/Dynamic/Bytecode Instrumentation; Code Emulation; Concolic Testing; Debbuging; Event-based Triggering; Random Fuzzing; Sandboxing; String-taint Analysis; Symbolic Data Flow Analysis; Symbolic Execution; System call Interposition; System call tracing; Taint Analysis; Taint Tracking; Time-based Triggering; VMI; Virtualization;
Análise baseada em políticas de segurança	Attribute Based Access Control; Context Based Access Control; Discretionary Access Control; Mandatory Access Control; Role Based Access Control;
Análise Híbrida	Combinação de técnicas de AE e AD
Outros	Formal Concept Analysis; Fuzzy Inference; Risk Analysis;

Continua na próxima página

Tabela 3 – continuação da tabela

Título	**Técnica(s)**
Machine/Deep/Imbalanced Learning	(Linear) SVM; ADASYN; Active learning; Adaboost; Affinity Propagation; ANN; Bagging; Bayes Network (ou BayesNet); Bayesian Classification; Bayesian Model; Boosting; C4.5; Cased Based Reasoning; CART; CNN; Decision Fusion; DT; DecisionTable; DAE; DNN; DBN; Expectation-Maximisation clustering; EML; Extremely Randomized Trees; FC; GKMC; XGBoost; HoeffdingTree; IBK; KNN; LRLR; LSSVM; LR; LogisticModelTrees; LSTM; MC; MM; MLP; Multi-view adversarial network; Multimodal Approach; MIL; NB; PART; Random Committee; RF; Random Over Sampler; RNN; ROT; SMOTE; SVM RBF; Sensitivity Analysis; SMO; SCNN; SL; Stacking; Voted perceptron; KMC; k-star; Association Rule Analysis;
Processamento de Linguagem Natural	Bag of Words; Maximum Entropy; N-gram; Text Categorization; word embedding;

Fonte: o próprio autor.

A lista a seguir resume as técnicas da classe **Análise Estática**:

- ***Alias Analysis***: técnica que determina quais variáveis ou ponteiros podem se referir ao mesmo local na memória (MUCHNICK, 1997);
- ***Anomaly-Based Analysis***: técnica que detecta comportamentos anormais ou desviantes de um aplicativo em relação a um modelo normal, ou de referência (KUROSE; ROSS, 2013);
- ***Bytecode-based Analysis***: técnica que analisa o código intermediário gerado a partir do código-fonte de um aplicativo, como o formato DEX do Android (PAN et al., 2020);
- ***Call Graph Analysis***: técnica que constrói e examina um grafo que representa as chamadas entre funções ou métodos em um aplicativo (SHAN; SAMUEL; NEAMTIU, 2019);
- ***Code dependency Analysis***: técnica que identifica as dependências entre partes do código de um programa (RODRIGUEZ; MOSTAFA; WANG, 2017);

- ***Control Flow Analysis***: técnica que determina o fluxo de execução de um aplicativo, como as condições, loops e saltos que podem ocorrer (NIELSON; NIELSON; HANKIN, 2010);
- ***Dataflow Analysis***: técnica que rastreia o fluxo de dados de um aplicativo, como as origens, destinos e transformações dos dados (NIELSON; NIELSON; HANKIN, 2010);
- ***Decompiling***: técnica que converte o código intermediário ou binário de um aplicativo em um código-fonte de alto nível, como Java ou C (SIKORSKI; HONIG, 2012);
- ***Entry Point Analysis***: técnica que identifica os pontos de entrada de um aplicativo, como as atividades, serviços, receptores e provedores declarados no AndroidManifest.xml (ARZT, 2017);
- ***Fuzzy Hashing***: técnica que calcula uma assinatura única para um arquivo ou parte dele, que pode ser comparada com outras assinaturas para medir a similaridade entre eles (BREITINGER et al., 2014);
- ***Information Flow Analysis***: técnica que verifica se o fluxo de informações de um aplicativo respeita certas propriedades de segurança, como a confidencialidade e a integridade dos dados (BERGERETTI; CARRé, 1985);
- ***Manifest based Analysis***: técnica que analisa o arquivo AndroidManifest.xml de um aplicativo, que contém informações sobre os componentes, permissões, recursos e intenções do aplicativo (PAN et al., 2020);
- ***Opcode-based Analysis***: técnica que analisa as instruções de baixo nível que compõem o código intermediário ou binário de um aplicativo, como os opcodes DEX do Android (PAN et al., 2020);
- ***Permission based Analysis***: técnica que analisa as permissões solicitadas ou concedidas a um aplicativo, que determinam os recursos ou funcionalidades que o aplicativo pode acessar no sistema Android (ARZT, 2017);
- ***Points-to Analysis***: técnica que determina os possíveis valores dos ponteiros em um aplicativo, ou seja, os objetos aos quais eles podem apontar em tempo de execução (CHAKARAVARTHY, 2003);
- ***Reachability Analysis***: técnica que determina quais partes do código de um aplicativo podem ser alcançadas ou executadas a partir de um determinado ponto de entrada ou condição inicial (AHO et al., 2007);

- ***Signature-based Analysis***: técnica que compara o código ou o comportamento de um aplicativo com uma base de dados de assinaturas conhecidas de malwares ou padrões maliciosos (KUROSE; ROSS, 2013);
- ***Similarity based Analysis***: técnica que mede o grau de similaridade entre dois ou mais programas (AUCH et al., 2020);
- ***String Analysis***: técnica para tentar estimar possíveis valores de strings presentes no código de um aplicativo (RODRIGUEZ; MOSTAFA; WANG, 2017);
- ***String-taint Analysis***: técnica que marca as strings sensíveis ou maliciosas em um aplicativo e rastreia seu fluxo através do código ou da memória (WASSERMANN; SU, 2007);
- ***Symbolic execution***: técnica que executa simbolicamente o código de um aplicativo, usando valores simbólicos em vez de concretos para as entradas e variáveis, e gerando restrições lógicas para os possíveis caminhos de execução (NIELSON; NIELSON; HANKIN, 2010);
- ***Taint Analysis***: técnica que marca os dados sensíveis ou maliciosos em um aplicativo e rastreia seu fluxo através do código ou da memória, detectando vazamentos ou usos indevidos dos dados (ARZT, 2017);
- ***Value Set Analysis***: técnica que determina os possíveis valores das variáveis em um aplicativo, usando uma aproximação conservativa que garante a correção da análise (LIN et al., 2019);
- ***Model Checking***: técnica automática para verificar se um modelo de um sistema satisfaz uma especificação formal, expressa como uma fórmula de lógica temporal. A verificação de modelos permite encontrar erros em sistemas complexos (CASOLARE et al., 2021);
- ***Rule-based Analysis***: técnica que aplica um conjunto de regras pré-definidas para detectar padrões ou características específicas no código ou no comportamento de um aplicativo (SHAN; NEAMTIU; SAMUEL, 2018).

Na classe **Análise Dinâmica**, encontram-se técnicas que análogas às técnicas estáticas. No entanto, elas consideram o programa enquanto ele está sendo executado. A seguir, apresentam-se os conceitos que definem cada uma dessas técnicas:

- ***(API, Inline) Hooking***: técnica que intercepta e modifica as chamadas de funções ou métodos em tempo de execução, para monitorar ou alterar o comportamento de um aplicativo (SIKORSKI; HONIG, 2012);

- ***Anomaly-Based Analysis***: técnica que detecta comportamentos anormais ou desviantes de um aplicativo em relação a um modelo normal ou de referência;
- ***Behaviour based Analysis***: técnica que analisa o comportamento de um aplicativo em tempo de execução, como as ações realizadas, os recursos acessados ou as interações com o usuário ou o sistema;
- ***Binary/Dynamic/Bytecode Instrumentation***: técnica que insere código adicional no código binário ou intermediário de um aplicativo em tempo de execução, para monitorar ou alterar seu comportamento (WANG et al., 2017);
- ***Code Emulation***: técnica que simula a execução do código de um aplicativo em um ambiente controlado, para analisar seu comportamento sem arriscar a segurança do sistema real (SIKORSKI; HONIG, 2012);
- ***Concolic Testing (Symbolic Execution)***: técnica que executa simbolicamente o código de um aplicativo, usando valores simbólicos em vez de concretos para as entradas e variáveis, e gerando restrições lógicas para os possíveis caminhos de execução (ANAND et al., 2012);
- ***Debbuging***: técnica que pausa e retoma a execução do código de um aplicativo em pontos específicos, para inspecionar o estado do aplicativo e rastrear a origem dos bugs (SIKORSKI; HONIG, 2012);
- ***Event-based Triggering***: técnica que ativa a execução do código de um aplicativo em resposta a eventos específicos, como cliques do usuário, mensagens recebidas ou alterações no sistema (AYSAN; SAKIZ; SEN, 2019);
- ***Random Fuzzing***: técnica que alimenta o aplicativo com entradas aleatórias ou malformadas, para testar sua robustez (ONWUZURIKE et al., 2019);
- ***Sandboxing***: técnica que isola o aplicativo em um ambiente restrito e controlado, para limitar seu acesso aos recursos do sistema e prevenir danos potenciais (SIKORSKI; HONIG, 2012);
- ***String-taint Analysis***: técnica que marca as strings sensíveis ou maliciosas em um aplicativo e rastreia seu fluxo através do código ou da memória durante a execução;
- ***Symbolic Data Flow Analysis***: técnica que rastreia o fluxo simbólico dos dados de um aplicativo durante a execução, para gerar expressões simbólicas de estruturas condicionais (WANG et al., 2017);
- ***System call Interposition***: técnica que intercepta e modifica as chamadas de sistema feitas por um aplicativo durante a execução, para monitorar ou alterar seu acesso aos recursos do sistema (BIANCHI et al., 2015);

- ***System call tracing***: técnica que registra as chamadas de sistema feitas por um aplicativo durante a execução, para analisar seu comportamento e interações com o sistema (BHANDARI et al., 2017);
- ***Taint Analysis (Taint Tracking)***: técnica que marca os dados sensíveis ou maliciosos em um aplicativo e rastreia seu fluxo através do código ou da memória durante a execução;
- ***Time-based Triggering***: técnica que ativa a execução do código de um aplicativo em momentos específicos ou após intervalos de tempo específicos (AYSAN; SAKIZ; SEN, 2019);
- ***Virtual Machine Introspection (VMI)***: técnica que monitora o estado e o comportamento de uma máquina virtual a partir do hipervisor ou do host, por meio da interceptação de chamadas de sistema (BHANDARI et al., 2018);
- ***Virtualization***: técnica que executa o aplicativo em uma máquina virtual ou emulador, para isolar o aplicativo do sistema real e permitir a análise detalhada de seu comportamento (SIKORSKI; HONIG, 2012).

Também, resumiram-se os itens da classe **Análise baseada em políticas de segurança**, de acordo com Lake (2022):

- ***Attribute Based Access Control (ABAC)***: política de controle de acesso que concede ou nega acesso a recursos com base em atributos associados ao sujeito (usuário ou processo);
- ***Context Based Access Control***: política de controle de acesso que considera o contexto do pedido de acesso, como a localização do usuário, o horário do dia, a carga de trabalho do sistema e assim por diante;
- ***Discretionary Access Control (DAC)***: política de controle de acesso que permite aos proprietários ou criadores de recursos controlar quem pode acessar seus recursos;
- ***Mandatory Access Control (MAC)***: política de controle de acesso que impõe restrições com base em classificações (níveis de segurança) dos sujeitos e objetos;
- ***Role Based Access Control (RBAC)***: política de controle de acesso que concede ou nega acesso a recursos com base nas funções dos usuários. RBAC simplifica o gerenciamento de permissões em grandes organizações, mas requer uma boa definição das funções.

Técnicas matemáticas e computacionais mais específicas encontram-se na classe **Outros**. São elas:

- ***Formal Concept Analysis (FCA)***: técnica matemática que possibilita descobrir estruturas hierárquicas e dependências entre dados (WANG; WANG, 2015);
- ***Fuzzy Inference***: técnica que aplica a lógica fuzzy para tomar decisões com base em regras difusas e variáveis de entrada difusas. Essa técnica é útil quando os dados são imprecisos ou incertos (COPPIN, 2004);
- ***Risk Analysis***: técnica que identifica e avalia os riscos potenciais que podem afetar um projeto, um processo ou uma decisão. A análise de risco permite tomar decisões informadas e preparar planos de contingência (SHRIVASTAVA; KUMAR, 2019).

Técnicas de Aprendizado de Máquina, Aprendizado de Máquina Profundo, Aprendizado Desbalanceado e Processamento de Linguagem Natural, também foram identificadas nos estudos primários. A seguir uma breve elucidação de cada conceito:

- ***Support Vector Machine (SVM)***: modelo de aprendizado supervisionado que usa algoritmos de classificação para construir hiperplanos em um espaço multidimensional (RUSSELL; NORVIG, 2021);
- ***ADASYN***: técnica de aprendizado para conjuntos de dados desbalanceados, baseado em uma distribuição de pesos de classes minoritárias distintas de acordo com seu nível de dificuldade de aprendizado (HE et al., 2008);
- ***Active learning***: abordagem de aprendizado de máquina onde o modelo interage com os dados para rotulá-los (SETTLES, 2009);
- ***Adaboost***: algoritmo de boosting que constrói um modelo forte a partir de vários modelos fracos por um processo iterativo (FREUND; SCHAPIRE, 1995);
- ***Affinity Propagation***: algoritmo de agrupamento baseado em passagem de mensagens entre pontos de dados (FREY; DUECK, 2007);
- ***Artificial Neural Network (ANN)***: modelo computacional inspirado pela estrutura dos neurônios do cérebro humano (RUSSELL; NORVIG, 2021);
- ***Bagging***: técnica de *ensemble* que visa melhorar a estabilidade e precisão do algoritmo de aprendizado de máquina, reduzindo a variância (RUSSELL; NORVIG, 2021);
- ***Bayes Network***: representação gráfica para dependências probabilísticas entre um conjunto de variáveis aleatórias (RUSSELL; NORVIG, 2021);
- ***Bayesian Classification***: tipo de classificador probabilístico baseado no teorema de Bayes (RUSSELL; NORVIG, 2021);

- ***Bayesian Model***: modelo estatístico que encapsula as declarações probabilísticas sobre quantidades desconhecidas (RUSSELL; NORVIG, 2021);
- ***Boosting***: técnica de *ensemble* que combina vários modelos sendo treinado com um conjunto de treinamento com pesos (RUSSELL; NORVIG, 2021);
- ***C4.5***: algoritmo para geração de árvore de decisão, que usa entropia da informação (RUSSELL; NORVIG, 2021);
- ***Case Based Reasoning***: processo de resolução de problemas que utiliza experiências ou casos passados (LEAKE, 2001);
- ***Classification and Regression Trees (CART)***: técnica de modelagem preditiva que constrói árvores de decisão para regressão e classificação, com base no índice de impureza de Gini (RUSSELL; NORVIG, 2021);
- ***Convolutional Neural Network (CNN)***: classe de redes neurais contêm conexões espacialmente locais, pelo menos nas primeiras camadas, com padrões de pesos replicados entre as unidades em cada camada (RUSSELL; NORVIG, 2021);
- ***Decision Fusion***: técnica que combina várias decisões, tomadas por classificadores, para formar uma única decisão final (CHANDOLA et al., 2021);
- ***Decision Tree***: modelo de previsão usando uma estrutura em forma de árvore que mapeia observações sobre um item para conclusões sobre o valor alvo do item (RUSSELL; NORVIG, 2021);
- ***DecisionTable***: tabela visual que mostra uma série de decisões e suas possíveis consequências (WITTEN et al., 2017);
- ***Deep Autoencoder***: rede neural artificial que usa o algoritmo de *backpropagation* para aprendizado de *features* (BHUVANESHWARI et al., 2021);
- ***Deep Neural Network (DNN)***: rede neural artificial com várias camadas entre a camada de entrada e a camada de saída, chamadas camadas ocultas (AOUICHAOUI et al., 2021);
- ***Deep Belief Network (DBN)***: classe de redes neurais profundas compostas por várias camadas de *Restricted Boltzman Machines* (LIU, 2021b);
- ***Expectation-Maximisation clustering***: algoritmo iterativo para a estimativa de máxima verossimilhança de uma distribuição paramétrica subjacente a alguns dados incompletos ou ausentes (GARRIGA et al., 2016);

- ***Extreme Machine Learning***: técnica de aprendizado baseada na topologia de *Multi-layer Perceptron*, mas se difere deste ao inicializar pesos e *biases* randomicamente (LIU, 2021a);
- ***Extremely Randomized Trees***: algoritmo de aprendizado de máquina que usa muitas árvores aleatórias para prever o resultado (GEURTS; ERNST; WEHENKEL, 2006);
- ***Fully Connected Neural Network***: rede neural onde cada neurônio na camada i está conectado a cada neurônio na camada $i + 1$ (RUSSELL; NORVIG, 2021);
- ***Global K-means clustering***: variação do algoritmo *K-means* que inicializa os centróides de maneira mais eficaz para melhorar a qualidade do agrupamento (LIKAS; VLASSIS; VERBEEK, 2003);
- ***Extreme Gradient Boosting (XGBoost)***: algoritmo que implementa o método de *gradient boosting* por meio de *Decision Trees* (SUBASI et al., 2022);
- ***HoeffdingTree***: algoritmo que emprega árvore de decisão para aprender a partir de um fluxo de dados (VERMA, 2021);
- ***IBK***: algoritmo de aprendizado de máquina baseado em instância que usa o algoritmo kNN (KOULIARIDIS et al., 2020);
- ***K-Nearest Neighbor (kNN)***: Um algoritmo de aprendizado supervisionado que armazena todos os casos disponíveis e classifica novos casos com base em uma medida de similaridade (RUSSELL; NORVIG, 2021);
- ***Label Regularized Logistic Regression***: técnica de regressão logística que usa regularização para evitar o *overfitting* (NGUYEN-VU; AHN; JUNG, 2019);
- ***Least Squares SVM***: Uma variação do SVM que usa mínimos quadrados para a função de perda;
- ***Logistic Regression***: modelo estatístico usado para prever a probabilidade de uma classe binária;
- ***LogisticModelTrees***: algoritmo de aprendizado de máquina que combina árvores de decisão e regressão logística (CASOLARE et al., 2021);
- ***Long Short-Term Memory (LSTM)***: rede neural recorrente capaz de aprender dependências de longo prazo (RUSSELL; NORVIG, 2021);
- ***Markov Chain***: modelo estatístico que segue a propriedade de Markov (a probabilidade de cada evento depende apenas do estado do evento anterior) (RUSSELL; NORVIG, 2021);

- ***Markov Model***: modelo estatístico que usa *Markov Chains* para representar sistemas aleatórios (RUSSELL; NORVIG, 2021);
- ***Multi-layer Perceptron (MLP)***: rede neural artificial do tipo *feedforward*, que possui múltiplas camadas de perceptrons (LIU, 2021a);
- ***Multi-view adversarial network***: rede neural que utiliza múltiplas visões dos dados e um componente adversarial para melhorar o desempenho do modelo (CHEN et al., 2018);
- ***Multimodal Approach***: abordagem que combina dados de naturezas distintas, como imagem, áudio e texto para treinar um modelo de aprendizado de máquina (QAISAR; LI, 2021);
- ***Multiple Instance Learning***: tipo de aprendizado supervisionado onde cada exemplo é associado a vários rótulos (CARBONNEAU et al., 2018);
- ***Naive Bayes***: classificador probabilístico baseado na aplicação do teorema de Bayes com suposições de independência fortes entre os recursos (RUSSELL; NORVIG, 2021);
- ***Partial Decision Tree***: árvore de decisão baseado em divisão e conquista (DHALARIA; GANDOTRA, 2020);
- ***Random Committee***: método do tipo *ensemble* que constrói um comitê de modelos aleatórios (KOULIARIDIS et al., 2020);
- ***Random Forest***: método *ensemble* que constrói várias árvores de decisão e agrega suas previsões (RUSSELL; NORVIG, 2021);
- ***Recurrent Neural Network (RNN)***: rede neural na qual as conexões entre os nós formam uma sequência temporal de passos a partir das entradas (RUSSELL; NORVIG, 2021);
- ***Rotation Forest***: algoritmo de *ensemble* que usa extração de *features* para melhorar a precisão do modelo (RODRIGUEZ; KUNCHEVA; ALONSO, 2006);
- ***SMOTE***: técnica de *oversampling* que cria instâncias sintéticas da classe minoritária em vez de criar cópias (DEHKORDY; RASOOLZADEGAN, 2021);
- ***SVM RBF***: variação do SVM que usa uma função de base radial como função do kernel;
- ***Sensitivity Analysis***: técnica usada para determinar como diferentes valores de uma variável independente impactam uma variável dependente particular (SHRIVASTAVA; KUMAR, 2019);

- ***Sequential Minimal Optimization (SMO)***: algoritmo para resolver o problema de otimização quadrática que surge durante o treinamento de SVMs (KOULIARIDIS et al., 2020);
- ***Serial Convolutional Neural Network***: rede neural convolucional que processa entradas em série, um após o outro (MAHINDRU; SANGAL, 2021);
- ***Simple Logistic***: classificador de regressão logística com suporte para seleção de atributos (DING et al., 2020);
- ***Stacking***: técnica de *ensemble* que combina vários modelos de aprendizado de classes distintas (RUSSELL; NORVIG, 2021);
- ***Voted perceptron***: variação do algoritmo do perceptron que considera a votação dos perceptrons durante o treinamento (RUSSELL; NORVIG, 2021);
- ***K-means clustering***: Um algoritmo de agrupamento que divide um conjunto de n observações em k clusters, cada um associado à média das observações no cluster (LIKAS; VLASSIS; VERBEEK, 2003);
- ***k-star***: algoritmo que encontra os k menores caminhos entre um par de vértices em um dado grafo direcionado e ponderado (ALJAZZAR; LEUE, 2011);
- ***Association Rule Analysis***: técnica que encontra relações entre variáveis ou atributos em um conjunto de dados, geralmente na forma de regras do tipo "se A, então B" (LAFOREST, 2021);
- ***Bag of Words***: neste modelo, um texto (como uma frase ou um documento) é representado como um *multiset* de suas palavras, desconsiderando a gramática e até a ordem das palavras, mas mantendo a multiplicidade (RUSSELL; NORVIG, 2021);
- ***Maximum Entropy***: princípio usado em modelagem estatística para selecionar uma distribuição de probabilidade que melhor representa os dados observados, sujeito a um conjunto de restrições conhecidas (RUSSELL; NORVIG, 2021);
- ***N-gram***: sequência de símbolos de tamanho n em que cada palavra depende da anterior (RUSSELL; NORVIG, 2021);
- ***Text Categorization***: É o processo de classificação de documentos de texto em diferentes categorias com base no conteúdo dos documentos (RUSSELL; NORVIG, 2021);
- ***Word Embedding***: técnica em processamento de linguagem natural onde palavras ou frases do vocabulário são mapeadas para vetores de números reais (RUSSELL; NORVIG, 2021).

De acordo com Zhang et al. (2019), técnicas de *Similarity Analysis* são comumente empregadas para detectar aplicativos que sofreram *repackaging*. Ainda, algumas heurísticas foram identificadas para detectar aplicativos maliciosos. Primeiramente, a discrepância entre os meta-dados do arquivo de manifesto e as classes em disco pode indicar o uso de *Packing*. Além disso, o uso dos carregadores de classes Android: *DexClassLoader*, *PathClassLoader* e *InMemoryClassLoader*, bem como o uso de *reflection calls*, como: *Class.forName()* e *Class.getDeclaredField()*, podem indicar atividades suspeitas. Em segundo lugar, se um aplicativo descriptografa, carrega e executa *bytecode*, isso é um forte indício de que tal aplicativo é malicioso (MACHIRY et al., 2018). Reforça-se que essas conclusões são heurísticas e podem não ser verdadeiras em todos os casos, mas fornecem um bom ponto de partida para analisar *malwares* que empregam técnicas mais sofisticadas de evasão.

Por fim, é importante salientar que, devido à natureza intrinsecamente indecidível do problema de determinar o comportamento de um programa sem a sua execução, pesquisadores têm procurado utilizar técnicas de Inteligência Artificial para classificar amostras. Entretanto, os modelos podem ser suscetíveis a viés e podem falhar na classificação de amostras de malware que empregam técnicas inovadoras ou mais sofisticadas. Portanto, torna-se evidente a necessidade de encontrar um equilíbrio entre as abordagens de Análise Estática e Inteligência Artificial, visando mitigar vieses e reduzir tanto os falsos-positivos quanto os falsos-negativos.

4.3 Ferramentas

A tabela 4 lista, conforme a classificação mais apropriada, quais ferramentas foram usadas ou criadas pelos autores dos estudos primários.

Tabela 4 – Categorização de ferramentas.

Título	**Ferramentas(s)**
Análise Estática	7zip; A3E; ACTEve; ADROIT; AIS; ANASTASIA; APK Auditor; APK-DNA; APKCombiner; APKParser; ASTROID; AXMLPrinter2; Adagio; Amandroid; AnDarwing; AndRadar; Andro-AutoPsy; AndroDialysis; AndroSimilar; Android ABI toolchain; Android SDK (dexdump, aapt); AndroidLeaks; Andronio; Apkanalyzer; Apkreader; AppContext; AppIntent; AppSpear; Apposcopy; AsDroid; AutoDroid; BayesDroid; BitShred; CHEX; Capstone; ComDroid; Concept Explorer Tool (Protégé); DIALDroid; DNADroid; DaDiDroid; Dare; Dendroid; DexHunter; DidFail; Draco; DroidAPIMiner; DroidAlarm; DroidAnalytics; DroidAntiRM; DroidDelver; DroidDelver; DroidDet; DroidDetective; DroidFusion; DroidLegacy; DroidLysis; DroidMOSS; DroidMat; DroidMiner; DroidNative; DroidRA; DroidRanger; DroidSIFT; DroidSafe; EMMA; EdgeMiner; Epicc; FAMD; FSquaDRA; FlowDroid; IC3; ICCDetector; IDA Pro; IccTA; JAD; JADX; JEB decompiler; Jitana; Juxtapp; Kirin; Kuafudet; LoopMC; MAMA; MUDFLOW; MaMaDroid; MalPat; Marvin; Mascat; MassVet; Mobile Application Security Triage (MAST); PIndroid; PScout; PUMA; Pegasus; PermPair; PiOS; PiggyApp; RAPID; ROAR; ResDroid; RevealDroid; RevealDroid*; Ripple; RobotDroid; SAAF; SAFEDroid; SCANDAL; SCanDroid; SIFTA; SIGPID; SMIT; SPARK; Sensdroid; Soot; StormDroid; Stowaway; SuSi; Transcend; TriFlow; TriggerScope; Uranine; VAnDroid; VILO; ViewDroid; WALA; Woodpecker; Yara; Z3 constraint solver; apk-decompiler; apktool; axplorer; dcfldd; ded; dedexer; dex2jar; grep; hooper; jd-cmd; jd-gui; k-map; mrshv2; mvHash-B; radare; sdhash; ssdeep; strings; xxd;

Continua na próxima página

Tabela 4 – continuação da tabela

Título	**Ferramentas(s)**
Análise Dinâmica	ANANAS; ANDRUBIS; ANDect; Adamant; AirBag; Anbox; Android Debug Bridge (ADB); Android SDK (Android Emulator); Android Virtual Device (AVD); Apex; AppGuard; AppsPlayground; ArtDroid; AuntieDroid; Aurasium; Autograph; BareDroid; Boxify; Brahmastra; CANDYMAN; CSCDroid; Cells; Condroid; CopperDroid; CrowDroid; Deep4MalDroid; DeepDroid; Docker; Dr. Android; Droid-Sec; DroidBot; DroidBox; DroidCat; DroidDolphin; DroidMon; DroidScope; DroidScribe; DroidSieve; DroidTrace; DynoDroid; EHBDroid; EarlyBird; Endroid; Ether; FireDroid; FireEye IDS; Frida; GAdroid; GUIRipper; Genymotion; Grab'n Run; Hamsa; HinDroid; Honeyspot; I-ARM-Droid; IntelliDroid; IntentDroid; IpVoid; KidLogger; Kprobes; LXC; LXD; Linux Memory Extractor (LiME); MADAM; MALHEUR; MBOX; MEGDroid; MIET; MOSES; Mal-warehouse; Malton; MockDroid; Monkey Tool (do ADB); NJAS; NMAP; NetworkProfiler; Ninja; PROTEUS; Panorama; Paranoid Android; PerbDroid; PolyGraph; PowerTutor; ProcMon; RetroSkeleton; Rippler; Robotium; SCREDENT; SCSDroid; SWORD; Secure Application Interaction (SAINT); SigMal; Smartpot; StaDART; StaDyna; SwiftHand; TISSA; TaintBochs; TaintDroid; TamiFlex; TargetDroid; TextDroid; Tracedroid; TstructDroid; VetDroid; VizMal; Volatility; WebWitness; Webprophet; XManDroid; Xposed Framework; logcat; netem; netstat; ps; ptrace; rkt; strace; tcpdump; tcpstat;
Análise baseada em políticas de segurança	FlaskDroid;

Continua na próxima página

Tabela 4 – continuação da tabela

Título	**Ferramentas(s)**
Análise Híbrida	AASandbox; AV-Class; Alde; AndroTotal; Androguard; Andromaly; ApkScan; AppProfiler; BRIDEMAID; Clam-AV; Cuckoo Sandbox; CuckooDroid; Curiousdroid; DL-Droid; Droid Analyst Combo; Droid-Sec; DroidDetector; Dynalog; Euphony; GroddDroid; HARVESTER; Jackdaw; Limon Sandbox; MONET; MobiSec Lab; Mobile Sandbox; RNPDroid; SEALANT; SMV-Hunter; SandDroid; ScanMe Mobile; SmartDroid; ThreatAnalyzer; UnDroid; VirusTotal; VxStream; uiautomator;
Risk Analysis	RiskInDroid; RiskRanker;
Dataset	Additional Malware Families 2015; AndroMalshare; AndroTracker; AndroZoo; Android Malware Dataset (AMD do Arguslab); Android Malware Genome Project (Malgenome/Genome); Android Proguard; AndroidBotnet2015; AppsApk; CICAAGM; CICAndMal2017; CICInvesAndMal2019; CICMalDroid2020; Contagio minidump; Dataset 1 e 2 (DHALARIA; GANDOTRA, 2020); DexShare; Dr. Web; Drebin; HelDroid; IccRE; Intel Security (McAfee Labs); Kharon; Maldozer; Microsoft Windows Defender; M0DROID; Offensive Computing; OmniDroid; PlayDrone; VirusShare
Machine/Deep Learning	Alligator; Apache Spark; H20 Flow; Keras; LIBSVM; Matlab Neural Network toolbox; OpenNLP; Python Libraries (h5py, numpy, pandas, pymc3, matplotlib, seaborn); Scikit-learn (sklearn); TensorFlow; Theano; WEKA; J48; JRip;
Representações Intermediárias	CFGO-IL; Jimple; MAIL; REIL; Smali/Baksmali; VINE; WIRE;

Fonte: o próprio autor.

4.4 Tipos de *Malwares*

Além das técnicas e ferramentas, também, buscava-se enumerar quais tipos de *malwares* são estudados pelos pesquisadores da área. A lista a seguir agrupa os tipos de *malwares* encontrados nos estudos primários, assim como uma breve descrição de cada um:

- ***Adware***: aplicativo que apresenta propagandas para usuários, na tentiva de gerar renda para seu dono;
- ***Banking Trojan* (Trojan Bancário)**: app que rouba credenciais bancárias do usuário;
- ***Botnet***: sistema distribuído composto por aparelhos infectados por aplicativos maliciosos que enviam e recebem comandos de um servidor de comando e controle;
- ***Hybrid* (Híbrido)**: app que implementa diversos tipos de códigos maliciosos;
- **Minerador de criptomodedas**: código malicioso que se esconde em apps legítimos, em busca de usar a capacidade de processamento do aparelho para *minerar* criptomoedas para seu dono;
- ***Ransomware***: aplicativo que criptografa os dados do usuário com uma chave de criptografia e só compartilha a chave de descriptografia com o usuário mediante pagamento de resgate (*cryptoransomwares*); aplicativo que interrompe o funcionameto do Android por meio de telas persistentes que não podem ser fechadas, ou pelo uso de senha nova (*lockers*); aplicativo que forja falsas evidências de crimes cibernéticos como pornografia infantil, para coagir ou difamar o usuário, publicamente (*scarewares*);
- ***Repackaged* App**: cópia de um aplicativo legítimo desempacotado e reempacotado com alguma alteração em seu código, como mudança de dono do APK;
- ***Piggybacked* App**: aplicativo legítimo que sofreu *repackaging*, mas teve código malicioso inserido, como uma biblioteca de propagandas para gerar renda para o usuário malicioso que o clonou;
- ***Spyware***: app que monitora e coleta informações pessoais do usuário, como e-mail, páginas visitadas com frequência, informações de contatos do usuário, número de cartão de crédito, entre outros, sem o consentimento do usuário;
- ***Worm***: app que se replica entre dispositivos, por meio de Bluetooth, Wi-Fi, SMS, MMS, entre outros;
- **Vírus**: app que se replica entre arquivos de um mesmo sistema, para permitir a execução código malicioso no lugar de código legítimo.
- ***Zero-day***: *malwares* que ainda não foram totalmente identificados, após seu lançamento na rede;
- ***Backdoor***: código malicioso que permite o acesso não autorizado a um dispositivo Android, sem que o dono saiba;

- ***Self-hiding* App**: aplicativo malicioso que implementa funções para: ocultar ícone do App, esconder atividade, apagar mensagens, apagar log de chamadas, bloquear mensagens, bloquear chamadas, ocultar alertas, ocultar notificações, colocar telefone no aspecto mudo, excluir App da lista de Apps usados recentemente e apagar logs do sistema;

Dessa forma, conclui-se que um detector de *malwares* deve conseguir identificar aplicativos maliciosos que apresentem os comportamentos maliciosos descobertos.

4.5 Técnicas Anti-Análise de *Malware*

Como resposta às técnicas usadas por profissionais analisarem *malwares* de Android, usuários maliciosos empregam técnicas anti-análise de malware que inviabilizam as análises estática e dinâmica de uma amostra. As técnicas mais recorrentes nos estudos primários são:

- **Ofuscação (*obfuscation*)**: transformações no código-fonte de um aplicativo que inviabilizam a análise estática, como exemplo, a ofuscação do grafo de fluxo de controle (*Control flow graph*) de um app por meio da inserção de *junk code* e *call inderection*;
- ***Reflection* e *dynamic code loading***: capacidade que permite a um app tratar seu próprio código como dado e manipulá-lo em tempo de execução, seja para atualizar seus recursos, seja para estender suas funcionalidades;
- ***Logic Bomb***: o app malicioso se ativa depois de um período de uso, após sua instalação e tentativa de utilização;
- **Detecção de emuladores/virtualizadores**: código embutido, em app malicioso, que consegue identificar se ele está sendo executado em um emulador/virtualizador de Android;
- ***Collusion attack***: ataque realizado por um malware composto por mais de um app que exige permissões mínimas para operar, mas quando eles se comunicam é realizado atividade maliciosa;
- ***Packing***: ferramenta que envolve o código-fonte, recursos e outros dados de um app em um arquivo, comprimido ou criptografado, que é decodificado (*unpacked*) em tempo de execução;
- ***Antidecompiling***: o código da aplicação é manipulado por meio da inserção de código lixo (*trash code*) em seu fluxo de execução ou pela modificação inteligente

de seus estruturas de dados, de maneira que um *decompiler* (ferramenta que faz descompilação) não consiga gerar código correto;

- ***Antidebugging***: código embutido na aplicação que muda seu fluxo de execução caso alguma modificação em sua integridade seja identificada, uso do mecanismo conhecido como *antitamper*;
- **Manipulação direcionada de *features***: o desenvolvedor do *malware* modifica sua aplicação para que ela não gere *features* usadas por um dado sistema de detecção;

Vale ressaltar que algumas das técnicas listadas não são usadas apenas para fins maliciosos, mas também por empresas que desejam proteger seu *software* contra pirataria ou modificação de seu funcionamento (KIM; KWAK; RYOU, 2015).

4.6 Sistema Implementado

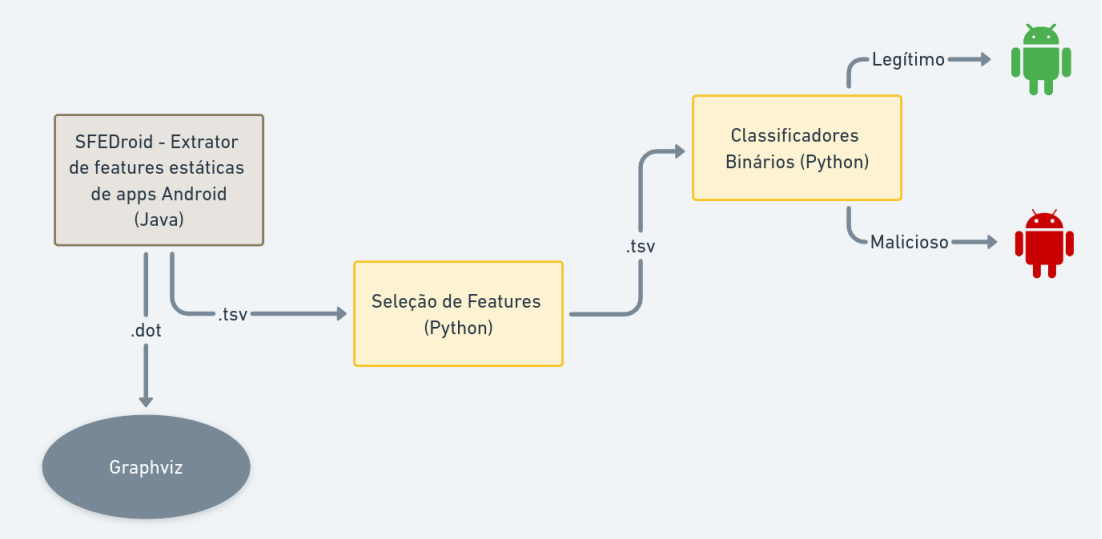
A fim de compreender como as técnicas e ferramentas dos estudos primários podem ser empregadas na prática, implementou-se um sistema que monta um *dataset* com *features* estáticas de aplicativos Android, para treinamento de diferentes classificadores binários. Na sequência, são apresentados uma visão geral do sistema, os tipos de *malwares* contidos no *dataset*, detalhamento sobre as *features* extraídas, os resultados estatísticos do treinamento dos classificadores e as limitações do sistema implementado.

4.6.1 Visão Geral

O sistema implementado consiste em duas grandes partes: a primeira implementada em Java e a segunda implementada em Python. O programa implementado em Java é chamado *Static Features Extractor for Android* (SFEDroid). Sua finalidade é coletar *features* estáticas de APKs, sem executá-los. Já a implementação em Python consiste em um programa que seleciona *features* relevantes, do dataset, e treina os seguintes classificadores binários: Decision Tree, Random Forest, Adaboost, Naive Bayes e SVM (RBF).

O projeto é de código aberto e está disponível em https://is.gd/SHVWsY. A figura 15 mostra a organização geral do sistema implementado.

Figura 15 – Visão geral do sistema implementado.



Fonte: o próprio autor.

O sistema SFEDroid realiza análises estáticas de APKs por meio do framework *FlowDroid*, para extrair *features estáticas*. Esta ferramenta é incluída no projeto como uma biblioteca, e auxilia nas implementações dos algoritmos de *Call Graph Analysis*, *Permission based Analysis* e *Taint Analysis*. Além disso, empregou-se a biblioteca *Graphviz* para impressão do *call graph* extraído do app analisado em formato DOT, como exemplificado na seção 3.2.2.2. Assim que as *features* são extraídas do APK, gera-se o *dataset* em formato TSV, para que o sistema em Python execute.

4.6.2 *Features* extraídas

Cada *feature* escolhida tenta descrever uma parte do aplicativo em análise, visando caracterizá-lo como um aplicativo legítimo ou malicioso. A lista a seguir descreve os motivos pelos quais cada *feature* foi escolhida:

- **Versão mínima do SDK do Android**: a versão mínima exigida pelo app em seu arquivo de manifesto (*AndroidManifest.xml*) dita qual versão mínima da API do Android será usada. Isso facilita na identificação de quais métodos de API podem ser usados pelo app;
- **Versão alvo do SDK do Android**: de forma análoga à versão mínimia do SDK, a versão alvo também permite o reconhecimento direcionado de quais métodos da API do Android um app deve estar utilizando;
- **Tamanho em bytes do APK**: APKs legítimos tendem a ter um tamanho maior que APKs maliciosos (KARBAB et al., 2018). Entende-se que aplicativos legítimos têm mais arquivos de mídia;

- **Entropia do arquivo *DEX* do aplicativo**: código executável (*bytecode* ou, no caso do Android, *classes.dex*) com entropia alta (maior que 7) é um forte indicativo de que a amostra está comprimida, encriptada ou *packed* (KIM; KWAK; RYOU, 2015);

- **Permissões exigidas pelo aplicativo**: as permissões exigidas por um app permitem que ele utilize um conjunto de métodos da API do Android para operar (BACKES et al., 2016), portanto, facilitam identificar métodos alcançáveis;

- **Métodos alcançáveis no *call graph***: utilizou-se o mapeamento permissões-métodos publicados por (BACKES et al., 2016) para escolha de uma *feature* binária para cada método que era alcançável ou não no *call graph* do app em análise. A intenção é captar quais métodos podem ser ativados pelos usuários enganados por aplicativos maliciosos;

- **Métodos *sinks* que vazam dados**: para cada método identificado como *sink* em um *leak* (vazamento) de dados na *Taint Analysis*, assume-se que tal método está vazando informação do usuário, ou não. Assim adaptou-se a capacidade da *FlowDroid* em identificar *leaks* para caracterizar um app que vaza dados sensíveis do usuário;

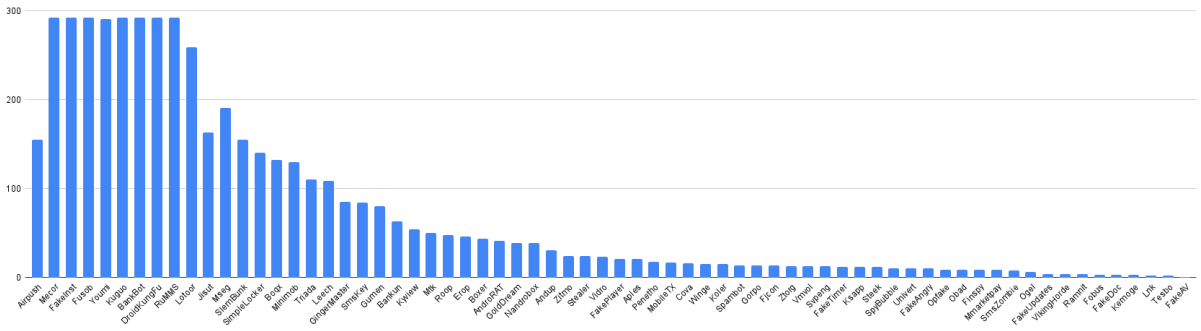
Cada aplicativo é de uma classe: 0 para apps legítimos e 1 para apps maliciosos. Ao final da montagem do arquivo *dataset*, agregaram-se 5.174 *features* de 10.000 amostras. Na seção 3.1.1, detalha-se como as amostras foram coletadas e de quais fontes.

4.6.3 Tipos de *malwares* do *dataset*

Ao todo, após a seleção dos arquivos não corrompidos com a linha de comando da seção 3.1.1, selecionaram-se 20.983 amostras do conjunto AMD. Porém, apenas 15.451 amostras foram analisadas estaticamente pelo sistema implementado. Isso ocorreu devido a algum dos seguintes motivos: o arquivo de manifesto falhou em ser analisado; as classes do aplicativo não foram carregas corretamente; o cabeçalho CEN era inválido; o aplicativo não terminou de ser analisado em um período pré-determinado, logo, não gerou nenhum resultado.

Visto que o conjunto de amostras legítimas continha 5.000 amostras analisadas com sucesso, reduziu-se o número de amostras maliciosas para 5.000. Implementou-se o *script* em Python, veja apêndice A.1, para selecionar 5.000 amostras distribuídas em 68 famílias. O gráfico da figura 16 mostra o número de amostras por família. Buscou-se manter o maior número de amostras por família possível.

Figura 16 – Total de amostras por família.



Fonte: o próprio autor.

Conforme a tabela de informações sobre o *dataset* AMD, recuperado de cache do site oficial do projeto, em https://is.gd/hKobYZ, mapeou-se o nome da família, o número de amostras usadas e o tipo de *malware* na tabela 5.

Tabela 5 – Tipos de *malwares* do *dataset*.

Família	**Nº de amostras**	**Tipo de *Malware***
Airpush	155	Adware
Mecor	292	Trojan-Spy
FakeInst	292	Trojan-SMS
Fusob	292	Ransom
Youmi	291	Adware
Kuguo	292	Adware
BankBot	292	Trojan-Banker
DroidKungFu	292	Backdoor
RuMMS	292	Trojan-SMS
Lotoor	259	HackerTool
Jisut	163	Ransom
Mseg	191	Trojan
SlemBunk	155	Trojan-Banker
SimpleLocker	140	Ransom
Boqx	132	Trojan-Dropper
Minimob	130	Adware
Triada	110	Backdoor
Leech	109	Trojan-SMS
GingerMaster	85	Backdoor
SmsKey	84	Trojan-SMS

Continua na próxima página

Tabela 5 – continuação da tabela

Família	**Nº de amostras**	**Tipo de *Malware***
Gumen	80	Trojan-SMS
Bankun	63	Trojan-Banker
Kyview	54	Adware
Mtk	50	Trojan
Roop	48	Ransom
Erop	46	Trojan-SMS
Boxer	44	Trojan-SMS
AndroRAT	41	Backdoor
GoldDream	39	Backdoor
Nandrobox	39	Trojan
Andup	31	Adware
Zitmo	24	Trojan-Banker
Stealer	24	Trojan-SMS
Vidro	23	Trojan-SMS
FakePlayer	21	Trojan-SMS
Aples	21	Ransom
Penetho	18	HackerTool
MobileTX	17	Trojan
Cova	16	Trojan-SMS
Winge	15	Trojan-Clicker
Koler	15	Ransom
Spambot	14	Backdoor
Gorpo	14	Trojan-Dropper
Fjcon	14	Backdoor
Ztorg	13	Trojan-Dropper
Vmvol	13	Trojan-Spy
Svpeng	13	Trojan-Banker
FakeTimer	12	Trojan
Ksapp	12	Trojan
Steek	12	Trojan-Clicker
SpyBubble	10	Trojan-SMS
Univert	10	Backdoor
FakeAngry	10	Backdoor
Opfake	9	Trojan-SMS
Obad	9	Backdoor

Continua na próxima página

Tabela 5 – continuação da tabela

Família	**Nº de amostras**	**Tipo de *Malware***
Finspy	9	Trojan-Spy
Mmarketpay	9	Trojan
SmsZombie	8	Trojan-Spy
Ogel	6	Trojan-SMS
FakeUpdates	4	Trojan
VikingHorde	4	Trojan-Dropper
Ramnit	4	Trojan-Dropper
Fobus	3	Backdoor
FakeDoc	3	Trojan
Kemoge	3	Trojan-Dropper
Lnk	2	Trojan
Tesbo	2	Trojan-SMS
FakeAV	1	Trojan

Fonte: o próprio autor.

Em suma, montou-se um conjunto de dados com 10.000 amostras, em que 5.000 são amostras legítimas, extraídas do *dataset AndroZoo*, e as outras 5.000 são amostras maliciosas, tiradas do *dataset* AMD.

4.6.4 Resultados do treinamento

Após montar o *dataset* em arquivo TSV com a ferramenta *SFEDroid*, forneceu-se tal arquivo como entrada ao programa no *Kaggle*, para treinar os classificadores binários escolhidos. Para isso, instanciou-se dois conjuntos de dados em memória, após selecionar as melhores *features* com os procedimentos detalhados na seção 3.2.4. Dessa forma, ambos conjuntos de dados possuem 10.000 amostras, porém, o primeiro possui 515 *features* e o segundo, 100 *features*.

Para treinar os classificadores, separaram-se, estratificadamente, 30% do *dataset* para teste e 70% para treinamento. Parametrizou-se cada classificador da seguinte forma: DT com profundidade máxima 5; RF com profundidade máxima 5 e 100 estimadores; AdaBoost e NB com as configurações padrões do *sklearn*; SVM com o *kernel* RBF, dado a natureza não linear do conjunto de dados (como mostra o gráfico de dispersão da figura 17).

Tabela 6 – Número de VP, VN, FP e FN de cada classificador treinado com 515 *features*.

Classificador	VN	VP	FP	FN
DT	1498	1490	10	2
RF	1500	1459	41	0
AdaBoost	1500	1496	4	0
NB	1420	705	795	80
SVM (RBF)	1440	689	811	60

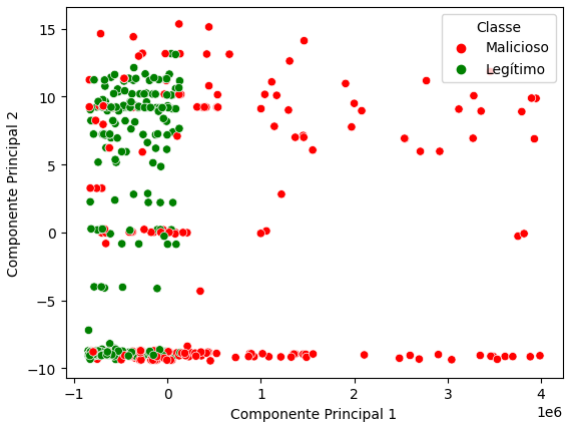
Fonte: o próprio autor.

Tabela 7 – Número de VP, VN, FP e FN de cada classificador treinado com 100 *features*.

Classificador	VN	VP	FP	FN
DT	1498	1490	10	2
RF	1500	1476	24	0
AdaBoost	1500	1496	4	0
NB	1420	705	795	80
SVM (RBF)	1440	689	811	60

Fonte: o próprio autor.

Figura 17 – Gráfico de dispersão do conjunto de dados.



Fonte: o próprio autor.

As tabelas 6 e 7 mostram o número de VP, VN, FP e FN de cada classificador treinado, usados para calcular os parâmetros de avaliação. Ademais, as tabelas 8 e 9 mostram as métricas estatísticas que caracterizam a qualidade de cada classificador treinado. Observa-se que não se realizou nenhum treinamento focado na busca pelos melhores parâmetros de cada classificador. Logo, sustenta-se que eles podem ser melhorados para realizar a tarefa proposta e que o desempenho dos modelos não mudam, significativamente, ao reduzir o número de 515 *features* para 100.

Ao final de cada treinamento, imprimiu-se a matriz de confusão e a curva ROC de

Tabela 8 – Resultados dos parâmetros de avaliação com 515 *features*.

Classificador	**Acurácia**	**F-measure**	**MCC**	**AUC**	**Média Acurácias (10-fold)**	**Desvio Padrão (10-fold)**
DT	0.996	0.996	0.992	1.00	0.998	0.001
RF	0.987	0.986	0.974	1.00	0.987	0.003
AdaBoost	0.999	0.999	0.997	1.00	0.999	0.001
NB	0.708	0.617	0.474	0.75	0.705	0.015
SVM (RBF)	0.710	0.613	0.484	0.78	0.709	0.013

Fonte: o próprio autor.

Tabela 9 – Resultados dos parâmetros de avaliação com 100 *features*.

Classificador	**Acurácia**	**F-measure**	**MCC**	**AUC**	**Média Acurácias (10-fold)**	**Desvio Padrão (10-fold)**
DT	0.997	0.997	0.994	1.00	0.998	0.001
RF	0.992	0.992	0.985	1.00	0.993	0.003
AdaBoost	0.999	0.999	0.997	1.00	0.998	0.001
NB	0.708	0.617	0.474	0.75	0.705	0.015
SVM (RBF)	0.710	0.613	0.484	0.78	0.709	0.013

Fonte: o próprio autor.

cada classificador. A figura 18, mostram os resultados em matrizes de confusão e curvas ROC após o treinamento dos classificadores com 100 *features*.

O projeto está disponível na plataforma *Kaggle*, para reprodução e conferência mais detalhada dos resultados https://is.gd/jKwngy.

4.6.5 Limitações

O sistema implementado apresenta algumas limitações. Escolheram-se as versões entre a 4.1 e 7.1 do Android para trabalhar, pois o mapeamento permissão-métodos mais recente encontrado na literatura contempla apenas tais versões. Ademais, como a *SFEDroid* é uma extensão da ferramenta *FlowDroid*, herda-se todas as limitações desta. Ainda, no que concerne ao *dataset* criado, ele contém apenas 10.000 amostras. Por último, os classificadores implementados não foram otimizados, particularmente, para realizar a tarefa proposta.

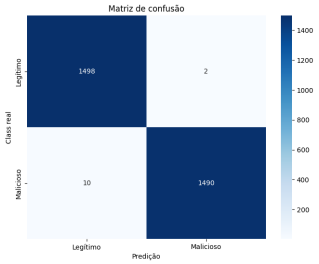
Primeiramente, a versão do sistema Android escolhida influencia diretamente nas análises estáticas empregadas, pois elas dependem de um conjunto de permissões e vários métodos. Ambos recebem modificações a cada versão lançada do sistema. Assim, escapa da análise aplicativos implementados com métodos e permissões desconhecidos ao mapeamento utilizado.

Em segunda instância, por ser uma extensão da ferramenta *FlowDroid*, *SFEDroid* herda limitações daquela. Ambas são suscetíveis a não detectarem *malwares* que utilizem alguma ou várias das seguintes técnicas anti-análise: *Dynamic code loading*, *Logic bomb*, *Packing* e *Antidecompiling*.

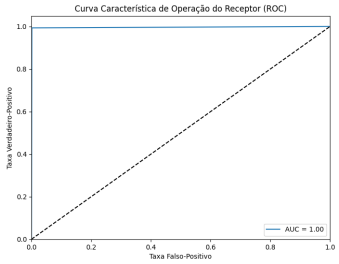
Em suma, o *dataset* montado não é tão compreensivo e está datado. O baixo número de amostras impede a obtenção de resultados de um treinamento mais otimizado. E, a versão dos aplicativos escolhidos impedem afirmar que o sistema encontra vazamentos de informações sensíveis do usuário em versões mais recentes do Android, como a versão 13, por exemplo.

Figura 18 – Resultados das classificações com 100 *features*.

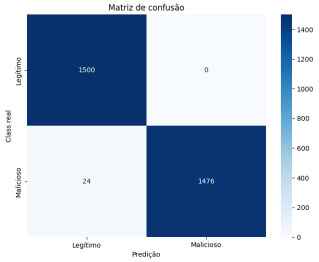
(a) Matriz de confusão: DT.



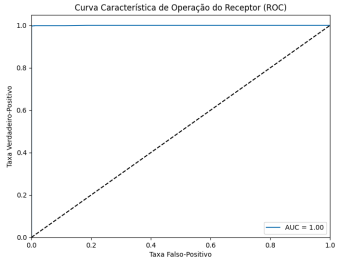
(b) Curva ROC: DT.



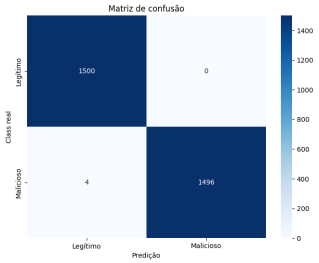
(c) Matriz de confusão: RF.



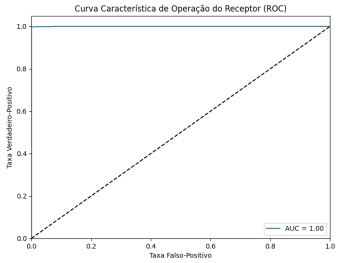
(d) Curva ROC: RF.



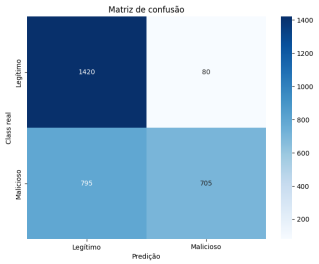
(e) Matriz de confusão: AdaBoost.



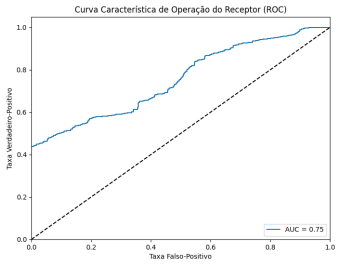
(f) Curva ROC: AdaBoost.



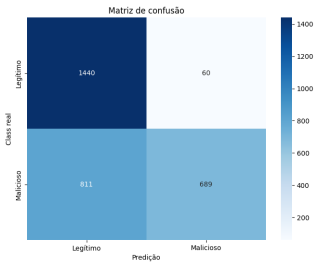
(g) Matriz de confusão: NB.



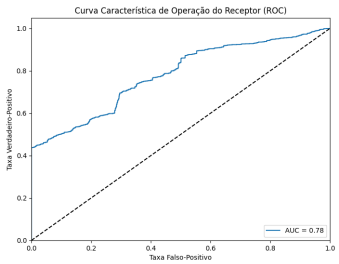
(h) Curva ROC: NB.



(i) Matriz de confusão: SVM (RBF).



(j) Curva ROC: SVM (RBF).



Fonte: o próprio autor.

5 Conclusão

A pesquisa realizada neste trabalho oferece uma visão abrangente sobre a crescente dependência da sociedade contemporânea em relação à tecnologia, em particular aos dispositivos móveis e os perigos decorrentes de aplicativos maliciosos. Diante desse cenário, a ameaça dos *malwares* em dispositivos Android torna-se cada vez mais preocupante, uma vez que eles podem ser usados para uma ampla gama de atividades ilícitas que afetam a segurança e a privacidade dos usuários.

Por meio da SLR conduzida, de estudos entre os anos de 2015 a 2021, identificaram-se as técnicas e ferramentas que estão sendo utilizadas para analisar aplicativos maliciosos de Android. Deste modo, leram-se 60 artigos, por completo, para identificar e classificar técnicas e ferramentas, tipos de *malwares* e técnicas anti-análise de *malware*. Identificaram-se 118 técnicas que tratam problemas dessa área de estudo. Elas foram categorizadas nas seguintes classes: Análise Estática, Análise Dinâmica, Análise baseada em políticas de segurança, Análise Híbrida, Outros, *Machine/Deep/Imbalanced Learning* e Processamento de Linguagem Natural. Também, classificaram-se 357 ferramentas nas mesmas categorias das técnicas; listaram-se 9 técnicas anti-análise de *malware*; e enumeraram-se 14 tipos de *malwares*.

Além disso, o trabalho vai além da revisão bibliográfica ao implementar um sistema prático que utiliza técnicas de análise estática de aplicativos Android para criar um *dataset* com *features* estáticas, treinar classificadores binários e avaliar seu desempenho. Por meio da montagem de um *dataset* com 10.000 aplicativos, legítmos e maliciosos, empregaram-se as técnicas *Permission based Analysis*, *Call Graph Analysis* e *Taint Analysis*. Dessa forma, concluiu-se que é possível caracterizar aplicativos Android, enquanto são identificados métodos alcançáveis no *call graph* de um app e métodos que vazam informações sensíveis do usuário. Com o *dataset* pronto, implementou-se e avaliou-se os resultados dos seguintes classificadores binários: *Decision Tree*, *Random Forest*, *Adaboost*, *Naive Bayes* e SVM (RBF). Sendo assim, após avaliação, concluiu-se que o classificador *AdaBoost* tem o melhor desempenho, mediante os parâmetros de avaliação e as 100 melhores *features* usados.

No entanto, este trabalho também identificou limitações e desafios, como a constante evolução das ameaças cibernéticas e a necessidade contínua de atualização das técnicas e ferramentas de análise. Portanto, esse estudo não apenas contribui para a compreensão da análise de *malwares* em dispositivos Android, mas também destaca a necessidade de pesquisa contínua e inovação para enfrentar os desafios emergentes no campo da segurança cibernética. Logo, é fundamental que pesquisadores e profissionais da área trabalhem em conjunto para garantir um ambiente digital seguro e confiável para todos.

6 Trabalhos futuros

Como sugestões para pesquisas futuras, elenca-se a elaboração de uma técnica de análise estática mais sofisticada para capturar características de outros ambientes de execução, como o da linguagem *Dart*. Por último, propõe-se o emprego de algumas propriedades de grafos para melhorar os resultados de análises de fluxo de dados.

Tanto ferramentas do mercado, como da academia, assumem como ponto de partida, nas análises, *bytecodes* da *Dalvik Virtual Machine*, máquina virtual Java do Android. Em seguida, outro módulo traduz o código executável para representações intermediárias como *Smali* e *Jimple*, por exemplo. Porém, *Dart* dispõe de sua própria máquina virtual, logo, não é analisada apropriadamente pelos aparatos mencionados. Portanto, entende-se que para analisar estaticamente esse tipo de aplicativo, é necessário a utilização de uma suíte de compiladores capaz de traduzir *bytecodes* da *DartVM*, para uma representação intermediária de alto nível, suscetível a otimizações, como a LLVM-IR.

Em suma, sugere-se que para obter resultados mais precisos em análises de fluxos de dados, o CFG de um programa deve ser analisado com mais rigor. Para isso, acredita-se que um grafo reduzido, é um forte candidato a eliminar informações irrelevantes para a distinção entre programas maliciosos e programas legítimos.

Referências

AHMAD, M. et al. StaDART: Addressing the problem of dynamic code updates in the security analysis of android applications. *Journal of Systems and Software*, v. 159, p. 1–16, 2019. ISSN 01641212. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0164121219301530>. Citado na página 58.

AHO, A. V. et al. (Ed.). *Compilers: principles, techniques, & tools*. 2nd ed. ed. [S.l.]: Pearson/Addison Wesley, 2007. Citado 4 vezes nas páginas 19, 20, 21 e 64.

ALAM, S. Applying Natural Language Processing for detecting malicious patterns in Android applications. *Forensic Science International: Digital Investigation*, v. 39, p. 301270, dez. 2021. ISSN 26662817. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S2666281721001888>. Citado na página 56.

ALAM, S. et al. DroidNative: Automating and optimizing detection of Android native code malware variants. *Computers & Security*, v. 65, p. 230–246, 2016. ISSN 01674048. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S016740481630164X>. Citado na página 60.

ALJAZZAR, H.; LEUE, S. K∗: A heuristic search algorithm for finding the k shortest paths. *Artificial Intelligence*, v. 175, n. 18, p. 2129–2154, dez. 2011. ISSN 00043702. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0004370211000865>. Citado na página 72.

ALMAHMOUD, M.; ALZU'BI, D.; YASEEN, Q. ReDroidDet: Android Malware Detection Based on Recurrent Neural Network. *Procedia Computer Science*, v. 184, p. 841–846, 2021. ISSN 18770509. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S187705092100747X>. Citado na página 50.

ALZAYLAEE, M. K.; YERIMA, S. Y.; SEZER, S. DL-Droid: Deep learning based android malware detection using real devices. *Computers & Security*, v. 89, p. 1–11, 2019. ISSN 01674048. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0167404819300161>. Citado na página 53.

ANAND, S. et al. Automated concolic testing of smartphone apps. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. Cary North Carolina: ACM, 2012. p. 1–11. ISBN 9781450316149. Disponível em: <https://dl.acm.org/doi/10.1145/2393596.2393666>. Citado na página 66.

ANDROID. *Android Activity Lifecycle*. 2023. Disponível em: <https://developer.android.com/guide/components/activities/activity-lifecycle>. Citado 2 vezes nas páginas 26 e 28.

ANDROID. *App manifest overview*. 2023. Disponível em: <https://developer.android.com/guide/topics/manifest/manifest-intro>. Citado na página 25.

ANDROID. *Application fundamentals*. 2023. Disponível em: <https://developer.android.com/guide/components/fundamentals>. Citado na página 25.

ANDROID. *Permissions on Android*. 2023. Disponível em: <https://developer.android.com/guide/topics/permissions/overview>. Citado na página 29.

AOUICHAOUI, A. R. et al. Comparison of Group-Contribution and Machine Learning-based Property Prediction Models with Uncertainty Quantification. In: *Computer Aided Chemical Engineering*. Elsevier, 2021. v. 50, p. 755–760. ISBN 9780323885065. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/B9780323885065501182>. Citado na página 69.

APPEL, A. W.; PALSBERG, J. *Modern Compiler Implementation in Java*. 2nd. ed. USA: Cambridge University Press, 2003. ISBN 052182060X. Citado 3 vezes nas páginas 20, 21 e 22.

APVRILLE, A.; APVRILLE, L. SherlockDroid: a research assistant to spot unknown malware in Android marketplaces. *Journal of Computer Virology and Hacking Techniques*, v. 11, n. 4, p. 235–245, nov. 2015. ISSN 2263-8733. Disponível em: <http://link.springer.com/10.1007/s11416-015-0245-z>. Citado na página 54.

ARZT, S. *Static Data Flow Analysis for Android Applications*. Tese (Doutorado), 2017. Disponível em: <http://tuprints.ulb.tu-darmstadt.de/5937/>. Citado 7 vezes nas páginas 33, 40, 41, 42, 43, 64 e 65.

AUCH, M. et al. Similarity-based analyses on software applications: A systematic literature review. *Journal of Systems and Software*, v. 168, p. 110669, out. 2020. ISSN 01641212. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0164121220301278>. Citado na página 65.

AYSAN, A. I.; SAKIZ, F.; SEN, S. Analysis of dynamic code updating in Android with security perspective. *IET Information Security*, v. 13, n. 3, p. 269–277, maio 2019. ISSN 1751-8717, 1751-8717. Disponível em: <https://onlinelibrary.wiley.com/doi/10.1049/iet-ifs.2018.5316>. Citado 3 vezes nas páginas 52, 66 e 67.

BACKES, M. et al. On demystifying the android application framework: Re-Visiting android permission specification analysis. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016. p. 1101–1118. ISBN 978-1-931971-32-4. Disponível em: <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/backes_android>. Citado 2 vezes nas páginas 39 e 81.

BARTEL, A. et al. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In: *ACM Sigplan International Workshop on the State Of The Art in Java Program Analysis*. [S.l.: s.n.], 2012. ISBN 978-1-4503-1490-9. Citado 2 vezes nas páginas 33 e 34.

BERGERETTI, J.-F.; CARRé, B. A. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, v. 7, n. 1, p. 37–61, jan. 1985. ISSN 0164-0925, 1558-4593. Disponível em: <https://dl.acm.org/doi/10.1145/2363.2366>. Citado na página 64.

BHANDARI, S. et al. Android inter-app communication threats and detection techniques. *Computers & Security*, v. 70, p. 392–421, set. 2017. ISSN 01674048. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0167404817301414>. Citado 2 vezes nas páginas 50 e 67.

BHANDARI, S. et al. SWORD: Semantic aWare andrOid malwaRe Detector. *Journal of Information Security and Applications*, v. 42, p. 46–56, out. 2018. ISSN 22142126. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S2214212617305616>. Citado 2 vezes nas páginas 54 e 67.

BHUVANESHWARI, M. et al. A comprehensive review on deep learning techniques for a BCI-based communication system. In: *Demystifying Big Data, Machine Learning, and Deep Learning for Healthcare Analytics*. Elsevier, 2021. p. 131–157. ISBN 9780128216330. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/B9780128216330000131>. Citado na página 69.

BIANCHI, A. et al. NJAS: Sandboxing Unmodified Applications in non-rooted Devices Running stock Android. In: *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. Denver Colorado USA: ACM, 2015. p. 27–38. ISBN 978-1-4503-3819-6. Disponível em: <https://dl.acm.org/doi/10.1145/2808117.2808122>. Citado 2 vezes nas páginas 50 e 66.

BOTACIN, M.; GRéGIO, A.; GEUS, P. de. Malware variants identification in practice. In: *Anais do XIX Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*. Porto Alegre, RS, Brasil: SBC, 2019. p. 29–42. ISSN 0000-0000. Disponível em: <https://sol.sbc.org.br/index.php/sbseg/article/view/13960>. Citado na página 18.

BREITINGER, F. et al. *Approximate matching : definition and terminology*. [S.l.], 2014. NIST SP 800–168 p. Disponível em: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-168.pdf>. Citado na página 64.

CALLEJA, A. et al. Picking on the family: Disrupting android malware triage by forcing misclassification. *Expert Systems with Applications*, v. 95, p. 113–126, 2017. ISSN 09574174. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0957417417307881>. Citado na página 57.

CARBONNEAU, M.-A. et al. Multiple instance learning: A survey of problem characteristics and applications. *Pattern Recognition*, v. 77, p. 329–353, maio 2018. ISSN 00313203. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0031320317304065>. Citado na página 71.

CASOLARE, R. et al. Android Collusion Detection by means of Audio Signal Analysis with Machine Learning techniques. *Procedia Computer Science*, v. 192, p. 2340–2346, 2021. ISSN 18770509. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S1877050921017208>. Citado 3 vezes nas páginas 50, 65 e 70.

CAVIGLIONE, L. et al. Seeing the Unseen: Revealing Mobile Malware Hidden Communications via Energy Consumption and Artificial Intelligence. *IEEE Transactions on Information Forensics and Security*, v. 11, n. 4, p. 799–810, 2015. ISSN 1556-6013, 1556-6021. Disponível em: <http://ieeexplore.ieee.org/document/7362010/>. Citado na página 55.

CHAKARAVARTHY, V. T. New results on the computability and complexity of points–to analysis. *ACM SIGPLAN Notices*, v. 38, n. 1, p. 115–125, jan. 2003. ISSN 0362-1340, 1558-1160. Disponível em: <https://dl.acm.org/doi/10.1145/640128.604142>. Citado na página 64.

CHANDOLA, Y. et al. End-to-end pre-trained CNN-based computer-aided classification system design for chest radiographs. In: *Deep Learning for Chest Radiographs*. Elsevier, 2021. p. 117–140. ISBN 9780323901840. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/B9780323901840000114>. Citado na página 69.

CHAU, N.-T.; JUNG, S. Dynamic analysis with Android container: Challenges and opportunities. *Digital Investigation*, v. 27, p. 38–46, dez. 2018. ISSN 17422876. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S1742287618301257>. Citado na página 52.

CHEN, T. et al. TinyDroid: A Lightweight and Efficient Model for Android Malware Detection and Classification. *Mobile Information Systems*, v. 2018, p. 1–9, out. 2018. ISSN 1574-017X, 1875-905X. Disponível em: <https://www.hindawi.com/journals/misy/2018/4157156/>. Citado 2 vezes nas páginas 53 e 71.

COBAN, O.; OZEL, S. Adapting Text Categorization for Manifest based Android Malware Detection. *Computer Science*, v. 20, n. 3, p. 305–327, 2019. ISSN 1508-2806. Disponível em: <http://journals.agh.edu.pl/csci/article/view/3285>. Citado na página 59.

COHEN, F. Computer viruses: Theory and experiments. *Computers & Security*, v. 6, n. 1, p. 22–35, 1987. ISSN 0167-4048. Disponível em: <https://www.sciencedirect.com/science/article/pii/0167404887901222>. Citado na página 19.

COLE, Y. et al. ScanMe mobile: a local and cloud hybrid service for analyzing APKs. In: *Proceedings of the 2015 Conference on research in adaptive and convergent systems*. Prague Czech Republic: ACM, 2015. p. 268–273. ISBN 978-1-4503-3738-0. Disponível em: <https://dl.acm.org/doi/10.1145/2811411.2811483>. Citado na página 50.

COPPIN, B. *Artificial intelligence illuminated*. 1st ed. ed. Boston: Jones and Bartlett Publishers, 2004. ISBN 9780763732301. Citado 2 vezes nas páginas 30 e 68.

DEAN, J.; GROVE, D.; CHAMBERS, C. Optimization of object-oriented programs using static class hierarchy analysis. In: *Proceedings of the 9th European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer-Verlag, 1995. (ECOOP '95), p. 77–101. ISBN 3540601600. Citado na página 40.

DEHKORDY, D. T.; RASOOLZADEGAN, A. A new machine learning-based method for android malware detection on imbalanced dataset. *Multimedia Tools and Applications*, v. 80, n. 16, p. 24533–24554, jul. 2021. ISSN 1380-7501, 1573-7721. Disponível em: <https://link.springer.com/10.1007/s11042-021-10647-z>. Citado 2 vezes nas páginas 59 e 71.

DHALARIA, M.; GANDOTRA, E. A Hybrid Approach for Android Malware Detection and Family Classification. *International Journal of Interactive Multimedia and Artificial Intelligence*, v. 6, n. 6, p. 174, 2020. ISSN 1989-1660. Disponível em: <https://www.ijimai.org/journal/sites/default/files/2021-05/ijimai_6_6_18.pdf>. Citado 4 vezes nas páginas 49, 57, 71 e 76.

DING, Y. et al. Android malware detection method based on bytecode image. *Journal of Ambient Intelligence and Humanized Computing*, jun. 2020. ISSN 1868-5137, 1868-5145. Disponível em: <https://link.springer.com/10.1007/s12652-020-02196-4>. Citado 2 vezes nas páginas 53 e 72.

ELISH, K. O. et al. Profiling user-trigger dependence for Android malware detection. *Computers & Security*, v. 49, p. 255–273, mar. 2015. ISSN 01674048. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0167404814001631>. Citado na página 55.

ESTADÃO. *Brasil tem 230 milhões de smartphones em uso*. 2019. Disponível em: <https://epocanegocios.globo.com/Tecnologia/noticia/2019/04/brasil-tem-230-milhoes-de-smartphones-em-uso.html>. Acesso em: 17 mai 2023. Citado na página 18.

FASANO, F. et al. Energy Consumption Metrics for Mobile Device Dynamic Malware Detection. *Procedia Computer Science*, v. 159, p. 1045–1052, 2019. ISSN 18770509. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S1877050919314693>. Citado na página 51.

FREUND, Y.; SCHAPIRE, R. E. A desicion-theoretic generalization of on-line learning and an application to boosting. In: GOOS, G. et al. (Ed.). *Computational Learning Theory*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995. v. 904, p. 23–37. ISBN 9783540591191 9783540491958. Disponível em: <http://link.springer.com/10.1007/3-540-59119-2_166>. Citado 2 vezes nas páginas 47 e 68.

FREY, B. J.; DUECK, D. Clustering by Passing Messages Between Data Points. *Science*, v. 315, n. 5814, p. 972–976, fev. 2007. ISSN 0036-8075, 1095-9203. Disponível em: <https://www.science.org/doi/10.1126/science.1136800>. Citado na página 68.

GALANTE, L. et al. Malicious linux binaries: a landscape. In: *WTICG, SBSeg'18, XVIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais, SBC*. [S.l.: s.n.], 2018. p. 213–222. Citado na página 18.

GARCIA, J.; HAMMAD, M.; MALEK, S. Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware. *ACM Transactions on Software Engineering and Methodology*, v. 26, n. 3, p. 1–29, 2018. ISSN 1049-331X, 1557-7392. Disponível em: <https://dl.acm.org/doi/10.1145/3162625>. Citado na página 60.

GARRIGA, J. et al. Expectation-Maximization Binary Clustering for Behavioural Annotation. *PLOS ONE*, v. 11, n. 3, p. e0151984, mar. 2016. ISSN 1932-6203. Disponível em: <https://dx.plos.org/10.1371/journal.pone.0151984>. Citado na página 69.

GEURTS, P.; ERNST, D.; WEHENKEL, L. Extremely randomized trees. *Machine Learning*, v. 63, n. 1, p. 3–42, abr. 2006. ISSN 0885-6125, 1573-0565. Disponível em: <http://link.springer.com/10.1007/s10994-006-6226-1>. Citado na página 70.

HARTMAN, K. G. *Calculate File Entropy*. 2013. Disponível em: <https://kennethghartman.com/blog/calculate-file-entropy/>. Citado na página 44.

HASAN, H.; LADANI, B. T.; ZAMANI, B. MEGDroid: A model-driven event generation framework for dynamic android malware analysis. *Information and Software Technology*, v. 135, p. 1–16, jul. 2021. ISSN 09505849. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0950584921000525>. Citado na página 58.

HE, H. et al. ADASYN: Adaptive synthetic sampling approach for imbalanced learning. In: *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*. [S.l.: s.n.], 2008. p. 1322–1328. ISSN: 2161-4407. Citado na página 68.

IRAZOQUI, G.; EISENBARTH, T.; SUNAR, B. MASCAT: Preventing Microarchitectural Attacks Before Distribution. In: *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. Tempe AZ USA: ACM, 2018. p. 377–388. ISBN 978-1-4503-5632-9. Disponível em: <https://dl.acm.org/doi/10.1145/3176258.3176316>. Citado na página 56.

JUNG, H. M.; KIM, K.-B.; CHO, H.-J. A study of android malware detection techniques in virtual environment. *Cluster Computing*, v. 19, n. 4, p. 2295–2304, dez. 2016. ISSN 1386-7857, 1573-7543. Disponível em: <http://link.springer.com/10.1007/s10586-016-0630-5>. Citado na página 51.

KARBAB, E. B.; DEBBABI, M. MalDy: Portable, data-driven malware detection using natural language processing and machine learning techniques on behavioral analysis reports. *Digital Investigation*, v. 28, p. S77–S87, abr. 2019. ISSN 17422876. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S1742287619300271>. Citado na página 54.

KARBAB, E. B. et al. MalDozer: Automatic framework for android malware detection using deep learning. *Digital Investigation*, v. 24, p. S48–S59, mar. 2018. ISSN 17422876. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S1742287618300392>. Citado 2 vezes nas páginas 55 e 80.

KARBAB, E. B.; DEBBABI, M.; MOUHEB, D. Fingerprinting Android packaging: Generating DNAs for malware detection. *Digital Investigation*, v. 18, p. S33–S45, ago. 2016. ISSN 17422876. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S1742287616300469>. Citado na página 51.

KAUR, P.; SHARMA, S. Spyware Detection in Android Using Hybridization of Description Analysis, Permission Mapping and Interface Analysis. *Procedia Computer Science*, v. 46, p. 794–803, 2015. ISSN 18770509. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S1877050915002124>. Citado na página 53.

KIM, D.; KWAK, J.; RYOU, J. DWroidDump: Executable Code Extraction from Android Applications for Malware Analysis. *International Journal of Distributed Sensor Networks*, v. 11, n. 9, p. 379682, set. 2015. ISSN 1550-1477, 1550-1477. Disponível em: <http://journals.sagepub.com/doi/10.1155/2015/379682>. Citado 4 vezes nas páginas 44, 52, 79 e 81.

KITCHENHAM, B.; CHARTERS, S. Guidelines for performing systematic literature reviews in software engineering. *EBSE Technical Report*, Keele e Durham - UK, v. 1, n. EBSE 2007-001, p. 16–29, 2007. Citado 2 vezes nas páginas 23 e 24.

KOULIARIDIS, V. et al. A Survey on Mobile Malware Detection Techniques. *IEICE Transactions on Information and Systems*, E103.D, n. 2, p. 204–211, fev. 2020. ISSN 0916-8532, 1745-1361. Disponível em: <https://www.jstage.jst.go.jp/article/transinf/E103.D/2/E103.D_2019INI0003/_article>. Citado 4 vezes nas páginas 51, 70, 71 e 72.

KUROSE, J. F.; ROSS, K. W. *Computer networking: a top-down approach*. 6th ed. ed. Boston: Pearson, 2013. OCLC: ocn769141382. ISBN 9780132856201. Citado 2 vezes nas páginas 63 e 65.

LAFOREST, A. *Association Analysis Explained*. 2021. Disponível em: <https://towardsdatascience.com/association-analysis-explained-255823c1cf9a>. Citado na página 72.

LAKE, K. *What Are the Different Types of Access Control?* 2022. Disponível em: <https://jumpcloud.com/blog/different-types-access-control>. Citado na página 67.

LEAKE, D. Problem Solving and Reasoning: Case-based. In: *International Encyclopedia of the Social & Behavioral Sciences*. Elsevier, 2001. p. 12117–12120. ISBN 9780080430768. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/B0080430767005453>. Citado na página 69.

LI, L. et al. Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting. *IEEE Transactions on Information Forensics and Security*, v. 12, n. 6, p. 1269–1284, jun. 2017. ISSN 1556-6013, 1556-6021. Disponível em: <http://ieeexplore.ieee.org/document/7828100/>. Citado na página 58.

LIKAS, A.; VLASSIS, N.; VERBEEK, J. J. The global k-means clustering algorithm. *Pattern Recognition*, v. 36, n. 2, p. 451–461, fev. 2003. ISSN 00313203. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0031320302000602>. Citado 2 vezes nas páginas 70 e 72.

LIN, J. et al. A Value Set Analysis Refinement Approach Based on Conditional Merging and Lazy Constraint Solving. *IEEE Access*, v. 7, p. 114593–114606, 2019. ISSN 2169-3536. Disponível em: <https://ieeexplore.ieee.org/document/8805076/>. Citado na página 65.

LIU, H. Data-driven spatial wind forecasting methods along railways. In: *Wind Forecasting in Railway Engineering*. Elsevier, 2021. p. 283–319. ISBN 9780128237069. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/B9780128237069000089>. Citado 2 vezes nas páginas 70 e 71.

LIU, H. Single-point wind forecasting methods based on reinforcement learning. In: *Wind Forecasting in Railway Engineering*. Elsevier, 2021. p. 177–214. ISBN 9780128237069. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/B9780128237069000053>. Citado na página 69.

LORENZO, A. D. et al. Visualizing the outcome of dynamic analysis of Android malware with VizMal. *Journal of Information Security and Applications*, v. 50, p. 1–9, 2019. ISSN 22142126. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S2214212619303837>. Citado na página 52.

MACHIRY, A. et al. Using Loops For Malware Classification Resilient to Feature-unaware Perturbations. In: *Proceedings of the 34th Annual Computer Security Applications Conference*. San Juan PR USA: ACM, 2018. p. 112–123. ISBN 978-1-4503-6569-7. Disponível em: <https://dl.acm.org/doi/10.1145/3274694.3274731>. Citado 2 vezes nas páginas 56 e 73.

MAHINDRU, A.; SANGAL, A. FSDroid:- A feature selection technique to detect malware from Android using Machine Learning Techniques: FSDroid. *Multimedia Tools and Applications*, v. 80, n. 9, p. 13271–13323, abr. 2021. ISSN 1380-7501, 1573-7721. Disponível em: <https://link.springer.com/10.1007/s11042-020-10367-w>. Citado 2 vezes nas páginas 61 e 72.

MARTIN, A.; HERNANDEZ-CASTRO, J.; CAMACHO, D. An in-Depth Study of the Jisut Family of Android Ransomware. *IEEE Access*, v. 6, p. 57205–57218, 2018. ISSN 2169-3536. Disponível em: <https://ieeexplore.ieee.org/document/8481441/>. Citado na página 57.

MARTINELLI, F.; MARULLI, F.; MERCALDO, F. Evaluating Convolutional Neural Network for Effective Mobile Malware Detection. *Procedia Computer Science*, v. 112, p. 2372–2381, 2017. ISSN 18770509. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S1877050917316204>. Citado na página 51.

MARTíN, A.; RODRíGUEZ-FERNáNDEZ, V.; CAMACHO, D. CANDYMAN: Classifying Android malware families by modelling dynamic traces with Markov chains. *Engineering Applications of Artificial Intelligence*, v. 74, p. 121–133, set. 2018. ISSN 09521976. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0952197618301374>. Citado na página 54.

MCNEIL, P. et al. SCREDENT: Scalable Real-time Anomalies Detection and Notification of Targeted Malware in Mobile Devices. *Procedia Computer Science*, v. 83, p. 1219–1225, 2016. ISSN 18770509. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S1877050916302873>. Citado na página 51.

MONTEIRO, M. et al. *Informática Forense*. 1. ed. São Paulo: Leud, 2018. 424 p. Citado 2 vezes nas páginas 22 e 23.

MUCHNICK, S. S. *Advanced compiler design and implementation*. San Francisco, Calif: Morgan Kaufmann Publishers, 1997. ISBN 9781558603202. Citado na página 63.

NAUMAN, M. et al. Deep neural architectures for large scale android malware analysis. *Cluster Computing*, v. 21, n. 1, p. 569–588, 2017. ISSN 1386-7857, 1573-7543. Disponível em: <http://link.springer.com/10.1007/s10586-017-0944-y>. Citado na página 59.

NGUYEN-VU, L.; AHN, J.; JUNG, S. Android Fragmentation in Malware Detection. *Computers & Security*, v. 87, p. 101573, nov. 2019. ISSN 01674048. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0167404819301361>. Citado 2 vezes nas páginas 53 e 70.

NIELSON, F.; NIELSON, H. R.; HANKIN, C. *Principles of program analysis: with 51 tables*. Softcover version of original hardcover edition 1999. Berlin: Springer, 2010. ISBN 9783642084744. Citado 3 vezes nas páginas 19, 64 e 65.

ONWUZURIKE, L. et al. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models (Extended Version). *ACM Transactions on Privacy and Security*, v. 22, n. 2, p. 1–34, maio 2019. ISSN 2471-2566, 2471-2574. Disponível em: <https://dl.acm.org/doi/10.1145/3313391>. Citado 2 vezes nas páginas 60 e 66.

PAN, Y. et al. A Systematic Literature Review of Android Malware Detection Using Static Analysis. *IEEE Access*, v. 8, p. 116363–116379, 2020. ISSN 2169-3536. Disponível em: <https://ieeexplore.ieee.org/document/9118907/>. Citado 2 vezes nas páginas 63 e 64.

PONTTE, B. D.; DOMINICO, S.; ALVES, M. Proposta para o uso de contadores de performance em hardware para detecção de malware. In: *Anais da XXIII Escola Regional de Alto Desempenho da Região Sul*. Porto Alegre, RS, Brasil: SBC, 2023. p. 77–80. ISSN 2595-4164. Disponível em: <https://sol.sbc.org.br/index.php/eradrs/article/view/24506>. Citado na página 18.

QAISAR, Z. H.; LI, R. Multimodal information fusion for android malware detection using lazy learning. *Multimedia Tools and Applications*, v. 81, n. 9, p. 12077–12091, 2021. ISSN 1380-7501, 1573-7721. Disponível em: <https://link.springer.com/10.1007/s11042-021-10749-8>. Citado 3 vezes nas páginas 34, 57 e 71.

RASTOGI, S.; BHUSHAN, K.; GUPTA, B. Android Applications Repackaging Detection Techniques for Smartphone Devices. *Procedia Computer Science*, v. 78, p. 26–32, 2016. ISSN 18770509. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S1877050916000089>. Citado na página 50.

RODRIGUEZ, J.; KUNCHEVA, L.; ALONSO, C. Rotation Forest: A New Classifier Ensemble Method. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 28, n. 10, p. 1619–1630, out. 2006. ISSN 0162-8828, 2160-9292. Disponível em: <http://ieeexplore.ieee.org/document/1677518/>. Citado na página 71.

RODRIGUEZ, R.; MOSTAFA, S.; WANG, X. NTApps: A Network Traffic Analyzer of Android Applications. In: *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*. Indianapolis Indiana USA: ACM, 2017. p. 199–206. ISBN 978-1-4503-4702-0. Disponível em: <https://dl.acm.org/doi/10.1145/3078861.3084175>. Citado 3 vezes nas páginas 51, 63 e 65.

ROY, A. et al. Android Malware Detection based on Vulnerable Feature Aggregation. *Procedia Computer Science*, v. 173, p. 345–353, 2020. ISSN 18770509. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S1877050920315441>. Citado na página 52.

RUSSELL, S. J.; NORVIG, P. *Artificial intelligence: a modern approach*. Fourth edition. Hoboken: Pearson, 2021. (Pearson series in artificial intelligence). ISBN 9780134610993. Citado 9 vezes nas páginas 30, 46, 47, 48, 68, 69, 70, 71 e 72.

SAMPAIO, L. *Novo vírus intercepta transferências via Pix e altera valor e destinatário; veja como funciona*. 2023. Disponível em: <https://www.infomoney.com.br/minhas-financas/novo-virus-intercepta-transferencias-via-pix-e-altera-valor-e-destinatario-veja-como-funciona/>. Citado na página 18.

SENOCAK, G. *Revisão Sistemática da Literatura em Engenharia de Software Teoria e Prática*. [S.l.], 2019. Citado 3 vezes nas páginas 31, 35 e 36.

SETTLES, B. *Active Learning Literature Survey*. [s.n.], 2009. Disponível em: <https://api.semanticscholar.org/CorpusID:324600>. Citado na página 68.

SHAN, Z.; NEAMTIU, I.; SAMUEL, R. Self-hiding behavior in Android apps: detection and characterization. In: *Proceedings of the 40th International Conference on Software Engineering*. Gothenburg Sweden: ACM, 2018. p. 728–739. ISBN 978-1-4503-5638-1. Disponível em: <https://dl.acm.org/doi/10.1145/3180155.3180214>. Citado 2 vezes nas páginas 55 e 65.

SHAN, Z.; SAMUEL, R.; NEAMTIU, I. Device Administrator Use and Abuse in Android: Detection and Characterization. In: *The 25th Annual International Conference on Mobile Computing and Networking*. Los Cabos Mexico: ACM, 2019. p. 1–16. ISBN 978-1-4503-6169-9. Disponível em: <https://dl.acm.org/doi/10.1145/3300061.3345452>. Citado 2 vezes nas páginas 58 e 63.

SHRIVASTAVA, G.; KUMAR, P. SensDroid: Analysis for Malicious Activity Risk of Android Application. *Multimedia Tools and Applications*, v. 78, n. 24, p. 35713–35731, dez. 2019. ISSN 1380-7501, 1573-7721. Disponível em: <http://link.springer.com/10.1007/s11042-019-07899-1>. Citado 3 vezes nas páginas 59, 68 e 71.

SIKORSKI, M.; HONIG, A. *Practical Malware Analysis*. 1. ed. [S.l.]: No Starch Press, 2012. 800 p. Citado 6 vezes nas páginas 22, 23, 64, 65, 66 e 67.

SUBASI, A. et al. Advanced pattern recognition tools for disease diagnosis. In: *5G IoT and Edge Computing for Smart Healthcare*. Elsevier, 2022. p. 195–229. ISBN 9780323905480. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/B9780323905480000115>. Citado na página 70.

TAYLOR, P. *Global mobile OS market share 2022*. 2023. Disponível em: <https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/>. Citado na página 18.

TCHAKOUNTé, F. et al. LimonDroid: a system coupling three signature-based schemes for profiling Android malware. *Iran Journal of Computer Science*, v. 4, n. 2, p. 95–114, 2020. ISSN 2520-8438, 2520-8446. Disponível em: <https://link.springer.com/10.1007/s42044-020-00068-w>. Citado na página 59.

TSUTANO, Y. et al. Jitana: A modern hybrid program analysis framework for android platforms. *Journal of Computer Languages*, v. 52, p. 55–71, 2018. ISSN 25901184. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S1045926X1830209X>. Citado na página 55.

URCUQUI-LóPEZ, C.; CADAVID, A. N. Framework for malware analysis in Android. *Sistemas y Telemática*, v. 14, n. 37, p. 45–56, ago. 2016. ISSN 1692-5238. Disponível em: <https://www.icesi.edu.co/revistas/index.php/sistemas_telematica/article/view/2241>. Citado na página 54.

VERMA, Y. *A Beginner's Guide to Hoeffding Tree with Python Implementation*. 2021. Disponível em: <https://analyticsindiamag.com/a-beginners-guide-to-hoeffding-tree-with-python-implementation/>. Citado na página 70.

WANG, P.; WANG, Y.-S. Malware behavioural detection and vaccine development by using a support vector model classifier. *Journal of Computer and System Sciences*, v. 81, n. 6, p. 1012–1026, set. 2015. ISSN 00220000. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0022000014001780>. Citado 2 vezes nas páginas 57 e 68.

WANG, X. et al. Droid-AntiRM: Taming Control Flow Anti-analysis to Support Automated Dynamic Analysis of Android Malware. In: *Proceedings of the 33rd Annual Computer Security Applications Conference*. Orlando FL USA: ACM, 2017. p. 350–361. ISBN 978-1-4503-5345-8. Disponível em: <https://dl.acm.org/doi/10.1145/3134600.3134601>. Citado 2 vezes nas páginas 56 e 66.

WASSERMANN, G.; SU, Z. Sound and precise analysis of web applications for injection vulnerabilities. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. San Diego California USA: ACM, 2007. p. 32–41. ISBN 9781595936332. Disponível em: <https://dl.acm.org/doi/10.1145/1250734.1250739>. Citado na página 65.

WITTEN, I. H. et al. Output. In: *Data Mining*. Elsevier, 2017. p. 67–89. ISBN 9780128042915. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/B9780128042915000039>. Citado na página 69.

YOUSEFI-AZAR, M. et al. Malytics: A Malware Detection Scheme. *IEEE Access*, v. 6, p. 49418–49431, 2018. ISSN 2169-3536. Disponível em: <https://ieeexplore.ieee.org/document/8463441/>. Citado na página 56.

ZHANG, H.; YAO, D. D.; RAMAKRISHNAN, N. Causality-based Sensemaking of Network Traffic for Android Application Security. In: *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security*. Vienna Austria: ACM, 2016. p. 47–58. ISBN 978-1-4503-4573-6. Disponível em: <https://dl.acm.org/doi/10.1145/2996758.2996760>. Citado na página 56.

ZHANG, L. et al. App in the Middle: Demystify Application Virtualization in Android and its Security Threats. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, v. 3, n. 1, p. 1–24, mar. 2019. ISSN 2476-1249. Disponível em: <https://dl.acm.org/doi/10.1145/3322205.3311088>. Citado 2 vezes nas páginas 60 e 73.

ZHANG, X.; BREITINGER, F.; BAGGILI, I. Rapid Android Parser for Investigating DEX files (RAPID). *Digital Investigation*, v. 17, p. 28–39, jun. 2016. ISSN 17422876. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S1742287616300305>. Citado na página 55.

ZUO, C.; LIN, Z. SMARTGEN: Exposing Server URLs of Mobile Apps With Selective Symbolic Execution. In: *Proceedings of the 26th International Conference on World Wide Web*. Perth Australia: International World Wide Web Conferences Steering Committee, 2017. p. 867–876. ISBN 9781450349130. Disponível em: <https://dl.acm.org/doi/10.1145/3038912.3052609>. Citado na página 53.

A Apêndice - Códigos Auxiliares

Códigos que auxiliaram na manutenção dos resultados do sistema implementado.

A.1 *Script* para filtrar amostras por família

O *script*, descrito em *Python*, abaixo seleciona K amostras de um número fixo de famílias, tentando pegar pelo menos uma amostra de cada família.

```
import os
from pathlib import Path
import re
from functools import reduce
import math
from py7zr import SevenZipFile


# Filtering logs for skipped samples
def filterLogs():
  logsPath = Path('./logs')
  logsPath.resolve()
  dLogsSkipped = dict()
  dLogsAnalyzed = dict()
  for _, _, files in os.walk(logsPath):
    for logFile in sorted(files):
      if logFile.endswith('.log'):
        file = open(logsPath.joinpath(logFile), 'r')
        content = file.read()
        dLogsSkipped[logFile] = re.findall(r'\'(.*)\' file. Skipping
    \.\.\.', content)
        dLogsAnalyzed[logFile] = re.findall(r'\/1000\)\: \'(.*)\'\.\.\.'
    , content)
        file.close()
  skipped = list(reduce(lambda x, acc: acc + x, dLogsSkipped.values()))
  analyzed = list(reduce(lambda x, acc: acc + x, dLogsAnalyzed.values())
    )
  return skipped, analyzed


# Filtering samples that were not skipped
def filterSamplesFullyAnalyzed():
  skipped, analyzed = filterLogs()
  fullyAnalyzedSamples = dict()
  samplesByFamily = dict()
```

```
33    i = 1
34    with open('apksSortedAscendingBySize.txt', 'r') as f:
35      for cnt in f:
36        fileName = cnt.strip()
37        if fileName in analyzed and fileName not in skipped:
38          fullyAnalyzedSamples[i] = fileName
39          i += 1
40          familyName = fileName.split('/')[4]
41          if familyName in samplesByFamily.keys():
42            samplesByFamily[familyName].append(fileName)
43          else:
44            samplesByFamily[familyName] = [fileName]
45
46    return fullyAnalyzedSamples, samplesByFamily
47
48
49  # Mapping dataset samples by index
50  def mapSamplesIndex():
51    samplesIndex = dict()
52    with open('dataset_malware_noPkgNames_noTarget.tsv', 'r') as df:
53      for idx, sample in enumerate(df):
54        samplesIndex[idx] = sample
55    return samplesIndex
56
57
58  def clearEmptyListKeys(originalDict):
59    return {key: originalDict[key] for key, value in originalDict.items()
       if len(value) > 0}
60
61
62  def truncateDict(maxSize, originalDict):
63    return {key: originalDict[key] for key in originalDict.keys() if key <
       maxSize}
64
65
66  def selectKFamiliesfromDict(K, samplesByFamily, maxSamplesPerFamily,
       groups=dict({0: list()}), j=0):
67    filledChunkList = list(filter(lambda x: len(x) == maxSamplesPerFamily,
       groups.values()))
68    if len(filledChunkList) >= (K + 1):
69      return truncateDict((K + 1), groups)
70
71    samplesByFamily = dict(sorted(samplesByFamily.items(), key=lambda x:
       len(x[1]), reverse=True))
72    for samples in samplesByFamily.values():
73      while (len(samples) > 0) and (len(groups[j]) < maxSamplesPerFamily):
74        groups[j].append(samples.pop())
```

```
75
76      if len(groups[j]) == maxSamplesPerFamily:
77        j += 1
78        groups[j] = list()
79
80    samplesByFamily = clearEmptyListKeys(samplesByFamily)
81    return selectKFamiliesfromDict(K, samplesByFamily, maxSamplesPerFamily
      , groups, j)
82
83
84  def getPackedFileSize(file):
85    zipName = '/'.join(file.split('/')[:6])
86    archivedFile = '/'.join(file.split('/')[6:])
87    with SevenZipFile(zipName, 'r') as archive:
88      if archivedFile in archive.getnames():
89        fileSize = list(filter(lambda x: x.filename == archivedFile,
      archive.list()))[0].uncompressed
90    return fileSize
91
92
93  def sortByFileSize(list_):
94    mapSizesSamples = {x: getPackedFileSize(x) for x in list_}
95    sortedSizes = dict(sorted(mapSizesSamples.items(), key=lambda x: x[1])
      )
96    return list(map(lambda x: x, sortedSizes.keys()))
97
98
99  def main():
100   K_samples = 5000
101   mapFullyAnalyzedSamples = dict()
102   samplesIndex = mapSamplesIndex()
103   fullyAnalyzedSamples, samplesByFamily = filterSamplesFullyAnalyzed()
104   with open(f'{K_samples}_samples_dataset.tsv', 'w') as ff:
105     samplesToExport = list()
106     for key in sorted(samplesIndex.keys()):
107       if key == 0:
108         samplesToExport.append(samplesIndex[key])
109         continue
110       mapFullyAnalyzedSamples[fullyAnalyzedSamples[key]] = samplesIndex[
      key]
111
112     maxSamplesPerFamily = math.floor(K_samples / len(samplesByFamily.
      keys()))
113
114     groups = selectKFamiliesfromDict(len(samplesByFamily),
      samplesByFamily, maxSamplesPerFamily)
115
```

```
116     toExportList = list(reduce(lambda x, acc: x + acc, groups.values()))
117
118     # filter out not mapped samples
119     notMappedSamples = [x for x in toExportList if x not in
    mapFullyAnalyzedSamples.keys()]
120     toExportList = list(filter(lambda x: x not in notMappedSamples,
    toExportList))
121
122     # remove the excess
123     for _ in range(len(toExportList)-K_samples):
124       toExportList.pop()
125
126     families = dict()
127     for x in toExportList:
128       fam = x.split('/')[4]
129       if fam in families.keys():
130         families[fam] += 1
131       else:
132         families[fam] = 1
133     fms = ''
134     for fam, tot in families.items():
135       fms += f'{fam}, {tot}\n'
136     famtot = open('famtot.csv', 'w')
137     famtot.write(fms)
138     famtot.close()
139
140     # fixing prefixes
141     toExportList = [k.replace('/home/rafalzdev/AndroidMalwareDataset(
    Arguslab)/', '/media/casperento/hd-extra-2/AMD/') for k in
    toExportList]
142
143     # fixing 7z files names
144     toExportList = [re.sub(r'(/media/casperento/hd-extra-2/AMD/)(.*)(/
    variety\d+/.*\.apk)', r'\1Argus.\2.7z/\2\3', t) for t in toExportList
    ]
145
146     # sort by uncompressed file size, without extracting apks from 7z
    files
147     toExportList = sortByFileSize(toExportList)
148
149     # refixing prefixes
150     toExportList = [k.replace('/media/casperento/hd-extra-2/AMD/', '/
    home/rafalzdev/AndroidMalwareDataset(Arguslab)/') for k in
    toExportList]
151
152     # refixing files names
```

```
153     toExportList = [re.sub(r'(/home/rafalzdev/AndroidMalwareDataset\(
    Arguslab\))Argus\.(.*)\.7z/\2(/variety\d+/.*\.apk)', r'\1\2\3', t)
    for t in toExportList]
154
155     samplesToExport += list(map(lambda x: mapFullyAnalyzedSamples[x],
    toExportList))
156     samplesToExport = ''.join(samplesToExport)
157     ff.write(samplesToExport)
158
159
160 main()
```