

UNIVERSIDADE FEDERAL DO PARANÁ

RODRIGO GOMES LEMOS

DETECÇÃO DE ATAQUES NO ANDROID
UM MODELO BASEADO NA COMUNICAÇÃO ENTRE PROCESSOS

CURITIBA PR

2021

RODRIGO GOMES LEMOS

DETECÇÃO DE ATAQUES NO ANDROID
UM MODELO BASEADO NA COMUNICAÇÃO ENTRE PROCESSOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Carlos Alberto Maziero.

CURITIBA PR

2021

Catálogo na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

L557d

Lemos, Rodrigo Gomes

Detecção de ataques no Android - um modelo baseado na comunicação entre processos [recurso eletrônico] / Rodrigo Gomes Lemos. – Curitiba, 2021.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2021.

Orientador: Carlos Alberto Maziero

1. Android (Recurso eletrônico). 2. Teste de invasão (Medidas de segurança para computadores). 3. Malware (software de computador). Vírus de computador. I. Universidade Federal do Paraná. II. Maziero, Carlos Alberto. III. Título.

CDD: 005.35

Bibliotecário: Elias Barbosa da Silva CRB-9/1894



MINISTÉRIO DA EDUCAÇÃO
SETOR DE CIÊNCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA -
40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **RODRIGO GOMES LEMOS** intitulada: **Deteção de Ataques no Android - Um Modelo Baseado na Comunicação entre Processos**, sob orientação do Prof. Dr. CARLOS ALBERTO MAZIERO, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 14 de Setembro de 2021.

Assinatura Eletrônica

14/09/2021 15:08:50.0

CARLOS ALBERTO MAZIERO

Presidente da Banca Examinadora

Assinatura Eletrônica

14/09/2021 15:23:22.0

RAUL CERETTA NUNES

Avaliador Externo (UNIVERSIDADE FEDERAL DE SANTA MARIA)

Assinatura Eletrônica

14/09/2021 23:06:43.0

LUIZ CARLOS PESSOA ALBINI

Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

*À Marcelle Moreira de Azevedo
Aos meus pais, Edmo & Ednair*

AGRADECIMENTOS

Primeiramente, não poderia deixar de agradecer a minha esposa, Marcelle Moreira de Azevedo, por ter superado mais esse desafio ao meu lado. Agradeço também aos meus pais que sempre me apoiaram em todas as etapas de minha vida e com o mestrado não foi diferente.

Aos amigos e familiares, também sou muito grato por todo o apoio e incentivo prestado; especialmente ao Tiago Heinrich, que me acompanhou durante praticamente todo o período que estive na UFPR. Deixo também um agradecimento especial a chefia do 11º Centro de Telemática pelo suporte prestado.

Finalmente, gostaria de agradecer ao meu orientador, Carlos Maziero, por ter acreditado no meu potencial, compreendido minhas limitações de horário e estar sempre disponível para me auxiliar.

RESUMO

Atualmente o uso de celulares faz parte do cotidiano das pessoas e como consequência dessa popularização o Android se tornou o sistema operacional mais utilizado no mundo. Nesse contexto, os dispositivos Android são utilizados para conversas particulares, realizar operações bancárias, manter arquivos pessoais, entre outras atividades, se tornando assim um alvo atrativo para ataques hackers e malwares. Essa exposição a ações maliciosas gerou uma maior preocupação com a segurança de dispositivos Android e, consequentemente, ao desenvolvimento de diferentes métodos de proteção para os mesmos. Dentre esses métodos se destacam os sistemas de detecção de intrusão (IDS - *Intrusion Detection System*), que têm como objetivo identificar ameaças aos dispositivos. Existem diversas abordagens para a detecção de intrusão no Android, que podem ser classificadas em estáticas ou dinâmicas; onde os métodos estáticos têm como objetivo identificar ameaças através da análise dos arquivos de aplicativos sem executá-los e os métodos dinâmicos realizam essa atividade através da análise de dados coletados durante a execução da aplicação analisada. Além disso, uma das medidas de segurança adotada pelo Android é o isolamento de processos, que garante que um aplicativo não tenha acesso aos dados de outro; dessa forma, para que os aplicativos e componentes do Android se comuniquem é utilizado o módulo Binder, que implementa o IPC (*Inter Process Communication*) controlando toda a comunicação entre processos no Android. Apesar disso, o monitoramento do funcionamento do módulo Binder para detecção de ataques foi pouco explorada na literatura. Dessa forma, este trabalho apresenta uma metodologia para detecção de malware utilizando dados da comunicação entre processos como fonte de dados; esses dados foram coletados através do monitoramento de chamadas de funções do módulo Binder ao se executar a aplicação analisada. Utilizando-se esse mecanismo foi construída uma nova base de dados (AndroBind), que representa o comportamento de aplicações Android referente a comunicação com outros processos. Finalmente, a proposta do trabalho consiste no uso de métodos de aprendizado de máquina para identificar aplicações com comportamentos maliciosos através da base de dados de traços de chamadas de função no Binder construída. O modelo proposto alcançou como resultado uma taxa de identificação superior a 70%, validando assim a proposta. Além disso, também foi apresentada uma análise dos dados obtidos para identificar comportamentos maliciosos caracterizados nos mesmos, mostrando assim a relação entre o modo de uso de IPC e atividades maliciosas.

Palavras-chave: Android, Detecção de Intrusão, Binder, Detecção de Malware

ABSTRACT

Currently the use of smartphones and IoT devices is part of people's daily lives and due to this popularization Android has become the most used operating system in the world. That said, Android devices are used for private conversations, banking, maintaining personal files such as photos, among other activities, thus becoming an attractive target for hackers and malware. This exposure to malicious activities created a greater concern about Android devices security and, then, the development of different methods for protecting them. Among these methods, there are intrusion detection systems (IDSs), which aim to identify threats to devices. The different approaches to identifying threats on Android devices can be classified as static or dynamic; static methods aim to identify threats by analyzing application files without executing them and dynamic methods perform this goal by analyzing data collected during the execution of the analyzed application. Furthermore, one of the core security measures adopted by Android is the processes isolation, which ensures that an application does not have access to the data of another; then, for Android applications and components to communicate between them it is used the Binder module, which implements the IPC (Inter Process Communication) that allows it to manage all communication between processes on Android. Despite this, using the Binder module operational data to detect attacks has been little explored in the literature. Therefore, this work presents a methodology for intrusion detection that uses the inter-process communication as data source; this data is collected by monitoring function calls in the Binder module when executing the analyzed application. Using this mechanism a new dataset (AndroBind) was built, which represents the behavior of Android applications relative to communication with other processes. Finally, this work's approach consists of using machine learning methods to detect malicious behaviors in applications through this built dataset consisting of Binder function call traces. The model proposed achieved an over 70% identification rate and therefore validated the proposal. Besides that, it was introduced an analysis on the dataset in order to identify the malicious behaviors characterized on it and then show the relation about the way of using IPC and malicious activities.

Keywords: Android, Intrusion Detection, Binder, Malware Detection

LISTA DE FIGURAS

1.1	Evolução do <i>market share</i> de sistemas operacionais (StatCounter, 2021b)	12
2.1	Arquiteturas de IDS	18
2.2	Matriz de confusão	19
2.3	Exemplo de curvas ROC	20
3.1	Arquitetura da plataforma Android (Google, 2019b).	22
3.2	Comunicação entre processos através do Binder (Schreiber, 2011)	23
3.3	Aplicação dos diferentes contextos do Binder	25
3.4	Estrutura das <i>parcel</i> Binder	27
7.1	Sequência para gerar base de traços	45
7.2	Exemplo de <i>bag of words</i> (Zheng e Casari, 2018)	46
7.3	Exemplo de n-gramas (Stephanie, 2018)	46
7.4	Formato do traço gerado no <i>ftrace</i>	48
7.5	Histograma do tamanho dos traços	49
7.6	Histograma do tamanho dos traços após a retirada de <i>outliers</i>	50
7.7	Divisão da média de chamadas por aplicativos maliciosos e benignos.	51
8.1	<i>Pipeline</i> de execução	54
8.2	Evolução de <i>market share</i> das versões de Android nos últimos 12 meses (StatCounter, 2021a).	54
8.3	Curva ROC do modelo selecionado utilizando a base de validação	60
8.4	Curva ROC do modelo selecionado utilizando a base de testes	61

LISTA DE TABELAS

3.1	Identificadores de comandos Binder	26
5.1	Resumo dos trabalhos relacionados	38
8.1	Desempenho médio por representação dos dados	55
8.2	Desempenho médio por valor de n	55
8.3	Resultados na base original	56
8.4	Resultados na base considerando apenas comandos BC_*.	57
8.5	Resultados do modelo híbrido na base original.	57
8.6	Resultados do modelo híbrido na base considerando apenas comandos BC_*. . .	58
8.7	Desempenho e desvio padrão por representação	59
8.8	Desempenho do modelo selecionado na base de testes.	60

LISTA DE ACRÔNIMOS

TI	Tecnologia da Informação
API	Application Programming Interface
CPU	Central Processing Unit
IDS	Intrusion Detection System
HIDS	Host-based Intrusion Detection System
NIDS	Network-based Intrusion Detection System
CIDS	Collaborative Intrusion Detection System
IPS	Intrusion Prevention System
IDPS	Intrusion Detection and Prevention System
IDMEF	Intrusion Detection Message Exchange Format
P2P	Peer-to-Peer
ART	Android RunTime
JVM	Java Virtual Machine
IPC	Inter Process Communication
JNI	Java Native Interface
AIDL	Android Interface Definition Language
kNN	K-Nearest Neighbors
ICC	Inter Component Communication
NIV	Next Intent Vulnerability
SSL	Secure Socket Layer
DDoS	Distributed Denial of Service
ITU	International Telecommunications Union

SUMÁRIO

1	INTRODUÇÃO	12
1.1	MOTIVAÇÃO	13
1.2	OBJETIVO	13
1.3	ESTRUTURA DA PROPOSTA	13
2	DETECÇÃO DE INTRUSÃO	15
2.1	CLASSIFICAÇÃO PELO LOCAL DE COLETA DOS DADOS	16
2.2	CLASSIFICAÇÃO PELO MÉTODO DE DETECÇÃO	16
2.3	ARQUITETURA	17
2.4	DESEMPENHO	18
2.4.1	Métricas de avaliação	19
2.5	CONSIDERAÇÕES FINAIS	20
3	O SISTEMA ANDROID	21
3.1	BINDER	22
3.2	COMUNICAÇÃO ATRAVÉS DO BINDER	24
3.3	CONSIDERAÇÕES FINAIS	28
4	DETECÇÃO DE MALWARES NO AMBIENTE ANDROID	29
4.1	MALWARE	29
4.2	FORMAS DE ANÁLISE	30
4.2.1	Análise estática	30
4.2.2	Análise dinâmica	30
4.2.3	Análise híbrida	31
4.3	CONSIDERAÇÕES FINAIS	31
5	TRABALHOS RELACIONADOS	32
5.1	MODELOS DE ANÁLISE ESTÁTICA	32
5.2	MODELOS DE ANÁLISE DINÂMICA	34
5.3	MODELOS DE ANÁLISE HÍBRIDA	35
5.4	CONSIDERAÇÕES SOBRE OS MODELOS	36
5.5	CONSIDERAÇÕES FINAIS	37
6	PROPOSTA DA PESQUISA	40
6.1	JUSTIFICATIVA	40
6.2	OBJETIVOS	41
6.3	CONTEXTO DE APLICAÇÃO	41
6.4	CONSIDERAÇÕES FINAIS	42

7	OBTENÇÃO E REPRESENTAÇÃO DA BASE DE DADOS.	43
7.1	AQUISIÇÃO DA BASE DE DADOS	43
7.1.1	Base de Dados Androzoo	43
7.1.2	Geração da Base de Dados	44
7.2	TÉCNICAS DE REPRESENTAÇÃO DE DADOS.	45
7.2.1	Bag-of-Words	45
7.2.2	n-Gramas	46
7.2.3	Bag-of-n-Grams.	47
7.2.4	Tf-Idf	47
7.3	EXPLORAÇÃO E LIMPEZA DOS DADOS OBTIDOS	47
7.4	BASE DE DADOS HÍBRIDA	51
7.5	LIMITAÇÕES	52
7.6	CONSIDERAÇÕES FINAIS	52
8	AVALIAÇÃO DA PROPOSTA	53
8.1	<i>PIPELINE</i> DO MODELO DE APRENDIZADO DE MÁQUINA.	53
8.2	PARÂMETROS DE IMPLEMENTAÇÃO	54
8.3	ESCOLHA DA MELHOR REPRESENTAÇÃO	55
8.4	RESULTADOS DOS TESTES	56
8.5	AVALIAÇÃO DOS RESULTADOS	58
8.6	VALIDAÇÃO DO MODELO.	60
8.7	CONSIDERAÇÕES FINAIS	62
9	CONCLUSÃO	63
	REFERÊNCIAS	65

1 INTRODUÇÃO

Nos últimos anos houve um aumento significativo na quantidade e na sofisticação dos ataques cibernéticos. Segundo a ITU (*International Telecommunications Union*), em um período de 12 meses, medidos entre 2017 e 2018, o Brasil teve um prejuízo de US\$ 20 bilhões advindos de ataques cibernéticos (Alves, 2020).

Esse cenário gerou uma demanda de avanço e inovação dos métodos de defesa, mais particularmente nos sistemas de detecção de intrusão (*Intrusion Detection System* - IDS) e sistemas de proteção de intrusão (*Intrusion Protection System* - IPS). Esses sistemas são dispositivos ou softwares responsáveis por monitorar atividades maliciosas e as reportar, sendo os IPS também responsáveis por tomar ações contra as mesmas.

Paralelamente à evolução dos hackers, o uso de dispositivos móveis aumentou significativamente e o Android se tornou o sistema operacional mais utilizado no mundo, conforme pode ser observado na imagem 1.1 que mostra a evolução do *market share* de sistemas operacionais nos últimos 10 anos.

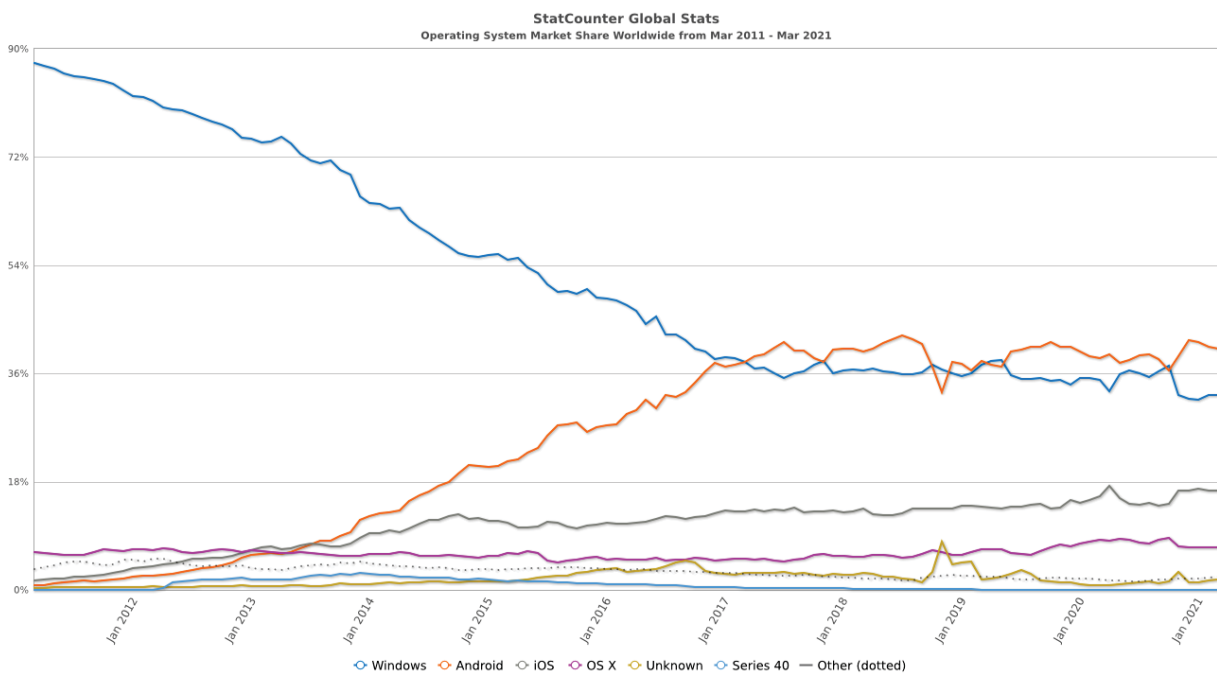


Figura 1.1: Evolução do *market share* de sistemas operacionais (StatCounter, 2021b)

Com isso, os dispositivos Android se tornaram alvos frequentes de ataques cibernéticos e consequentemente se tornou necessário o desenvolvimento de mais mecanismos de defesa para o mesmo. Essa necessidade é acentuada pelo caráter privado e, por vezes, confidencial das informações contidas em dispositivos Android como os *smartphones*.

Entre as defesas existentes para o Android estão o uso do princípio do privilégio mínimo para os aplicativos, sistema de permissões, diversos antivírus, entre outros. Nesse contexto os sistemas de detecção de intrusão para Android ganharam importância e se encontram em constante evolução.

1.1 MOTIVAÇÃO

Os IDSs são dispositivos utilizados para aumentar a segurança de sistemas, evitando o sucesso de ataques e consequentemente aumentando a segurança de dados. Com isso, o desenvolvimento de IDSs para o Android se tornou uma prioridade, uma vez que o Android além de ser o sistema operacional mais utilizado do mundo, é responsável por armazenar dados privados como: fotos, mensagens, aplicativos bancários, etc.

O Android é um sistema operacional baseado em Linux utilizado principalmente em dispositivos móveis. Entretanto, o kernel Linux do Android possui alterações em sua arquitetura; em particular no Android cada processo é executado isoladamente e com privilégio mínimo, sendo necessário um mecanismo para comunicação entre processos.

Para suprir essa necessidade, o sistema Android utiliza um módulo de núcleo chamado *Binder*. O Binder é responsável por gerenciar toda a comunicação entre processos no Android (Yeon et al., 2016). Dessa forma, qualquer requisição de aplicativo a recursos externos ao próprio é intermediada pelo Binder; são exemplos de requisições desse tipo: acesso a recursos do hardware, requisições à internet, requisições a outros aplicativos, acesso a arquivos de mídia, etc.

O mecanismo de comunicação entre processos no Android é relevante na execução de atividades maliciosas através de aplicativos, pois caso o aplicativo não utilize o mesmo, suas ações serão restritas apenas aos dados e atividades do próprio aplicativo. Apesar disso, foi observado que não existe na literatura uma proposta que considere as informações de execução do Binder como fonte de dados para detecção de aplicativos maliciosos.

1.2 OBJETIVO

O objetivo principal deste trabalho consiste em avaliar a viabilidade do uso de informações sobre a comunicação entre processos no Android como fonte de dados para detecção de aplicativos maliciosos. Não se encontra no escopo desse trabalho a detecção de ataques que envolvam mais de um aplicativo para acontecer, como ataques em conluio.

Os dados referentes à comunicação entre processos no Android serão coletados como traços de execução do Binder, pois se trata do módulo do Kernel responsável por intermediar toda a comunicação entre processos no Android.

Para possibilitar o cumprimento do objetivo principal definido são necessários passos intermediários no desenvolvimento da pesquisa. Dessa forma, foram definidos os seguintes objetivos parciais:

- definir um mecanismo para extrair informações do Binder;
- analisar as informações coletadas e identificar as mais relevantes para a detecção de aplicações maliciosas;
- identificar os procedimentos de aprendizado de máquina mais adequados para esse contexto;
- construir um protótipo para avaliar a efetividade e limitações da proposta.

1.3 ESTRUTURA DA PROPOSTA

O capítulo 2 apresenta os conceitos relacionados a detecção de intrusão e seus sistemas relacionados. O capítulo 3 apresenta o sistema operacional Android, com ênfase em seu

mecanismo de comunicação entre processos. O capítulo 4 apresenta os conceitos de malware, enquanto o capítulo 5 apresenta os conceitos de detecção de malware para o ambiente Android e os trabalhos relacionados a esse assunto que compõem o estado da arte no mesmo. O capítulo 6 contém as especificações da proposta e seu contexto, além da análise dos dados obtidos. O capítulo 7 apresenta os resultados encontrados e a discussão sobre os mesmos. Finalmente, o capítulo 8 contém a conclusão do trabalho e proposta de trabalhos futuros.

2 DETECÇÃO DE INTRUSÃO

A detecção de intrusão monitora e analisa sistemas computacionais de modo a identificar atividades maliciosas. Caso a identificação de determinada atividade maliciosa seja realizada através da análise do software em si, se trata de detecção de malware; a detecção de malware é um conceito que abrange tanto a análise de ações e efeitos (análise de malware dinâmica), quanto a análise do software em si (análise de malware estática).

Os sistemas de detecção de intrusão são sistemas utilizados para identificar ações maliciosas no ambiente que está sendo monitorado, reportá-las e possivelmente ter alguma ação preventiva em relação à ameaça identificada. Caso o sistema execute alguma ação de prevenção às ameaças, o mesmo se trata de um IDPS (*Intrusion Detection and Prevention System*) ou simplesmente IPS (*Intrusion Prevention System*).

Um sistema de detecção de intrusão pode se tratar de um software ou de um hardware dedicado a essa função. Um IDS genérico possui três componentes principais: coletor de dados, conversor de dados e motor de decisão (Bridges et al., 2020).

O coletor de dados tem como função receber os dados que estão sendo monitorados pelo IDS e armazená-los. Esses dados podem ser coletados em diversos pontos do ambiente computacional monitorado de acordo com o tipo de IDS e podem se tratar de fluxos de rede, logs, chamadas de sistema, uso de recursos, etc.

O conversor de dados tem como função tratar os dados coletados para deixá-los no formato certo para o seu processamento pelo classificador. Esse processo inclui filtrar na base de dados os parâmetros utilizados no processo de decisão (parâmetros escolhidos no processo de seleção de atributos) e formatar os dados na representação predefinida (Bridges et al., 2020). O produto desse processo é chamado de vetor de atributos.

O motor de decisão tem como função processar o vetor de atributos para decidir se os dados coletados representam uma ação maliciosa ou não. Os dados são processados através de algoritmos ou heurísticas que utilizam aprendizado de máquina, processos estatísticos, comparação com assinaturas conhecidas, etc.

Para coletar dados de possíveis ataques, o IDS pode agir de forma completamente passiva ao se limitar ao monitoramento ou agir de forma ativa através do uso de *honeypots*. *Honeypot* se trata de um dispositivo ou sistema desprotegido usado como "isca", ou seja, seu objetivo é ser atacado e comprometido para coletar informações sobre o atacante. O uso de *honeypots* aumenta a precisão do IDS (Kabiri e Ghorbani, 2005).

Após a coleta dos dados, o IDS pode realizar a sua análise de modo *online* ou *offline*. A detecção *online* tem como objetivo identificar uma intrusão antes que seja causado dano ao sistema. Neste caso, a análise dos dados é feita em tempo real e consequentemente o algoritmo utilizado não pode ser muito custoso, para não prejudicar o desempenho do sistema. Já a detecção *offline* tem como objetivo identificar uma intrusão com dados previamente armazenados, sem o compromisso com o tempo decorrido entre o ataque e a detecção. Neste caso, podem ser usados algoritmos mais custosos e consequentemente mais minuciosos, possibilitando a detecção de ataques mais discretos.

Os sistemas de detecção de intrusão se diferenciam e podem ser classificados de acordo com o local de coleta de dados, funcionalidade, mecanismo de detecção e abordagem de implantação (Pharate et al., 2015). Nesta revisão serão explorados os seguintes aspectos: local de coleta dos dados e mecanismo de detecção.

2.1 CLASSIFICAÇÃO PELO LOCAL DE COLETA DOS DADOS

A fonte de dados do IDS e sua posição no sistema monitorado influenciam os resultados do mesmo; isso ocorre pois o IDS depende dos dados disponíveis para detecção, além disso ele influencia o desempenho do ambiente monitorado de maneiras diferentes, dependendo de onde foi implementado. Sendo assim, a posição do IDS é uma decisão crítica de arquitetura (Bridges et al., 2020).

De acordo com o local de implementação do sistema de detecção de intrusão ele pode ser classificado em: baseado em host (HIDS), baseado em rede (NIDS) ou híbrido, onde são levados em consideração dados do host e da rede.

HIDS normalmente são implementados como um software instalado diretamente no host e podem ser responsáveis por monitorar um programa específico ou o sistema operacional do dispositivo como um todo. Os HIDS têm como vantagem o acesso a dados de qualidade, por exemplo chamadas de sistemas e logs de aplicações, permitindo assim um melhor entendimento do estado do sistema monitorado (Nisioti et al., 2018). Vale ressaltar, que o host é o local de coleta dos dados, porém não necessariamente o local de análise dos dados; existem HIDS com análise remota para não sobrecarregar os processadores do host, utilizando a nuvem por exemplo (Kumar e Hanumanthappa, 2015).

Entretanto os HIDS têm como desvantagens o não isolamento do host e os custos no desempenho do host. O não isolamento do host possibilita que um atacante com acesso ao mesmo possa enganar ou até desabilitar o IDS (Bridges et al., 2020). Como o HIDS é instalado diretamente no host monitorado, ele também poderá gerar um impacto significativo na performance do host para processar os dados ou na rede utilizada pelo mesmo, caso seja utilizado o processamento remoto (Nisioti et al., 2018). Além disso, no caso de um ambiente de TI grande, para proteger toda a rede seria necessário instalar o HIDS em cada dispositivo da mesma, o que gera um custo operacional considerável.

Já os NIDS normalmente são implementados através de um hardware dedicado colocado na borda da rede para conseguir monitorar o tráfego de entrada e saída da rede inteira. Os NIDS coletam e analisam dados de rede e podem fazê-lo com base apenas no fluxo de dados ou levando em conta o conteúdo do tráfego ao inspecionar cada pacote. Apesar dos dados de rede não permitirem visualizar o estado do sistema monitorado diretamente, eles permitem monitorar a rede inteira através de um único dispositivo e detectar ataques coordenados, por exemplo um DDoS (*Distributed Denial of Service*).

Além disso, os NIDS funcionam independentemente dos sistemas operacionais e plataformas utilizadas, pois eles simplesmente monitoram e processam os dados da rede, sendo assim completamente isolados e transparentes para os sistemas na rede (Nisioti et al., 2018). Como desvantagem, os NIDS não possuem informações sobre os sistemas, tornando assim mais difícil a detecção de intrusões e o tráfego de rede encriptado também dificulta a sua atuação (Bridges et al., 2020).

Os sistemas híbridos consideram dados coletados nos hosts e no tráfego na borda da rede para detectar intrusões. Para isso esses sistemas se utilizam de arquiteturas com monitores distribuídos que compartilham seus dados para uma análise conjunta. Os tipos de arquiteturas colaborativas serão abordadas na seção 2.3.

2.2 CLASSIFICAÇÃO PELO MÉTODO DE DETECÇÃO

Após a coleta de dados, o motor de decisão de um IDS pode processar os dados utilizando diferentes heurísticas ou algoritmos para classificar o comportamento analisado em

malicioso ou não (Bridges et al., 2020). Esses métodos são divididos em baseados em anomalia e baseados em assinaturas.

Os métodos baseados em assinatura consistem em comparar o comportamento observado com comportamentos maliciosos salvos em uma base de dados; essa base de dados mantém as assinaturas dos ataques conhecidos. Logo, a eficiência de um IDS baseado em assinaturas depende da qualidade de sua base de dados, uma vez que ataques não conhecidos pela mesma não serão detectados (Jyothsna et al., 2011).

A vantagem dos métodos baseados em assinaturas é a baixa ocorrência de falsos positivos, uma vez que dificilmente um comportamento normal seria equivalente a um comportamento malicioso da base de dados. Sua desvantagem é a impossibilidade de detectar ataques novos, cujas assinaturas ainda não foram adicionadas à base de dados do método.

Os métodos baseados em anomalia consistem em detectar comportamentos que diferem do comportamento normal do sistema. Um comportamento é considerado como anomalia se o seu desvio do comportamento normal do sistema for superior a um limiar predefinido. Essa estratégia de detecção pode provocar muitos falsos positivos, uma vez que qualquer comportamento diferente do normal será alertado como uma anomalia; entretanto como essa estratégia não se baseia em assinaturas de ataques, ela é capaz de detectar ataques desconhecidos até então (Jyothsna et al., 2011).

De acordo com o tipo de algoritmo usado, os métodos de detecção por anomalia podem ser divididos em: estatísticos, supervisionados (classificação), não supervisionados (detecção de *outlier*, clusterização), inteligência artificial e combinações de métodos (Nisioti et al., 2018).

2.3 ARQUITETURA

Inicialmente, os IDSs eram implementados como um hardware ou software isolado responsável por monitorar e analisar o comportamento de uma única rede ou host. Sem comunicação entre essas várias instâncias de IDS, não era possível relacionar eventos maliciosos ocorrendo no ambiente ao mesmo tempo e consequentemente não era possível detectar ataques sofisticados e distribuídos (Vasilomanolakis et al., 2015).

Para possibilitar a detecção desse tipo de ataque foram desenvolvidos os IDS colaborativos (CIDS), que consistem de monitores e unidades de análise de dados que compartilham informações de acordo com uma arquitetura. Essa arquitetura pode ser centralizada, distribuída ou hierárquica, conforme mostra a figura 2.1.

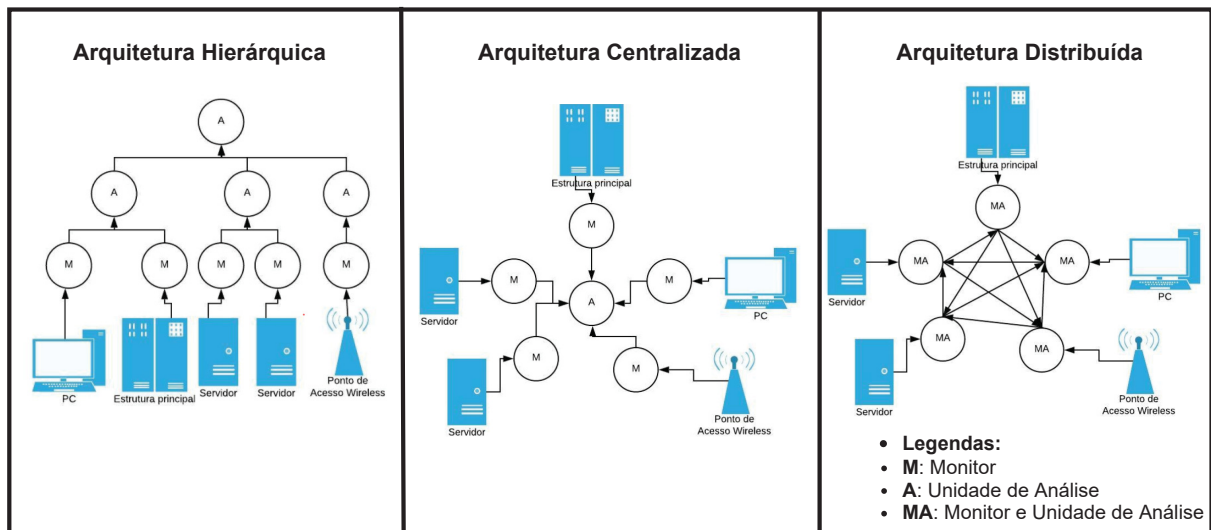


Figura 2.1: Arquiteturas de IDS

Na arquitetura centralizada existe apenas uma unidade de análise de dados e todos os dados coletados pelos monitores são enviados à mesma para a tomada de decisões. As desvantagens dessa arquitetura são a sobrecarga computacional em apenas um ponto do sistema (unidade de análise de dados) e consequentemente a não escalabilidade do mesmo.

Na arquitetura distribuída cada nó do sistema é composto por um monitor e uma unidade de análise e os nós compartilham suas informações com todos os demais nós. A maior desvantagem dessa arquitetura é que o tempo de detecção de um ataque distribuído é longo, pois cada IDS tem que receber e analisar os resultados dos outros IDS para identificar um ataque desse tipo; entretanto ele distribui a sobrecarga em todos os pontos do sistema, não existindo assim um ponto único de falha.

Na arquitetura hierárquica os dados são coletados e encaminhados através de camadas de análise de dados, onde os dados são pré-processados e correlacionados até convergirem para uma unidade central de análise dos dados. Dessa forma, a arquitetura hierárquica não sobrecarrega a unidade central e possui um atraso menor do que a arquitetura distribuída para detectar ataques distribuídos em ambientes de TI grandes; entretanto, caso algum nó da camada de análise de dados falhe, o sistema perderá informações sobre diversos ativos monitorados.

De forma a possibilitar a comunicação entre diferentes IDSs e sistemas de gerenciamento dos mesmos foi desenvolvido um protocolo que padroniza o formato dos dados compartilhados e o método de compartilhamento dos mesmos. Esse algoritmo é o IDMEF (*Intrusion Detection Message Exchange Format*) e permite o compartilhamento de alertas simples e complexos sem ambiguidades (Debar et al., 2007).

2.4 DESEMPENHO

O processo de classificação feito por um IDS tem como premissa que um ataque ou ação maliciosa implica em um comportamento específico ou diferente do comportamento normal. Entretanto, essa premissa pode gerar erros de decisão no IDS. Dessa forma, o desempenho de um IDS é medido através de métricas de avaliação. Essas métricas são calculadas com base na quantidade de falsos positivos e falsos negativos, definidos de acordo com a matriz de confusão da figura 2.2.

		Valor Previsto	
		Positivo	Negativo
Valor Verdadeiro	Positivo	Verdadeiro Positivo	Falso Negativo
	Negativo	Falso Positivo	Verdadeiro Negativo

Figura 2.2: Matriz de confusão

A quantidade de decisões falsas do IDS depende, além da corretude do algoritmo utilizado, da sensibilidade adotada (Azeez et al., 2019). A sensibilidade de um IDS se trata do limite utilizado para decidir se determinado comportamento é malicioso ou não, ou seja, é o parâmetro que define o quanto um comportamento tem que ser similar a uma assinatura maliciosa ou diferente do comportamento normal do sistema para ser considerado um comportamento malicioso.

Uma sensibilidade muito baixa implica em um alto número de falsos negativos e consequentemente um alto número de ameaças não detectadas; já uma sensibilidade muito alta implica em um alto número de falsos positivos e consequentemente uma menor credibilidade nas decisões do IDS, por reportar vários comportamentos comuns como maliciosos (Azeez et al., 2019). Sendo assim, a sensibilidade do IDS deve ser calibrada de acordo com o nível de criticidade do sistema monitorado para otimizar os resultados em cada caso.

2.4.1 Métricas de avaliação

Para se avaliar um modelo, deve-se escolher a métrica adequada para o mesmo, uma vez que elas apresentam uma perspectiva diferente do problema abordado (Ceschin et al., 2019). Isso ocorre pois ao se forçar uma melhora em uma das métricas pode-se piorar o resultado medido por outra métrica. Serão abordadas as métricas baseadas nos valores da matriz de confusão: acurácia, *recall*, precisão, curvas ROC (*Receiver Operating Characteristic*), AUC (*Area Under Curve*) e *f1score*.

A acurácia mede a porcentagem das amostras que o classificador acertou, sendo assim expressa pela fórmula: $Acc = (TP + TN) / (TP + TN + FN + FP)$. Essa métrica tem valor entre 0 e 1 e quanto mais próximo de 1 melhor o resultado do classificador. Entretanto, a mesma não é recomendada para problemas desbalanceados, porque ela permite que o classificador erre todos os exemplares da classe minoritária sem grande impacto na métrica (Japkowicz e Shah, 2011).

O *recall* mede a razão entre o número de verdadeiros positivos medidos pelo modelo e o número de positivos existentes na amostra, sendo assim expresso pela fórmula: $Rec = TP / (TP + FN)$. Já a precisão mede a razão entre o número de verdadeiros positivos e os número de amostras classificadas como positivas pelo classificador, sendo expresso pela fórmula: $Prec = TP / (TP + FP)$. Ambas as métricas tem valor entre 0 e 1 e quanto mais próximas de 1 melhor o resultado, entretanto existe um *tradeoff* entre as duas.

O *f1score* é uma métrica que combina *recall* e precisão com objetivo de considerar o valor de ambas na medida e é dada pela média harmônica das duas métricas: $f1 = 2 * (Prec * Rec) / (Prec + Rec)$.

As curvas ROC plotam o valor do *recall* em relação a taxa de falsos positivos para cada taxa de falsos positivos considerada ao se mover o limite do classificador para escolher entre as classes positivas e negativas (Japkowicz e Shah, 2011). Sendo assim, é uma curva que apresenta o desempenho do classificador em todos seus pontos de operação, conforme exemplificado na imagem 2.3 que apresenta cinco curvas ROC.

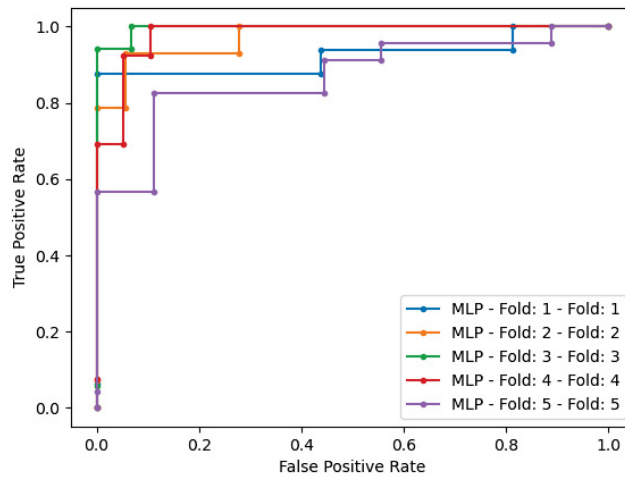


Figura 2.3: Exemplo de curvas ROC

A métrica AUC mede a área abaixo da curva ROC e é normalmente utilizada em detrimento da acurácia por ter um desempenho melhor com bases de dados desbalanceadas. Entretanto, essa métrica também possui problemas, como por exemplo, a possibilidade de as curvas ROC se cruzarem, não permitindo uma comparação adequada do desempenho dos classificadores através do AUC (Japkowicz e Shah, 2011). Essa métrica também varia entre 0 e 1 e quanto mais próximo de 1 melhor o resultado.

Além das métricas baseadas na matriz de confusão apresentadas, também será usado o *Brier Score* que mede a perda do modelo avaliado; em casos unidimensionais ele é equivalente ao erro quadrático médio. O seu valor varia entre 0 e 1 e quanto mais próximo de 0 melhor o resultado, pois ele é uma medida do erro de predição em relação aos resultados verdadeiros.

2.5 CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentados os principais conceitos relacionados à detecção de intrusões, abordando o conceito de detecção de intrusão, a estrutura geral de sistemas de detecção de intrusão e suas classificações de acordo com o local de coleta dos dados ou método de detecção; além disso, é mostrada também uma análise qualitativa de como a sensibilidade de um IDS pode afetar seu desempenho influenciando a quantidade de resultados falsos.

3 O SISTEMA ANDROID

O Android é um sistema operacional desenvolvido principalmente para dispositivos móveis, como celulares e tablets, entretanto ele também é utilizado em TVs, carros, relógios, entre outros. Ele foi desenvolvido pela *Open Handset Alliance*, que se trata de um consórcio de empresas para criar padrões abertos para telefonia móvel, sendo seu principal colaborador o Google (Rogers et al., 2009).

O Android é uma pilha de software *open source* baseado em Linux com arquitetura baseada em camadas de abstração. As cinco camadas da plataforma Android são: núcleo do Linux, camada de abstração do hardware, bibliotecas nativas/Android Runtime (ART), API Java e aplicativos do sistema (Google, 2019b). Essa estrutura está representada na figura 3.1.

O núcleo do Linux é a base do sistema Android e seus drivers são responsáveis pela interação direta com o hardware. Isso permite que o Android se utilize de funcionalidades e recursos de segurança do Linux, como o SELinux e o sistema de permissões UNIX, por exemplo.

A camada de abstração de hardware é composta por interfaces para componentes específicos de hardware que são utilizadas pelas APIs Java fazer chamadas a esses componentes.

As bibliotecas nativas são implementadas em C/C++ e são responsáveis por implementar serviços e sistemas principais do Android. O ART é responsável por implementar a máquina virtual utilizada por cada aplicativo Android; o ART é semelhante à JVM, entretanto é baseado em registradores e consequentemente mais rápido.

As APIs Java permitem que os desenvolvedores de aplicativos tenham acesso aos recursos do sistema operacional, permitindo assim a reutilização de componentes e serviços, como por exemplo as bibliotecas nativas. Já os aplicativos do sistema são os programas disponíveis aos usuários da plataforma, como o gerenciador de mensagens ou visualizador de vídeos, por exemplo.

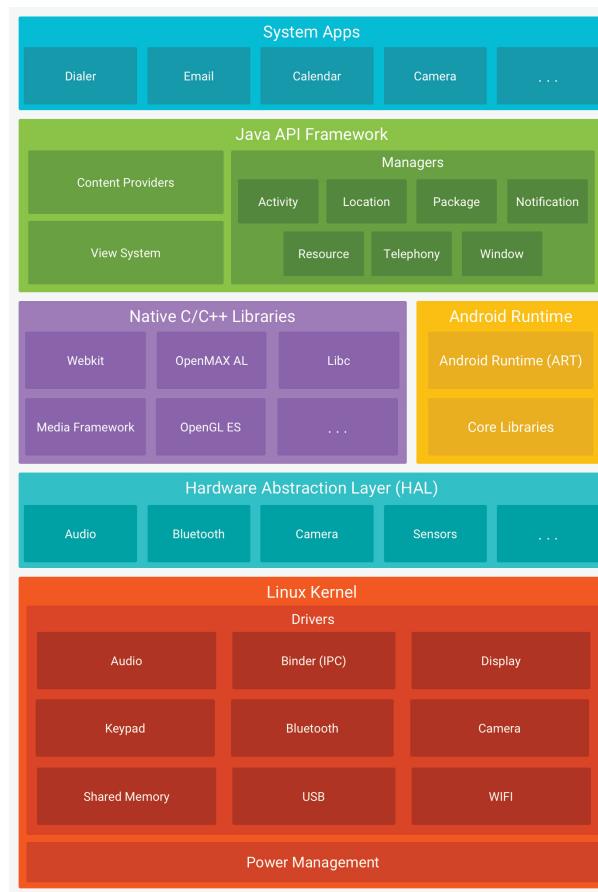


Figura 3.1: Arquitetura da plataforma Android (Google, 2019b)

Essa arquitetura de sistema operacional permite que o Android implemente o conceito de privilégio mínimo em seus aplicativos através do uso de sandbox e do sistema de permissões.

No Android as permissões são utilizadas para autorizar ou não que um aplicativo utilize recursos de outro. As permissões são categorizadas em quatro níveis, de acordo com o protocolo que as concede: normais, para permissões que não oferecem riscos à privacidade do usuário; de execução, para permissões perigosas para a privacidade do usuário (devem ser autorizadas pelo usuário manualmente); de assinatura, para aplicativos assinados com o mesmo certificado; e especiais, para aplicativos assinados com o mesmo certificado da imagem do sistema operacional (Kumar e Shukla, 2020).

Para garantir que cada aplicativo tenha acesso apenas aos seus recursos, o Android utiliza o conceito de *sandbox*, em que cada instância de um aplicativo é executada em uma máquina virtual (ART) isolada com um *user id* e *group id* únicos e que permitem acesso somente à pasta do próprio aplicativo. Dessa forma, os aplicativos Android são executados de maneira isolada do resto do sistema.

Para se comunicar com outros aplicativos e com os recursos do sistema e hardware, os aplicativos Android utilizam mecanismos de IPC (*Inter Process Communication*). Esse mecanismo de comunicação entre processos é implementado no Android através de um módulo do núcleo Linux denominado Binder, que será detalhado na próxima seção.

3.1 BINDER

Para implementar o princípio do privilégio mínimo aos aplicativos e manter a segurança do sistema operacional, o Android isola cada aplicativo em um processo diferente, de modo

que outros aplicativos não tenham acesso ao seu espaço de memória. Esse isolamento garante a segurança da memória dos aplicativos, entretanto os aplicativos precisam comunicar entre si e com os recursos de hardware para funcionar (Schreiber, 2011).

Isso posto, existe a necessidade de um mecanismo de comunicação entre os processos. O mecanismo adotado para atender a essa demanda é o Binder. O Binder se trata de um módulo do kernel que foi baseado no projeto OpenBinder da Be Inc e posteriormente Palm Inc; o projeto tinha como objetivo: “fornecer uma abstração de alto nível mais rica sobre os serviços tradicionais do sistema operacional moderno” (Inc., 2005).

O Binder, além de permitir a comunicação entre diferentes aplicativos, é o único recurso do Kernel do Android disponível diretamente para os aplicativos. Dessa forma, ele é o ponto único de troca de informações e requisições de serviços dos aplicativos.

A comunicação através do Binder ocorre em um modelo cliente-servidor, envolvendo uma thread do aplicativo do usuário como cliente, uma thread do serviço que oferece o recurso solicitado como servidor e o Binder no Kernel como intermediário (Shen et al., 2017). Essa comunicação entre processos ocorre seguindo os seguintes passos, ilustrados na figura 3.2:

1. A thread do aplicativo cliente, através de um proxy, envia uma requisição para o Binder;
2. O Binder armazena os dados da requisição do cliente e acorda uma thread do servidor em questão para responder a requisição;
3. A thread acordada processa a requisição e envia a resposta ao Binder;
4. O Binder identifica a thread cliente através dos dados salvos no passo 2 e encaminha a resposta para a mesma.

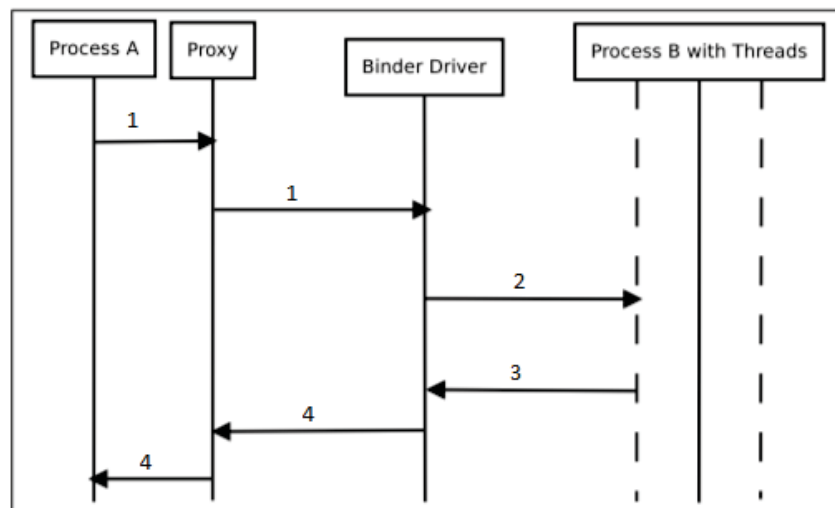


Figura 3.2: Comunicação entre processos através do Binder (Schreiber, 2011)

Essas mensagens trocadas entre os aplicativos e o Binder são chamadas *transactions* e podem transportar diferentes tipos de dados, como descritores de arquivos, inteiros, objetos locais e remotos, etc. Para transferir essas estruturas complexas de um processo para o outro, o Binder possui diferentes camadas de abstração, que vão desde o espaço do usuário até o espaço do Kernel.

Na camada de aplicação da arquitetura Android, existem diferentes implementações que permitem acesso ao Binder para os aplicativos: *intents*, *Content Providers*, *Broadcast Receiver*,

System Service Manager, AIDL (*Android Interface Definition Language*), *Messengers*. Dessa forma, a aplicação cliente pode realizar chamadas assíncronas, chamadas síncronas através de CRUD (*Create, Read, Update, Delete*), receber solicitações de serviços de outros aplicativos, definir o comportamento dos proxy e stub de determinado serviço, solicitar serviços oferecidos por outros aplicativos ou de hardware, etc, através das funcionalidades do Binder de modo transparente para o mesmo, tendo acesso assim a serviços remotos como se fossem locais do processo.

A desvantagem de se utilizar uma ferramenta de comunicação entre processos como o Binder foi descrita por (Artenstein e Revivo, 2014), onde ele mostra que o Binder é um ponto único de falha na arquitetura Android. Isso quer dizer que se o Binder for comprometido por um ataque, todo o sistema será comprometido também, uma vez que todas as informações trocadas entre processos no Android trafegam pelo Binder. Tal afirmação foi reforçada posteriormente por (Shen et al., 2017), onde ele propôs alterações do Binder de modo a diminuir ainda mais suas vulnerabilidades e aumentar a segurança do Android.

3.2 COMUNICAÇÃO ATRAVÉS DO BINDER

No Android existem três diferentes tipos de processos: processos de aplicação (*app processes*), estrutura (*framework processes*) e fornecedor (*vendor processes*). Os processos de aplicação são responsáveis por executar os aplicativos instalados, por exemplo o aplicativo de câmera. Os processos de estrutura são responsáveis por executar os processos do sistema Android, por exemplo o processo *system server*, e logo são independentes do dispositivo. Já os processos de fornecedor são dependentes do dispositivo e responsáveis pelas camadas de abstração do *hardware*.

Primeiramente, deve-se observar que, a partir da versão 8, o Android oferece suporte a três contextos diferentes para realizar a comunicação entre processos: */dev/binder*, */dev/hwbinder* e */dev/vndbinder*. Esses contextos são usados respectivamente para: IPC entre processos de estrutura/aplicação com interfaces AIDL; IPC entre processos de estrutura/fornecedor com interfaces HIDL (*HAL Interface Description Language*); e IPC entre processos do fornecedor com interfaces AIDL (Google, 2020). O uso desses três domínios está exemplificado na imagem 3.3, onde HAL é a camada de abstração de hardware.

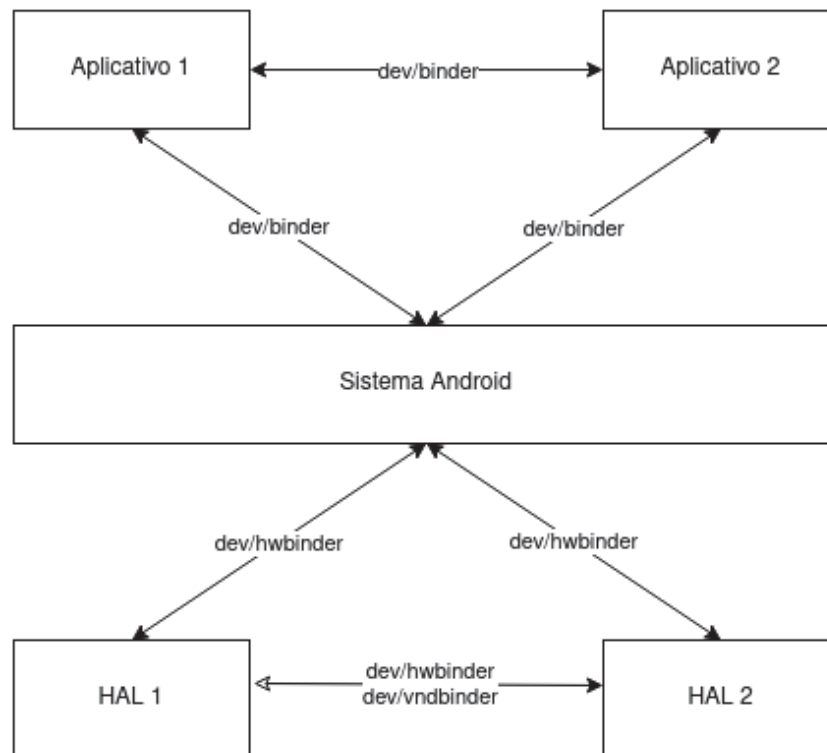


Figura 3.3: Aplicação dos diferentes contextos do Binder

Dessa forma, o Android isola as comunicações de processos de aplicações, do sistema e do fornecedor. O escopo desse trabalho se limita a comunicação iniciada por aplicativos, dessa forma será detalhada a seguir a comunicação que ocorre no */dev/binder*.

Como o Android é um sistema operacional baseado em Linux, portanto na camada de abstração mais alta da comunicação entre processos são utilizadas chamadas `ioctl` para se comunicar com o módulo Binder. No Kernel, o Binder executa as tarefas mais críticas envolvidas na comunicação entre processos: procurar o serviço requisitado; acordar uma thread do servidor encontrado; copiar os dados para o processo servidor; e manter um mapeamento entre as respostas dos servidores e as threads corretas de cada processo cliente (Artenstein e Revivo, 2014). Para implementar esse controle de threads, o Binder mantém os objetos Binder compartilhados, contendo referências fracas e fortes aos processos, em uma árvore rubro-negra com um complexo mecanismo de contagem de referências que indica se o objeto em questão está sendo usado para realizar alguma operação.

O Binder aceita os seguintes comandos: `BINDER_SET_MAX_THREAD`, `BINDER_SET_CONTEXT_MGR`, `BINDER_THREAD_EXIT`, `BINDER_VERSION`, `BINDER_WRITE_READ`. Para a comunicação entre processos, é utilizado o `BINDER_WRITE_READ`; os demais comandos são utilizados para definir parâmetros, mostrar versão e quando uma thread sai do *pool*. Dessa forma, nessa camada de abstração a comunicação entre processos ocorre apenas através do envio e recebimento de chamadas `ioctl` com o comando `BINDER_WRITE_READ` pelo Binder.

Indo um pouco mais a fundo, para se enviar essa chamada deve-se preencher uma estrutura de dados que contém os buffers de leitura e escrita e aponta para uma outra estrutura de dados que contém o objeto `binder_transaction_data` prefixado por um identificador de comando.

Os identificadores de comando do Binder são responsáveis por indicar o tipo de ação a ser executada pelo Binder, e portanto se tratam de uma nova camada de abstração na comunicação

entre processos. Esses comandos são da forma BC_* e BR_*, que são referências às funções do módulo Binder `binder_command` e `binder_return`. Os comandos do tipo BC_* são utilizados pelos processos para se enviar requisições ao Binder e os comandos do tipo BR_* são utilizados pelo Binder para enviar requisições aos processos. Existem 37 identificadores de comandos Binder, apresentados na tabela 3.1.

Tabela 3.1: Identificadores de comandos Binder

Identificador de comando	Funcionalidade
BR_ERROR	Binder informa que ocorreu um erro na operação
BR_OK	Operação completa
BR_TRANSACTION	A thread do processo alvo foi acordada e recebeu uma transação do Binder
BR_REPLY	Resposta enviada pelo Binder ao processo requisitante
BR_ACQUIRE_RESULT	Responde ao BC_ATTEMPT_ACQUIRE com a referência forte para o objeto solicitado ou com zero caso a operação falhe
BR_ATTEMPT_ACQUIRE	Requisita a referência de um objeto ao processo
BR_DEAD_REPLY	Binder informa o processo requisitante que o processo alvo morreu
BR_TRANSACTION_COMPLETE	Binder informa ao processo requisitante que o pedido foi aceito e será processado
BR_INCREFS	Pedido para adicionar 1 ao índice da referência fraca do objeto que representa esse nó
BR_ACQUIRE	Pedido para adicionar 1 ao índice da referência forte do objeto que representa esse nó
BR_RELEASE	Pedido para subtrair 1 ao índice da referência forte do objeto que representa esse nó
BR_DECREFS	Pedido para subtrair 1 ao índice da referência fraca do objeto que representa esse nó
BR_NOOP	Não faz nada (sem operação)
BR_SPAWN_LOOPER	O Binder envia esse comando para que o processo alvo gere uma nova thread para o serviço, pois não encontrou threads disponíveis
BR_FINISHED	Parar thread do <i>pool</i> de threads
BR_DEAD_BINDER	Notifica ao processo que o objeto proxy do Binder morreu
BR_CLEAR_DEATH_NOTIFICATION_DONE	Notifica que a limpeza da notificação de morte foi concluída
BR_FAILED_REPLY	Informa ao processo requisitante que a solicitação falhou (falta de memória, processo alvo não existe, etc)
BC_TRANSACTION	Envia requisição ao Binder (mantido separado do BC_TRANSACTION_SG para fins de compatibilidade com clientes Binder antigos)
BC_TRANSACTION_SG	Envia requisição ao Binder (tem suporte ao envio de buffers de memória brutos sem serializá-los em parcel)
BC_REPLY	Envia ao Binder a resposta a uma requisição do mesmo (mantido separado do BC_REPLY_SG para fins de compatibilidade com clientes Binder antigos)

Identificador de comando	Funcionalidade
BC_REPLY_SG	Envia ao Binder a resposta a uma requisição do mesmo (tem suporte ao envio de buffers de memória brutos sem serializá-los em <i>parcel</i>)
BC_ACQUIRE_RESULT	Responde ao BR_ATTEMPT_ACQUIRE com a referência forte para o objeto solicitado ou zero caso a operação falhe
BC_FREE_BUFFER	Notifica o Binder para liberar o buffer determinado
BC_INCREFS	Adiciona 1 ao índice da referência fraca do objeto
BC_ACQUIRE	Adiciona 1 ao índice da referência forte do objeto
BC_RELEASE	Subtrai 1 ao índice da referência forte do objeto
BC_DECREFS	Subtrai 1 ao índice da referência fraca do objeto
BC_INCREFS_DONE	Responde o pedido feito pelo Binder através do comando BR_INCREFS
BC_ACQUIRE_DONE	Responde o pedido feito pelo Binder através do comando BR_ACQUIRE
BC_ATTEMPT_ACQUIRE	Requisita a referência de um objeto ao Binder
BC_REGISTER_LOOPER	Muda o status do binder looper para BINDER_LOOPER_STATE_REGISTERED; indica ao Binder que essa thread filha está pronta para receber mensagens
BC_ENTER_LOOPER	Muda o status do binder looper para BINDER_LOOPER_STATE_ENTERED; indica ao Binder que a thread principal está pronta para receber mensagens
BC_EXIT_LOOPER	Muda o status do binder looper para BINDER_LOOPER_STATE_EXITED; indica ao Binder que essa thread foi encerrada
BC_REQUEST_DEATH_NOTIFICATION	Requisita uma notificação de morte ao processo alvo
BC_CLEAR_DEATH_NOTIFICATION	Requisita que o processo alvo limpe as notificações de morte
BC_DEAD_BINDER_DONE	Informa que a notificação de morte foi processada

O objeto `binder_transaction_data` encapsula a estrutura de dados *parcel*, que consiste em um buffer de dados e um buffer de *offsets*. O buffer de *offsets* indica a posição onde se inicia um novo parâmetro no buffer de dados, enquanto o buffer de dados carrega o tipo do dado passado e suas informações, conforme indica a figura 3.4.

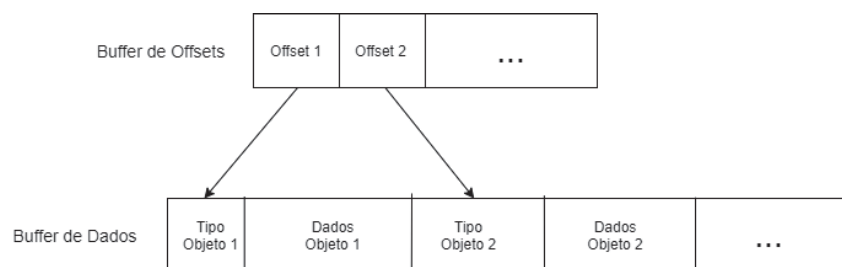


Figura 3.4: Estrutura das *parcel* Binder

3.3 CONSIDERAÇÕES FINAIS

Neste capítulo foi apresentada a arquitetura do sistema operacional Android e suas camadas de abstração, bem como seus principais métodos de segurança: aplicativos executam com privilégio mínimo e sistema de permissões. Além disso, na seção 3.1 o funcionamento e importância do Binder, como módulo de comunicação entre processos do Android, foram apresentados em detalhes.

4 DETECÇÃO DE MALWARES NO AMBIENTE ANDROID

A plataforma de computação móvel Android está crescendo em popularidade nos últimos anos devido à sua natureza *open-source* e grátis. Com um crescimento tão grande no *market share*, o Android se tornou o alvo número um dos criadores de malware (Kumar e Shukla, 2020).

Com esse aumento na quantidade de ataques e malwares para Android, foram aceleradas também as pesquisas de detecção de malware específicas para a plataforma. Apesar de ser baseado em Linux, o Android possui arquitetura e sistemas de segurança específicos e consequentemente necessita de IDSs que considerem essas diferenças.

Além da arquitetura particular do Android, o fato do sistema operacional ser utilizado em ambientes móveis e de Internet das Coisas impõe limites de uso de recursos, como processamento e bateria, ao desenvolvimento de IDSs para a plataforma.

Levando esses fatores em conta, as pesquisas sobre detecção de malware no Android têm buscado diferentes combinações de atributos para otimizar esse processo, que vão desde o código fonte de aplicações até chamadas de sistema e consumo de recursos.

Uma maneira de selecionar os atributos mais relevantes para um sistema de detecção é utilizar métodos de seleção de atributos, que têm como objetivo eliminar os atributos redundantes e irrelevantes para a análise dos dados. Os métodos de seleção de atributos não apenas diminuem a dimensão dos dados, facilitando a sua visualização e entendimento, como também levam a modelos mais compactos e gerais (Solorio-Fernández et al., 2019).

4.1 MALWARE

Malwares se tratam de *softwares* hostís, que normalmente têm como objetivo corromper ou burlar o uso normal de sistemas. Malwares podem ser definidos como qualquer código adicionado, modificado ou removido de um sistema de *software* com objetivo de causar danos ou usar de maneira diferente uma determinada funcionalidade do sistema (Idika e Mathur, 2007).

Devido a essa natureza maliciosa, os malwares são responsáveis por inúmeros prejuízos a instituições e pessoas; esse prejuízo pode ser financeiro, de dados, na imagem das mesmas, etc. Logo, combater a propagação de malwares é uma prioridade atualmente. Os malwares podem ser classificados em diferentes tipos de acordo com a maneira como ele se propaga, explora o sistema ou pelo seu *payload*; isso permite que os malwares sejam divididos nas seguintes categorias: vírus, *worm*, cavalo de Troia, *adware*, *rootkit*, *bots* e *ransomware* (Namanya et al., 2018).

Vírus são malwares que se infiltram em hosts e programas; se tratam de arquivos executáveis, sendo assim são passivos e dependem que o usuário os execute para que possam realizar suas atividades e se replicar. Já os *worms* são programas autônomos e se replicam através da rede de maneira ativa explorando suas vulnerabilidades. Os cavalos de Troia se tratam de malwares disfarçados, pois são *softwares* que possuem uma funcionalidade conhecida pelo usuário e uma maliciosa que é executada junto, sem o conhecimento do mesmo. *Adwares* geram anúncios automáticos expostos diretamente nas aplicações ou através de pop-ups. *Rootkits* são malwares complexos e difíceis de serem removidos, pois eles adulteram o funcionamento do sistema operacional para se ocultar; para isso eles utilizam os privilégios mais altos possíveis, tendo assim controle do sistema operacional para se ocultar e executar suas atividades maliciosas. Os *bots* são programas desenvolvidos para executar tarefas específicas; eles são utilizados para formar *botnets*, que se tratam de redes de hosts controladas por um hacker. Os *ransomwares* são

programas desenvolvidos para criptografar os dados do host atacado e então cobrar um resgate pelos mesmos.

Ao longo do tempo, os malwares tem evoluído para aumentar seus meios de propagação e dificultar a sua detecção, resultando em um aumento no número de malwares e na complexidade dos mesmos. Os malwares modernos são desenvolvidos através da mutação de características, o que gera um crescimento acentuado no número de variantes de malwares existentes. Devido a isso, os malwares são caracterizados com base em suas características maliciosas, que definem assim as famílias de malwares (Ahmadi et al., 2016). Sendo assim, uma família de malwares é um conjunto de malwares desenvolvidos utilizando o mesmo código base.

4.2 FORMAS DE ANÁLISE

Quanto à análise dos dados selecionados, os sistemas de detecção de malware para Android se dividem em três categorias: análise estática, que visa analisar características inerentes à aplicação declaradas em seu código fonte; dinâmica, que visa analisar o comportamento da aplicação em execução; e híbrida, que consiste em combinar a análise estática e dinâmica (Kouliaridis et al., 2020).

4.2.1 Análise estática

A análise estática consiste em identificar vulnerabilidades e malwares em uma aplicação ao se analisar o código fonte da mesma, sem executá-la (Qiu et al., 2019). O arquivo executável de uma aplicação Android utiliza o formato `.apk`, que se trata de um arquivo compactado de recursos, arquivos de configuração e um arquivo `.dex`, que contém o bytecode que será interpretado na máquina virtual ART. Para fazer o *disassembling* do arquivo `.dex` e ter acesso às classes utilizadas na aplicação, existem ferramentas que convertem esse arquivo em um arquivo de classes java; um exemplo de ferramenta que realiza essa tarefa é a `dex2jar`.

As técnicas de análise estática têm como vantagem o fato de serem rápidas e leves, por não exigirem que o aplicativo seja executado (Hamed et al., 2019). Sua desvantagem é a possibilidade dos aplicativos maliciosos esconderem a sua real atividade no arquivo executável através de técnicas de ofuscação como códigos encriptados, geração de código em tempo de execução, etc; esse tipo de comportamento só poderia ser detectado na análise dinâmica em tempo de execução (Nirumand et al., 2018).

Os algoritmos de análise estática podem ser classificados em baseados em assinaturas, baseados em permissões e baseados no bytecode da máquina virtual (arquivos `.dex`). Os algoritmos baseados em assinaturas buscam por padrões semânticos e strings específicas no código fonte; os algoritmos baseados em permissões analisam o arquivo *AndroidManifest.xml* em busca de permissões suspeitas para identificar potenciais malwares; já os baseados nos arquivos `.dex` analisam os fluxos de dados, chamadas de APIs, etc; em busca de comportamentos maliciosos. Além disso, também existem algoritmos que combinam a análise de mais de um desses aspectos da aplicação Android, sendo assim utilizam um algoritmo de análise estática híbrido, como por exemplo o trabalho de (Xu et al., 2018) apresentado em 5.1.

4.2.2 Análise dinâmica

A análise dinâmica consiste em monitorar e analisar o comportamento do aplicativo durante a sua execução para detectar alguma intrusão e consequentemente se ele é malicioso ou não. Existem vários atributos que podem ser utilizados na análise dinâmica, como uso de recursos, comunicação com a Internet, chamadas de sistemas, comunicação entre aplicativos, etc.

Essa técnica resolve a limitação da análise estática quanto à ofuscação de código, uma vez que é analisado o comportamento da aplicação e não o código em si. Outra vantagem da análise dinâmica é que ela gera bases de dados de grande escala para serem analisadas, possibilitando o acesso a mais informações para a classificação da aplicação (Hamed et al., 2019). As principais desvantagens da análise dinâmica são o seu alto custo em recursos, que são limitados em ambientes móvel, e que malwares avançados podem perceber que estão sendo executados em ambiente emulado e esconder seus comportamentos maliciosos (Qiu et al., 2019).

De acordo com a técnica de análise utilizada, esses algoritmos podem ser classificados em: baseados em anomalia, baseados em emulação e *taint analysis* (Hamed et al., 2019). A análise baseada em anomalia consiste em analisar se o comportamento do dispositivo diverge do normal enquanto a aplicação em questão é executada. A técnica de *taint analysis* consiste em marcar cada objeto/dado utilizado pela aplicação para rastrear o fluxo desses objetos através do seu ciclo de vida (Fritz et al., 2013). As técnicas baseadas em emulação consistem em monitorar a execução da aplicação fora do ambiente de execução, através do uso de *sandboxes*, mitigando assim o risco de um malware danificar o dispositivo antes de ser identificado.

4.2.3 Análise híbrida

A análise híbrida consiste na utilização de dados provenientes de fontes estáticas e dinâmicas e tem como objetivo mitigar as desvantagens de ambos os métodos. Apesar de resolver o problema de desempenho na detecção de aplicações maliciosas, os modelos de análise híbrida ainda são ineficientes (Arshad et al., 2018). As principais ineficiências apontadas por (Arshad et al., 2018) estão ligadas ao consumo de memória e queda de desempenho do dispositivo durante a análise.

Quanto ao processamento dos dados, os modelos de análise híbrida podem utilizar as características estáticas e dinâmicas coletadas simultaneamente ou sequencialmente. Normalmente, ao se utilizar os dados simultaneamente, a análise é feita através de algoritmos de aprendizado de máquina por ser flexível à combinação de dados de diferentes fontes.

4.3 CONSIDERAÇÕES FINAIS

Neste capítulo foram abordados os conceitos de detecção de malware no Android. Para isso, primeiro foram abordados os conceitos de malware e família de malware e posteriormente foram apresentados os tipos de métodos de detecção de Malware para Android e suas diferenças a respeito de limitações e custos computacionais.

5 TRABALHOS RELACIONADOS

Neste capítulo serão apresentados os mecanismos de detecção de malware no Android presentes na literatura atual. Para isso serão apresentados os mecanismos mais relevantes propostos a partir de 2018 de acordo com os seus tipos de abordagem.

(Zachariah et al., 2017) comparou os sistemas de detecção de malware propostos até 2018 através de uma tabela que apresenta o método utilizado e suas vantagens, limitações e desempenho. Sua análise concluiu que os modelos de abordagem estática, apesar de menos eficientes na detecção, eram mais rápidos e efetivos contra ataques conhecidos; enquanto os de abordagem dinâmica eram em sua maioria baseados somente em chamadas de sistema e possuíam grandes limitações devido à falta de recursos no ambiente móvel como: drenagem de bateria, demora no processamento, perda de eficiência se muitas chamadas de sistema forem executadas, etc. O autor chamou atenção ainda para as abordagens híbridas que apresentaram desempenho alto sem tantas limitações decorrentes do *hardware*.

5.1 MODELOS DE ANÁLISE ESTÁTICA

O modelo proposto por (Shahzad, 2018) foca em validar uma abordagem capaz de detectar ataques de dia zero através do uso de aplicações sintéticas. Para isso ele utilizou dados do arquivo `AndroidManifest.xml` e do código SMALI¹. Duas abordagens foram avaliadas: a primeira para avaliar o impacto da fusão de atributos e algoritmos e a segunda para avaliar o impacto de dados gerados artificialmente. A primeira abordagem foi conduzida utilizando apenas os dados dos aplicativos coletados com uma base balanceada, já na segunda abordagem foi utilizada uma base desbalanceada e adicionados à mesma aplicativos sintéticos gerados através de SMOTE (*Synthetic Minority Oversampling Technique*). Os resultados mostram que as abordagens são capazes de identificar malwares com uma baixa taxa de falsos positivos, entretanto não foi identificado ganho de desempenho com o uso dos exemplos sintéticos, indicando que o seu método de criação (média dos 5 exemplos mais próximos) pode não ter sido efetivo em melhorar a detecção de ataques desconhecidos, apesar de evitar a necessidade de uma base de dados balanceada. Além disso, as abordagens propostas são custosas para o dispositivo, pois utilizam seleção e fusão de atributos e executar cinco algoritmos de aprendizagem de máquina.

A abordagem de (Xu et al., 2018) propõe uma análise dos aplicativos em duas camadas, onde a segunda camada é executada caso a primeira tenha um resultado classificado como incerto. A primeira camada utiliza uma rede neural (*Multilayer Perceptron*) para avaliar os arquivos XML do aplicativo, onde a maioria dos malwares são identificados. A segunda camada utiliza LSTM (*Long Short-Term Memory*), que se trata de uma arquitetura de rede neural recorrente (RNN) que lembra valores em intervalos arbitrários, baseado na semântica do *bytecode* da aplicação. Seu resultado apresentou uma alta taxa de detecção de malwares com um número de falsos positivos relativamente baixo. Apesar de ter se mostrado mais eficiente contra ofuscação de código que outros modelos baseados em análise estática, essa abordagem ainda está sujeita a limitações como: não identificar código carregado durante a execução do aplicativo; além disso, o uso de *deep learning* em ambas as camadas torna o modelo custoso de ser treinado em comparação a modelos que utilizam algoritmos de aprendizado de máquina tradicionais.

¹O código SMALI é uma forma mais legível para o `.dex`, que contém o *Dalvik bytecode*; no Android ele é equivalente ao Assembly de outros SO.

VanDroid (Nirumand et al., 2018) foi proposto para detectar vulnerabilidades em aplicativos Android usando uma técnica de engenharia reversa orientada por modelos. Nessa técnica, os aplicativos Android são descompilados e seus componentes estáticos são transformados em um único modelo, enquanto cada vulnerabilidade analisada também possui um modelo. Para avaliar um aplicativo para cada vulnerabilidade, o modelo do aplicativo é transformado para o modelo da vulnerabilidade. O método foi validado com sucesso usando os modelos *Intent Spoofing* e *Unauthorized Intent Receipt*, que são duas vulnerabilidades relacionadas à comunicação entre processos. Entretanto, a metodologia proposta tem algumas desvantagens, como ser estática e, portanto, vulnerável à ofuscação de código e, principalmente, precisar de um modelo diferente para identificar cada vulnerabilidade, impossibilitando acompanhar a evolução e variedade de *malwares* para Android atualmente.

(Shrivastava e Kumar, 2019) propôs uma abordagem estática para detectar a confiabilidade de permissões e intents como atributos na detecção de malware. Em sua metodologia, foram contadas as frequências de cada intent e permissão nas amostras analisadas e, em seguida, usado um modelo matemático para pontuar cada amostra, com base em sua frequência de aparecimento. Com essa abordagem, pretende-se prever o risco do aplicativo, entretanto esse método é muito vulnerável à ofuscação de código, uma vez que um invasor poderia adicionar permissões não arriscadas em seu aplicativo para diminuir sua pontuação. Outro problema da abordagem é que seria muito desafiador manter a tabela de pontuação atualizada com o enorme crescimento e variedade de aplicativos e serviços Android.

DroidEvolver (Xu et al., 2019) tem o objetivo de resolver a necessidade de envolvimento humano constante para que os mesmos continuem robustos através do tempo. Para isso, ele propôs um modelo onde os aplicativos analisados são utilizados como entrada para atualizar o *pool* de modelos de detecção e os atributos (chamadas de API). Dessa forma, o modelo se mantém atualizado com novas APIs utilizadas e novos ataques. Para validar o modelo ele foi treinado inicialmente com aplicativos antigos e então foi sendo testado com aplicativos dos anos seguintes. Como resultado, o algoritmo teve uma queda em seu desempenho ao longo dos anos significativamente menor que o modelo utilizado para comparação (Onwuzurike et al., 2019), além de ser um modelo mais leve. A limitação desse modelo é que os falsos negativos e positivos também são utilizados para atualizar os modelos de detecção, podendo assim propagar erros, sendo assim vulnerável a um envenenamento de sua base; além disso, o modelo é vulnerável a todo tipo de código carregado durante a execução e como se baseia apenas em chamadas de API ele não pode detectar malwares que dependam de outros componentes ou análises mais complexas.

Para detectar ataques de escalada de privilégios, que estão principalmente relacionados a vulnerabilidades no ICC (*Inter-Component Communication*), (Sharmeen et al., 2020) propôs dois modelos de detecção: um baseado em aprendizagem semi-supervisionada e outro em *deep learning*. O objetivo principal dos modelos é continuar aprendendo com os aplicativos desconhecidos de modo a se manter atualizado e capaz de detectar novos tipos de malware. Em ambos os modelos foram utilizadas chamadas a APIs e permissões declaradas como dados de entrada. Ambos os modelos alcançaram um alto desempenho em detectar malwares, inclusive os de dia zero, embora o modelo baseado em *deep learning* tenha alcançado uma precisão mais alta. A principal limitação desse modelo é o fato dele ser muito pesado, sendo utilizadas inclusive GPUs para executar os mesmos; apesar de pesado, o modelo se baseia em atributos estáticos simples e, portanto, não é capaz de detectar todo tipo de malware e é vulnerável a ofuscação de código. Outra limitação é dada pelo fato dele ser vulnerável ao envenenamento de sua base, uma vez que utiliza aplicativos desconhecidos para atualizar seus parâmetros.

NSDroid (Liu et al., 2020) tem como objetivo diminuir o tempo de processamento dos classificadores de malware no Android baseados em grafos de chamadas de função através do uso da similaridade dos vizinhos no grafo de chamadas de funções locais em detrimento de isomorfismo dos subgrafos, que é tradicionalmente utilizado. Para construir o grafo foi utilizado o software de engenharia reversa Androguard (Desnos et al., 2013) e o vetor de assinatura gerado a partir do grafo é processado em um classificador SVM. Com isso o NSDroid é capaz de identificar a família que cada malware pertence com uma redução da latência de detecção, em relação ao seu *benchmark* (Fan et al., 2016), de até vinte vezes. Apesar dos resultados promissores, sua limitação é dada pela natureza estática do Androguard, utilizado para coletar os grafos, que não possibilita prever o comportamento de chamada das funções durante a execução; sendo assim vulnerável, por exemplo, a ofuscação de código e código carregado dinamicamente.

Ofuscação de código é uma deficiência de IDSs de análise estática no Android. Para lidar com esse problema DANDroid (Millar et al., 2020) propôs um IDS de análise estática resiliente contra quatro tipos de ofuscação de código: strings criptografadas, recursos criptografados, APIs ofuscadas e classes criptografadas. O modelo proposto utiliza como atributos chamadas de APIs, permissões e o código SMALI do aplicativo e os analisa através de redes neurais adversárias discriminativas, que consiste em modificar uma rede adversária geradora ao substituir a rede neural geradora por outra discriminadora de forma que uma aprenda a detectar malwares, enquanto a outra detecte se o código foi ofuscado. Nos testes, a arquitetura utilizando as redes adversárias discriminadoras com funções diferentes de avaliação obtiveram um melhor desempenho que outras arquiteturas de redes neurais; o modelo obteve um resultado positivo, apesar de ter aplicativos ofuscados em sua base de testes. Apesar dos resultados promissores contra código ofuscado, o fato de todos os aplicativos terem sido ofuscados utilizando a mesma ferramenta e com os mesmos parâmetros deixa dúvidas sobre o desempenho do modelo com aplicativos desconhecidos e ofuscados utilizando outras ferramentas. Além disso, os parâmetros escolhidos não são afetados por strings e recursos criptografados, o que pode ser notado no alto desempenho com os aplicativos utilizando esse tipo de ofuscação em todos os modelos testados, inclusive no treinado apenas com aplicativos sem ofuscação e sem utilizar a rede adversária.

5.2 MODELOS DE ANÁLISE DINÂMICA

DroidCat (Cai et al., 2019) foi desenvolvido com o objetivo de encontrar um método resiliente de detecção dinâmica de malware que não dependa de chamadas de sistema. Para isso, *DroidCat* usou um conjunto de atributos composto de chamadas de métodos comuns e *Intents*, que se trata de uma das implementações na camada de aplicação do Android que permite a comunicação entre processos. Analisando-se esses atributos, foram observadas diferenças no uso de *Intents* e APIs da SDK por aplicações maliciosas e benignas: aplicações maliciosas utilizam *Intents* explícitas e chamam APIs da SDK de bibliotecas externas ou código do usuário com mais frequência que aplicações benignas. Através dessa técnica, *DroidCat* atingiu uma acurácia elevada na detecção de *malware*, entretanto para coletar esses dados foi utilizada instrumentação dos aplicativos e logo foram necessárias mudanças nos aplicativos analisados; além disso, em relação à comunicação entre processos, apenas as *Intents* são monitoradas e logo, caso o atacante utilize outro mecanismo de comunicação em vez de *Intents*, a metodologia adotada pelo *DroidCat's* não estará monitorando essa ação maliciosa.

DL-Droid (Alzaylaee et al., 2020) foi proposto como um sistema de aprendizado profundo para detectar aplicativos maliciosos por meio de análise dinâmica usando geração de entrada *stateful*, o que significa que o estado atual do aplicativo é levado em consideração ao decidir a próxima entrada. Neste trabalho, recursos dinâmicos (ou seja, chamadas de API e

intents - ações / eventos) foram coletados usando a metodologia *stateless* e *stateful* e ambos os resultados obtidos foram comparados; também foram feitos testes com um modelo híbrido ao se adicionar as permissões, entretanto esse experimento não será considerado nessa revisão por não ter mais contribuições ao trabalho. Embora a abordagem *stateful* melhore a cobertura de código, os resultados da detecção foram semelhantes para ambas as abordagens para todos os algoritmos de aprendizado de máquina e a rede neural profunda testada; a diferença nos resultados das abordagens foi de no máximo 0,03 usando a métrica w-FM (*weighted-F Measure*), que se trata de uma média ponderada do *f1score* para as duas classes da base de dados. Embora (Alzaylaee et al., 2020) tenha obtido resultados muito bons com validação cruzada de 10 vezes, a metodologia adotada não separou um conjunto de dados de teste do de validação e, portanto, os resultados são afetados por vieses das etapas de preparação de dados, como seleção de atributos, por exemplo.

Uma abordagem baseada em anomalia e baseada no *host* foi proposta por (Ribeiro et al., 2019). O modelo proposto monitora quinze atributos do dispositivo em tempo real; alguns exemplos de atributos analisados são: uso de CPU, quantidade de aplicativos instalados no *host*, tráfego de rede, bateria, temperatura, status da tela (acesa/apagada), entre outros. Por ser um modelo baseado em anomalia, o modelo tem como vantagem não precisar de aplicações maliciosas para o seu treinamento. O modelo foi testado com 9 algoritmos de aprendizagem de máquina diferentes e obteve como resultado um bom desempenho na detecção de malware. Apesar de ter obtido um bom resultado, a proposta não aborda a problemática de como identificar qual aplicativo é malicioso com mais de um aplicativo sendo executado simultaneamente, uma vez que é proposto um modelo online e baseado no *host* e todos os atributos utilizados são relativos ao dispositivo como um todo e não aos aplicativos; também não foi abordado o possível alto consumo de recursos ao se monitorar o dispositivo constantemente. Além disso, o modelo foi testado apenas contra 5 tipos de malware, sendo o alto consumo de CPU uma característica comum a todos eles, deixando dúvidas sobre a capacidade do modelo de se adaptar para detectar outros tipos de malware.

DroidLight (Barbhuiya et al., 2020) também propôs uma abordagem baseada em anomalia. Sua abordagem tem como objetivos ser capaz de detectar novos ataques em tempo real sem adicionar um *overhead* grande ao uso de recursos do dispositivo. O modelo coleta dados do uso de CPU e tráfego de rede gerados por cada aplicativo e gera remotamente uma assinatura do comportamento normal do aplicativo; caso o comportamento do aplicativo se altere significativamente um possível ataque é identificado. O modelo foi testado e demonstrou elevada precisão com malwares desenvolvidos pelo autor relacionados a roubo de informações, bot de mineração de criptomoedas e ataques DDoS; apesar dos resultados promissores, não foram realizados testes com um conjunto de dados maior para validar o modelo. Como todo o processamento dos dados para gerar as assinaturas de comportamento dos aplicativos é feito remotamente, nos testes realizados o IDS adicionou um *overhead* de apenas 1,5% nos recursos do dispositivo; entretanto o dispositivo depende do funcionamento da rede para identificar os comportamentos anômalos remotamente.

5.3 MODELOS DE ANÁLISE HÍBRIDA

SAMADroid (Arshad et al., 2018) se trata de um detector de análise híbrida proposto com a intenção de apresentar um bom desempenho na precisão e nos consumos de bateria e de armazenamento. Para tal, o modelo proposto se estrutura em três níveis: coleta de atributos estáticos e dinâmicos, comunicação entre *hosts* local e remoto e algoritmo de aprendizado de máquina. Como atributos dinâmicos são utilizados os *logs* de chamadas de sistema gerados no *host* local e enviados para o servidor remoto e como atributos estáticos são usados dados do

AndroidManifest.xml e do código SMALI² coletados no servidor remoto onde o aplicativo é descompilado. O servidor remoto analisa em tempo real os *logs* de chamadas de sistema enviados pelo *host* local juntamente com os atributos estáticos, através de algoritmos de aprendizado de máquina. Essa abordagem de análise remota permite ao *SAMADroid* analisar em tempo real com um *overhead* mínimo nos recursos de bateria e memória, embora tenha como limitação a necessidade de comunicação constante entre o *host* local e o servidor remoto para funcionar. Como resultado, o modelo alcançou uma alta precisão na detecção com um baixo índice de falsos positivos na base de dados Drebin (Arp et al., 2014), utilizada para os testes, que não contém aplicações atuais.

DroidFusion (Yerima e Sezer, 2019) propôs uma arquitetura multinível para combinar diferentes algoritmos de classificação que utilizam aprendizado de máquina. O *framework* foi proposto para combinar os resultados de classificadores genéricos e logo é independente dos atributos utilizados pelos mesmos, embora tenha sido testado apenas para algoritmos estáticos durante esse estudo. A fusão dos algoritmos é implementada através de um algoritmo que se baseia na precisão de cada algoritmo do nível abaixo para ranqueá-los e então gerar a classificação definitiva com base em suas saídas. Em comparação com os *frameworks* que utilizam uma arquitetura em pilha para combinar algoritmos, o *DroidFusion* se mostrou mais preciso em todos os testes realizados; além disso, nos testes realizados, o *framework* precisou em média de sete segundos para analisar um aplicativo e portanto indica possuir um desempenho adequado para ser utilizado em produção, entretanto a complexidade do mesmo cresce de acordo com os classificadores utilizados e os atributos dos mesmos.

5.4 CONSIDERAÇÕES SOBRE OS MODELOS

Foram apresentadas diferentes abordagens estado da arte para detecção de ataques em dispositivos Android; dessa análise é possível observar as seguintes tendências no assunto:

- Os IDSs em geral têm diminuído a necessidade de novas amostras de treinamento, uma vez que os malwares evoluem muito rápido e seria complicado manter o modelo atualizado com *inputs* manuais;
- Grande parte dos novos modelos baseados em análise estática têm buscado mitigar as suas limitações contra técnicas de evasão, uma vez que os modelos existentes já são suficientemente eficientes contra malwares que não as utilizam e consomem poucos recursos do dispositivo;
- Os modelos que utilizam análise dinâmica têm buscado reduzir o *overhead* de recursos da máquina, que atualmente é o maior limitador para utilizar esses modelos em produção;
- Os modelos baseados em análise dinâmica e híbrida têm buscado se especializar em detectar certos tipos de ataques onde existem limitações através da análise estática, como por exemplo ataques através da rede.

Os modelos analisados foram sumarizados para comparação na tabela 5.1. Vale ressaltar que as taxas de precisão não foram abordadas na tabela para fins de comparação, pois os modelos utilizam bases de dados e técnicas de testes distintas e logo as métricas de precisão utilizadas, embora representem a eficiência dos modelos, não podem ser comparadas diretamente para se determinar o mais preciso corretamente.

²O formato APK com o arquivo *.dex* é o formato binário de uma aplicação, que não é de fácil entendimento nem prático para efetuar modificações. O formato SMALI vem para permitir o fácil entendimento destas informações.

Em relação às abordagens dinâmicas observou-se uma expansão com relação aos atributos utilizados. Diferentemente dos modelos analisados por (Zachariah et al., 2017), onde utilizava-se majoritariamente chamadas de sistema como atributo, os modelos estado da arte recentes têm utilizado atributos de camadas acima do kernel Linux na arquitetura Android. Essa evolução é importante para os IDSs passarem a levar em consideração fatores específicos do sistema operacional Android na detecção em tempo real, tornando-se assim especializados e otimizados para a arquitetura do mesmo. Vale ressaltar, que apesar da diversificação nos atributos dinâmicos abordados, não foi encontrado na atual literatura outro trabalho que utilize o Binder como fonte de dados para representar o comportamento dos aplicativos ao utilizarem o IPC e consequentemente identificar aplicativos maliciosos.

5.5 CONSIDERAÇÕES FINAIS

Este capítulo apresentou os modelos de IDS para Android que representam o estado da arte no assunto, separando-os entre modelos de análise dinâmica, estática e híbrida e ressaltando os seus funcionamentos, vantagens e limitações. Após apresentar os modelos foi feita uma análise sobre as tendências de pesquisa no assunto.

Tabela 5.1: Resumo dos trabalhos relacionados

Artigo	Fonte de dados	online/offline	Local de processamento	Dados	Técnica de análise
(Shahzad, 2018)	Estática	offline	Não definido	Permissões, Intents e instruções SMALI	Fusão de atributos + Ensemble de 5 algoritmos de ML
(Xu et al., 2018)	Estática	offline	Não definido	Arquivos XML + dex bytecode	MLP + LSTM
(Nirumand et al., 2018)	Estática	offline	Não definido	Filtro de Intents, Intents, componentes, aplicação, permissões, grupos de permissões, SDK	Adequação a modelos de vulnerabilidades
(Shrivastava e Kumar, 2019)	Estática	offline	Não definido	Permissões e Intents	Modelagem matemática para estimar risco da aplicação
(Sharmeen et al., 2020)	Estática	offline	Não definido ³	Permissões e chamadas de API	Aprendizagem profunda ou aprendizado semi-supervisionado (Random Forest, ExtraTree, Decision tree, SVM ou XRGB)
(Xu et al., 2019)	Estática	online	Não definido	chamadas de API	pool de algoritmos lineares (Passive Aggressive (PA), Online Gradient Descent (OGD), Adaptive Regularization of Weight Vectors (AROW), Regularized Dual Averaging (RDA) e Adaptive Forward-Backward Splitting (Ada-FOBOS))
(Liu et al., 2020)	Estática	offline	Não definido	Grafo de chamadas de função	SVM
(Millar et al., 2020)	Estática	offline	Não definido	Chamadas de APIs, permissões e o código SMALI	Redes Neurais Adversárias Discriminativas
(Cai et al., 2019)	Dinâmica	online	Não definido	Chamadas de métodos e Intents	Random Forest
(Alzaylaee et al., 2020)	Dinâmica	online	Remoto (WEKA e H2O flow)	Chamadas de APIs e Intents	Aprendizagem Profunda

³ Apesar de o artigo não definir explicitamente, o fato de o modelo ser muito custoso e exigir o uso intenso de GPUs indica que o processamento deve ser realizado remotamente.

Artigo	Fonte de dados	online/offline	Local de processamento	Dados	Técnica de análise
(Ribeiro et al., 2019)	Dinâmica	online	Local	uso de CPU, memória e cache; taxa de pacotes e bits enviados/recebidos; número de aplicativos instalados; uso de bateria; temperatura; número de SMSs enviados, SMSs recebidos para números desconhecidos e ligações feitas; status da tela; número de sockets TCP abertos e de processos/serviços sendo executados	One Rule, Decision Tree, Naive Bayes, Bayesian Network, Logistic Regression, Random Forest, KNN, SVM ou K-means
(Barbhuiya et al., 2020)	Dinâmica	online	Remoto	Uso de CPU e tráfego de rede	modelo OCC
(Arshad et al., 2018)	Híbrida	online	Remoto	Chamadas de sistema, AndroidManifest.xml e código SMALI	<i>Linear SVM</i>
(Yerima e Sezer, 2019)	Híbrida	offline	Não definido	Genérico	Fusão de classificadores

6 PROPOSTA DA PESQUISA

Este trabalho propõe a utilização de informações dinâmicas referentes a comunicação entre processos no Android como fonte de dados para a detecção de ações maliciosas. Conforme visto na seção 4, existem trabalhos que se utilizam de informações estáticas referentes a essa comunicação, como por exemplo a declaração de *intents*; entretanto, não foram encontrados na literatura o uso de dados dinâmicos do Binder para detecção de ataques, se tratando assim de uma proposta inédita.

Conforme apresentado na seção 3.1, toda a comunicação entre processos no Android é intermediada pelo módulo do Kernel Binder. Dessa forma, o Binder se apresenta como o ponto ideal para se coletar esses dados por concentrar o tráfego oriundo de todas as aplicações do Android, permitindo assim que os dados sejam coletados sem alterar as aplicações analisadas ou o protocolo de comunicação do Android. No modelo proposto, essas informações serão coletadas no formato de traços de chamadas de função do Binder. Um traço de chamadas de função consiste no registro de informações referentes a todas as funções executadas no momento de sua entrada (Bird, 2009).

6.1 JUSTIFICATIVA

Conforme apresentado no capítulo 4, o desenvolvimento de IDSs para o ambiente Android é um tema com grande relevância nos últimos anos, com muitas pesquisas sendo realizadas na área. Por outro lado, os malwares para Android também têm evoluído para evitar esses mecanismos de detecção. O uso combinado de técnicas de ofuscação de código, por exemplo, têm um grande impacto na eficiência desses mecanismos, diminuindo as suas taxas de detecção em mais de 50% na média para mecanismos de análise estática (Ajiri et al., 2020).

Os métodos de detecção utilizando análise dinâmica não apresentam esse tipo de vulnerabilidade, sendo assim mais difíceis de serem burlados. Nesse sentido, foi observado na seção 5, ao se analisar os modelos estado da arte, que os modelos de análise dinâmica têm se tornado cada vez mais específicos para o ambiente Android, utilizando outras características além das chamadas de sistema do kernel Linux.

Embora essa evolução nos métodos de detecção dinâmica esteja acontecendo, foi observada a ausência de pesquisas levando em consideração as informações do mecanismo de comunicação Binder como atributo para detecção de malware. Esse módulo do kernel Android parece ser uma fonte de informações relevante para detecção de intrusões devido aos seguintes motivos:

- Toda a comunicação entre processos do Android passa por ele, conforme mostrado na seção 3.1; alguns exemplos de mensagens relevantes que passam pelo Binder são:
 - Requisições a internet;
 - Acesso a arquivos de mídia;
 - Acesso a recursos de hardware como câmera e gravador de voz;
 - Requisições a outros aplicativos no Android.
- Diversos modelos de análise estática utilizam com sucesso características relacionadas ao funcionamento do Binder, como declarações de Intents e permissões;

- Permite acesso a dados de todos aplicativos no mesmo local sem a necessidade de alterações nos aplicativos ou no Android;
- Rastrear e entender as comunicações que utilizam esse canal é um ponto chave para entender comportamentos específicos do Android (Tam et al., 2015)
- Ele é um ponto crítico para ataques, conforme descrito por (Artenstein e Revivo, 2014) e (Shen et al., 2017), por ter permissão de acesso a todos os recursos do Android.

A importância das informações do Binder para se compreender comportamentos do Android foi descrita por (Tam et al., 2015) em seu trabalho, onde o comportamento do Binder foi reconstruído através do dissecamento dos parâmetros das chamadas de sistema `ioctl` realizadas pelo módulo Binder. Essa abordagem mostrou que as informações referentes ao comportamento do Binder melhoraram a performance na detecção de malware, entretanto o método de coleta dessas informações gerou um *overhead* considerável no sistema operacional ao se executar tarefas através do Binder.

Dessa forma, este trabalho pretende investigar a validade da comunicação entre processos como fonte de dados para detecção de intrusões no Android; os dados são coletados ao se monitorar as chamadas de função do Binder e coletar o traço das mesmas. As chamadas de função do Binder podem ser monitoradas diretamente, não introduzindo assim muito *overhead* na coleta dos dados, e representam o funcionamento do Binder, pois contém cada uma das ações realizadas pelo mesmo.

6.2 OBJETIVOS

O objetivo principal deste trabalho consiste em avaliar a eficácia e eficiência do uso da comunicação entre processos, realizada através do Binder, como fonte de dados para a detecção de intrusões. Para atingir este objetivo, alguns objetivos intermediários foram definidos:

1. Para avaliar a eficiência desses dados em IDSs é necessário uma base de dados com essas informações para serem analisadas. Como essa base de dados não foi encontrada na literatura, o primeiro objetivo consiste em coletar os traços de execução do Binder;
2. Filtrar as informações desejadas nos traços coletados e representá-los de forma a serem consumidos por um algoritmo de aprendizado de máquina;
3. Definir o *pipeline* de execução do modelo de aprendizado de máquina;
4. Executar o *pipeline* definido;
5. Finalmente, o último passo consiste em avaliar os resultados obtidos de forma a validar a proposta.

6.3 CONTEXTO DE APLICAÇÃO

O contexto de aplicação interfere diretamente em decisões e premissas utilizadas no projeto da aplicação. Isso ocorre pois de acordo com o contexto definido é possível modelar o comportamento esperado da aplicação adequadamente.

Esse trabalho tem escopo limitado à análise de ações maliciosas oriundas dos aplicativos analisados individualmente. Dessa forma, estão fora do escopo deste trabalho a análise de

comportamentos resultantes de ações externas ao aplicativo analisado durante a sua execução ou ataques em conluio que envolvem ações de múltiplos aplicativos para ocorrerem.

Além disso, o IDS proposto realizará a análise dos dados de maneira *offline*, sendo indicado então para analisar aplicativos antes de serem instalados ou antes de serem disponibilizados nas lojas virtuais. Isso é necessário no modelo proposto, pois os traços gerados por execução dos aplicativos são arquivos grandes e logo manter e atualizar em tempo real traços de múltiplos aplicativos sendo utilizados pelo usuário, além de analisá-los, geraria *overhead* significativo no uso de recursos do Android.

O fato da detecção não ser *online* permite que o IDS proposto não possua grandes limitações de uso de recursos e tempo para a análise dos aplicativos; além de diminuir a complexidade de coleta dos traços, uma vez que os aplicativos analisados podem ser executados de um em um.

6.4 CONSIDERAÇÕES FINAIS

Neste capítulo foi apresentada a proposta desse trabalho, bem como os objetivos parciais necessários para cumprir a mesma. Para isso, o capítulo também conta com as justificativas sobre a relevância e viabilidade da proposta e a definição de seu escopo.

7 OBTENÇÃO E REPRESENTAÇÃO DA BASE DE DADOS

Os dados utilizados em um modelo de aprendizado de máquina são determinantes para o seu desempenho, uma vez que são responsáveis por toda informação utilizada pelo mesmo em suas classificações. Para garantir a qualidade dos mesmos e o melhor entendimento de possíveis limitações durante a classificação, o processo de criação da base de dados exige uma metodologia bem definida. Sendo assim, serão detalhados os processos utilizados para aquisição, limpeza e representação dos dados utilizados.

7.1 AQUISIÇÃO DA BASE DE DADOS

Conforme descrito na seção 6.2, não existe na literatura uma base de dados com traços de chamadas de função do Binder, sendo necessário a coleta dos traços para a construção da base de dados AndroBind. Para isso, se faz necessário uma lista de aplicativos rotulados de forma a gerar uma base de traços com seus respectivos rótulos.

A lista de aplicativos rotulados usados como base para gerar a base de dados foi retirada da base de dados Androzoo (Allix et al., 2016). O Androzoo foi escolhido por se tratar de uma base de dados pública e que se mantém atualizada constantemente, permitindo assim experimentos reproduzíveis e que refletem a realidade atual dos aplicativos Android.

Foram selecionados dentro da base de dados do Androzoo aplicativos disponibilizados na loja de aplicativos do Google (*Google Play*) e que foram lançados a partir de 2018. Foram escolhidos os aplicativos a partir de 2018, pois em 2017 houve uma atualização no Binder implementada a partir do Android Oreo, que permitiu passar buffers de memória brutos em uma transação sem serializá-los em `parcel` através dos comandos `BC_TRANSACTION_SG` e `BC_REPLY_SG` (Stultz, 2021). Além disso, o filtro por loja de aplicativos foi utilizado porque a *Google Play* é a loja oficial do Android e a loja de origem de aproximadamente 83% dos aplicativos presentes no Androzoo.

7.1.1 Base de Dados Androzoo

O Androzoo é uma base de dados pública de aplicativos Android constantemente atualizada; atualmente o Androzoo conta com 14,824,490 aplicativos e todos são analisados por dezenas de antivírus para identificar quais são malwares ou não. Os aplicativos são provenientes de 18 lojas de aplicativos diferentes, entretanto a maior parte veio da loja oficial do Google.

A base de dados conta com os aplicativos disponíveis para *download* diretamente da API do Androzoo, um arquivo CSV contendo uma lista de todos os aplicativos e seus metadados e as informações extraídas da análise dos antivírus disponibilizadas em arquivos JSON.

Os arquivos JSON contém informações sobre o aplicativo ter sido identificado como malware, a família e o tipo de malware que o mesmo pertence; entretanto as informações sobre família e tipo não estão disponíveis para todos os aplicativos identificados como maliciosos. Os metadados presentes na planilha incluem os seguintes dados:

- sha256, sha1 e md5: hashes dos aplicativos, que garantem a autenticidade dos mesmos;
- apk_size e dex_size: tamanho do aplicativo (arquivo .apk) e do arquivo classes.dex respectivamente;
- dex_date: data associada ao arquivo .dex;

- `pkg_name`: nome do pacote Android;
- `vercode`: código de versão do pacote Android;
- `vt_detection`, `vt_scan_date`: número de antivírus que detectaram o aplicativo como malware e data em que foi analisado;
- `markets`: lista de lojas virtuais que disponibilizam o aplicativo.

7.1.2 Geração da Base de Dados

A base de dados a ser construída (AndroBind) consiste em um conjunto de traços das chamadas de função no Binder, onde cada traço representa o comportamento de um aplicativo rotulado diferente. O AndroBind utiliza um rótulo binário, onde o aplicativo pode ser malicioso ou não; não foram utilizados os rótulos por família e tipo, pois existem muitos malwares do Androzoo que não possuem essa classificação.

Os aplicativos normais e maliciosos utilizados para gerar a base de dados foram selecionados aleatoriamente no Androzoo e baixados utilizando a API do Androzoo respeitando os critérios definidos: provenientes da loja do Google e com `dex_date` a partir de 2018. Foram considerados como malware aplicativos com `vt_detection` maior que cinco e normais os aplicativos com esse atributo igual a zero.

Os aplicativos selecionados foram instalados e executados em um emulador Android enquanto são monitorados para capturar o traço das chamadas de função do Binder através da ferramenta `ftrace` (Kukunas, 2015); para evitar ruídos provenientes de outros aplicativos ou do próprio sistema do Android o `ftrace` foi configurado para monitorar apenas as tarefas provenientes do `pid` do aplicativo analisado e seus processos filhos através da opção *event-fork*. Para estimular o funcionamento dos aplicativos analisados enquanto são monitorados, são injetados 1000 eventos nos mesmos por meio da ferramenta Monkey (Google, 2019a) como forma de simular a interação do usuário. Após o traço ser gerado, ele é extraído para o computador, o aplicativo analisado é desinstalado e o processo é repetido para o próximo aplicativo.

Esse processo foi automatizado através de um script utilizado para gerar a base de traços. A sequência de passos do script está representada na imagem 7.1. Essa sequência consiste em:

1. Instalar e abrir um aplicativo no emulador;
2. Configurar o `ftrace` para realizar traços referentes a chamadas do Binder realizadas a partir desse aplicativo e ativá-lo;
3. Executar 1000 iterações com o aplicativo através do simulador *monkey*;
4. Desativar o `ftrace` e desinstalar o aplicativo após fechá-lo;
5. Extrair o arquivo do traço para o computador e repetir o processo para o próximo aplicativo;

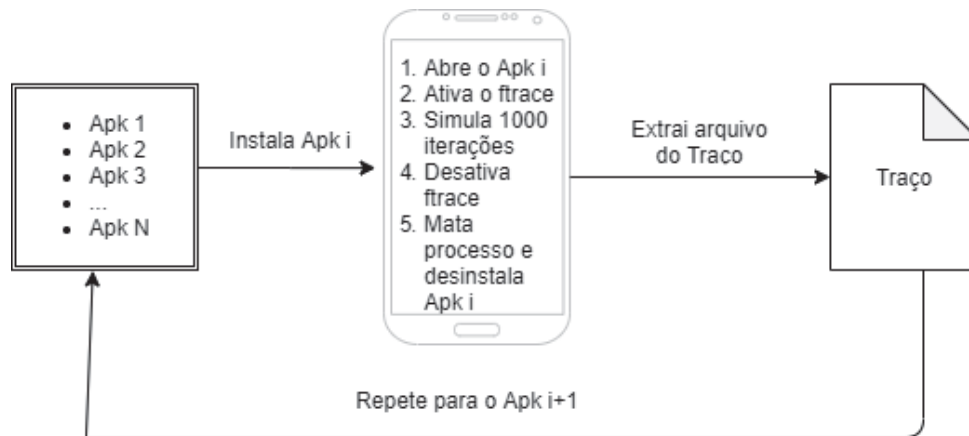


Figura 7.1: Sequência para gerar base de traços

Essa abordagem permite que os aplicativos sejam estimulados de forma equivalente, pois todos foram executados no mesmo ambiente, com a mesma quantidade de iterações e de forma isolada. Além disso, o uso do ftrace permite a coleta direta dos traços de execução do Binder.

7.2 TÉCNICAS DE REPRESENTAÇÃO DE DADOS

A maioria dos métodos de aprendizado operam com vetores numéricos representando os dados não tratados; dessa forma, se faz necessário construir um mapeamento para um espaço vetorial que faça interface com os métodos de aprendizado (Wressnegger et al., 2013).

No caso em que são utilizados dados textuais, a forma mais simples de representação é a *bag-of-words*, que tem como base a contagem de palavras (Zheng e Casari, 2018). *Bag-of-n-Grams* é uma variação do *bag-of-words*, onde são consideradas sequências de palavras. Já o *Tf-Idf* (*term-frequency times inverse document-frequency*) é uma transformação do *bag-of-words* que considera a frequência de palavras na lista de documentos analisada.

7.2.1 Bag-of-Words

A abordagem mais simples para representar dados textuais é o modelo *bag of words*, onde os dados são representados por um vetor que contém a quantidade de vezes que cada uma das palavras ocorre (Boulis e Ostendorf, 2005).

O vetor que representa os dados consiste de um dicionário, que possui como chaves cada uma das palavras possíveis no dicionário e o valor associado às mesmas representa a quantidade de vezes que as mesmas ocorrem, conforme apresentado na figura 7.2.

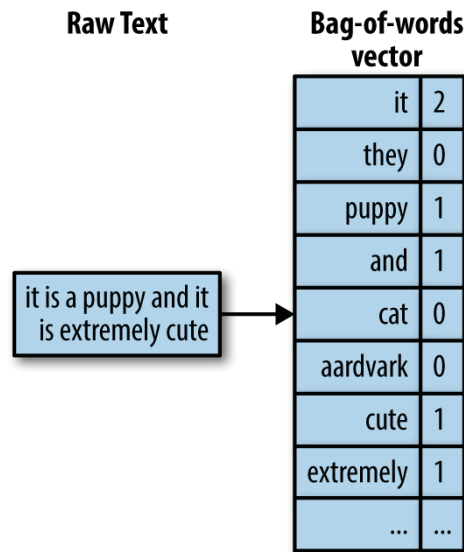


Figura 7.2: Exemplo de *bag of words* (Zheng e Casari, 2018)

Essa abordagem de representação não leva em consideração a estrutura original do texto, desconsiderando assim a ordem em que as palavras aparecem e qualquer relação semântica entre as mesmas. Dessa forma, essa representação implica na perda de informações dos dados originais, uma vez que a mesma se importa apenas com posição dos documentos no espaço vetorial onde cada palavra representa uma dimensão (Zheng e Casari, 2018).

7.2.2 n-Gramas

A representação de uma série de eventos em n-gramas consiste no conjunto ordenado de todas as sequências de n eventos contidas na série. Essa representação é obtida percorrendo a série de eventos com uma janela deslizante, onde cada n-grama é o conjunto de eventos presentes na janela a cada passo unitário da mesma. Esse processo é representado na figura 7.3.

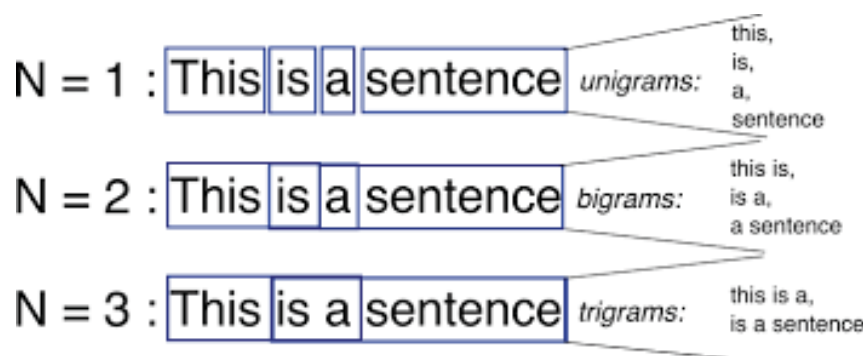


Figura 7.3: Exemplo de n-gramas (Stephanie, 2018)

Modelos baseados em n-gramas geralmente são mais precisos que modelos baseados simplesmente na distribuição de caracteres, entretanto exigem muitos recursos de processamento e memória mesmo com valores pequenos de n, onde n é o tamanho da janela (Zolotukhin e Hämmäläinen, 2013).

7.2.3 Bag-of-n-Grams

O *bag-of-n-grams* se trata de uma variação do *bag-of-words*, que tem como objetivo capturar mais informações sobre a estrutura dos textos analisados. Esse objetivo é alcançado ao se utilizar n-gramas como chaves do vetor de representação dos dados como forma de se levar em consideração sequências de palavras.

Essa forma de representação, apesar de carregar mais informações sobre os dados, gera espaços vetoriais bem maiores e esparsos; isso ocorre, pois um texto normalmente possui muito mais n-gramas do que palavras (Zheng e Casari, 2018). Sendo assim, quanto maior o valor de n , mais informações sobre o texto original e maior o custo para armazenar e processar as mesmas.

7.2.4 Tf-Idf

O uso de uma representação Tf-Idf (*Term Frequency-Inverse Document Frequency*) em detrimento das frequências de cada palavra tem como objetivo diminuir o impacto de palavras presentes em muitos documentos e, portanto, empiricamente menos informativas que palavras presentes apenas em uma pequena fração dos documentos (Pedregosa et al., 2011).

Para se atingir esse objetivo, a representação de uma palavra w em um documento d é dada pela multiplicação da frequência dessa palavra no documento d com o inverso da frequência de documentos que contém a palavra w considerando o total de documentos analisados N , conforme as equações abaixo (Zheng e Casari, 2018).

$$Tf - Idf(w, d) = BoW(w, d) * Idf(w, d)$$

$$Idf(w, d) = N / f(d), \text{ onde } f(d) \text{ é o número de documentos que contém } w$$

Alternativamente, pode-se equacionar $Idf(w, d)$ utilizando-se logaritmo como forma de transformar a escala em que os dados são representados. O uso da escala correta ao se representar os dados pode ser responsável por acentuar os dados relevantes e diminuir o impacto dos dados comuns, impactando assim no resultado do modelo adotado (Zheng e Casari, 2018).

7.3 EXPLORAÇÃO E LIMPEZA DOS DADOS OBTIDOS

Foram gerados traços de 622 aplicativos, sendo 303 classificados como maliciosos e 319 normais; desses aplicativos, 201 são de 2018, 243 de 2019 e 178 de 2020. Logo, a base de dados AndroBind está balanceada quanto às suas classes e ao longo do período de tempo analisado.

Cada linha dos traços gerados possui o formato mostrado na figura 7.4. Os campos são o nome que o SO dá para a tarefa que chamou a função logada, ID numérico da tarefa, CPU em que o processo está sendo executado, *latency-format* (opção do *ftrace* que quando ativada faz com que o traço gerado possua mais informações sobre o motivo das latências ocorridas), tempo da chamada da função (<segs>.<usegs>), função chamada e os parâmetros para o retorno da função.

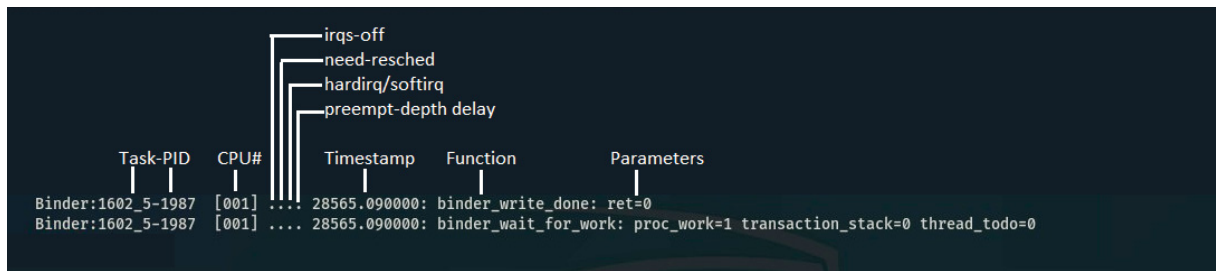


Figura 7.4: Formato do traço gerado no ftrace

Os traços gerados através do procedimento definido possuem uma grande variação de tamanho; o intervalo de variação vai de aproximadamente 14M até 2.3G, onde os maiores traços têm mais de um milhão de linhas. O grande tamanho de alguns traços torna o processamento dos mesmos pesado, sendo assim é necessário uma forma de filtrá-los ou um grande poder computacional. Dentre a lista de funções utilizadas e logadas no Binder, as funções `binder_command` e `binder_return` são responsáveis por identificar os comandos executados no Binder ao enviar ou receber mensagens. Por exemplo, o Binder utiliza o comando `BC_REQUEST_DEATH_NOTIFICATION` para ser alertado quando um serviço remoto é parado. Sendo assim, esses comandos identificam as operações realizadas no Binder e logo podem descrever o seu comportamento.

Dessa forma, os traços brutos do Binder foram filtrados para ser representados por traços de comandos do Binder. Os traços resultantes do filtro tiveram no máximo 32M, possibilitando assim o processamento dos mesmos mais facilmente. Com isso o comportamento dos aplicativos passou a ser representado por um traço formado pela sequência de comandos executados dentre os 37 comandos do Binder.

Após serem gerados os traços filtrados, foi observado a presença de *outliers* dentre os mesmos com relação ao tamanho dos traços, conforme pode ser observado no histograma da figura 7.5. Um dos possíveis motivos para isso, é que o *Monkey* pode não ter sido capaz de acionar muitas ações do aplicativo, gerando poucas chamadas de função no Binder ou o aplicativo gerou um excesso de comunicação através do Binder, o que pode indicar que ele está sendo utilizado de forma imprópria: por exemplo, para realizar a comunicação interna entre componentes do próprio aplicativo.

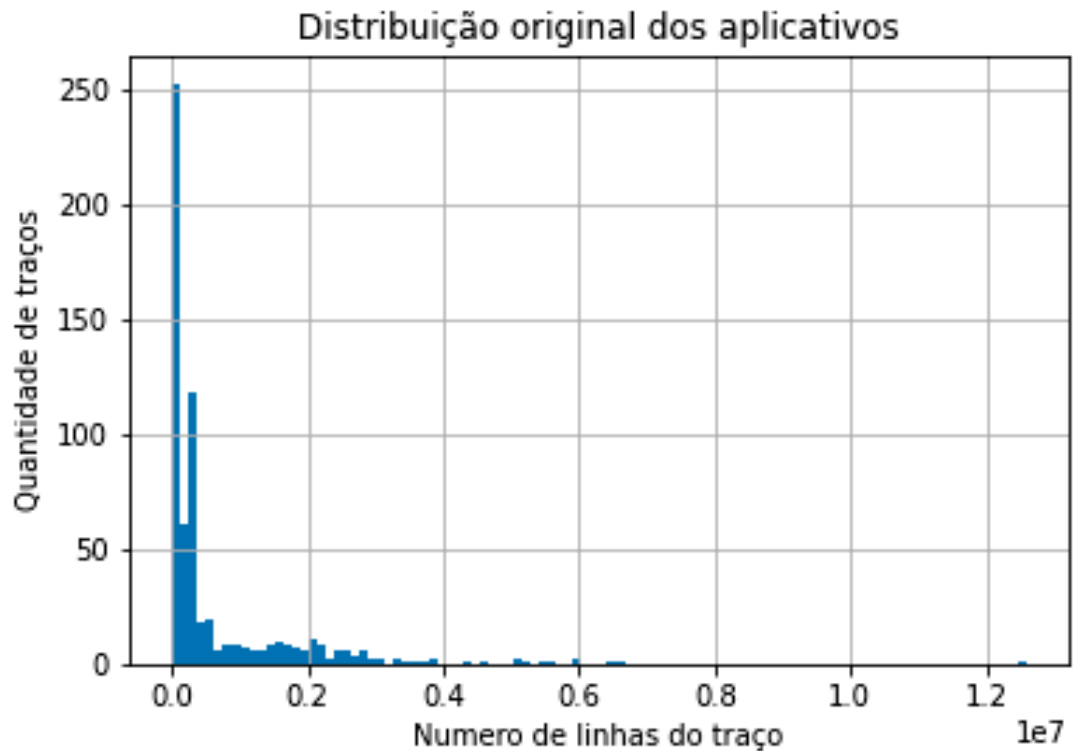


Figura 7.5: Histograma do tamanho dos traços

Esses *outliers* dificultam a generalização do modelo aprendido com os dados e logo devem passar por uma limpeza. Para isso, foram removidos da base de dados os aplicativos com menos de 1500 chamadas de função, por ser uma limitação conhecida no processo de geração dos traços, e posteriormente foram excluídos os demais *outliers*; foram considerados *outliers* os aplicativos que se desviaram mais de três vezes o desvio padrão em relação à média do total de chamadas de função por aplicativo. Após esse processo a nova base de dados possui 581 aplicativos, sendo 278 maliciosos e 303 normais, com distribuição pelo total de chamadas de função dada pelo histograma da figura 7.6.

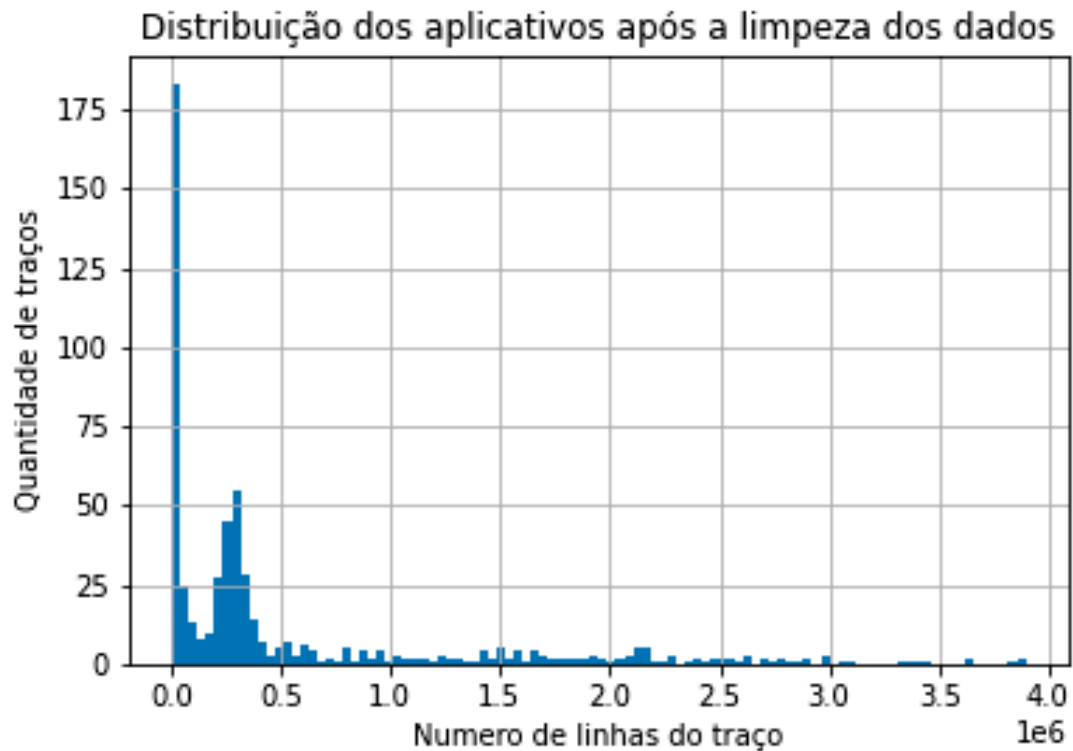


Figura 7.6: Histograma do tamanho dos traços após a retirada de *outliers*

Após a retirada dos *outliers*, os traços precisam ainda ser representados em um espaço vetorial comum a todos os aplicativos para serem utilizados em algoritmos de aprendizado de máquina. Foram utilizadas então as representações *Bag of Words* e Tf-Idf em suas versões comuns e suas variações utilizando n-gramas, com n igual a 2 e 3, como forma de representar a sequência de chamada das funções. Após gerar as 6 representações citadas, foram excluídas de cada uma delas as colunas constantes por não adicionarem informação relevante.

Ao se analisar as bases de dados geradas é possível observar que os aplicativos maliciosos realizam em média 20,8% mais chamadas ao Binder por aplicativo do que os normais. Essa diferença pode ser proveniente das atividades maliciosas realizadas nos mesmos, como por exemplo acessar arquivos sem a requisição do usuário, lançar anúncios, etc. A diferença por chamada pode ser observada no gráfico na Figura 7.7, que representa a relação entre a quantidade média de chamadas por aplicativos maliciosos e benignos.

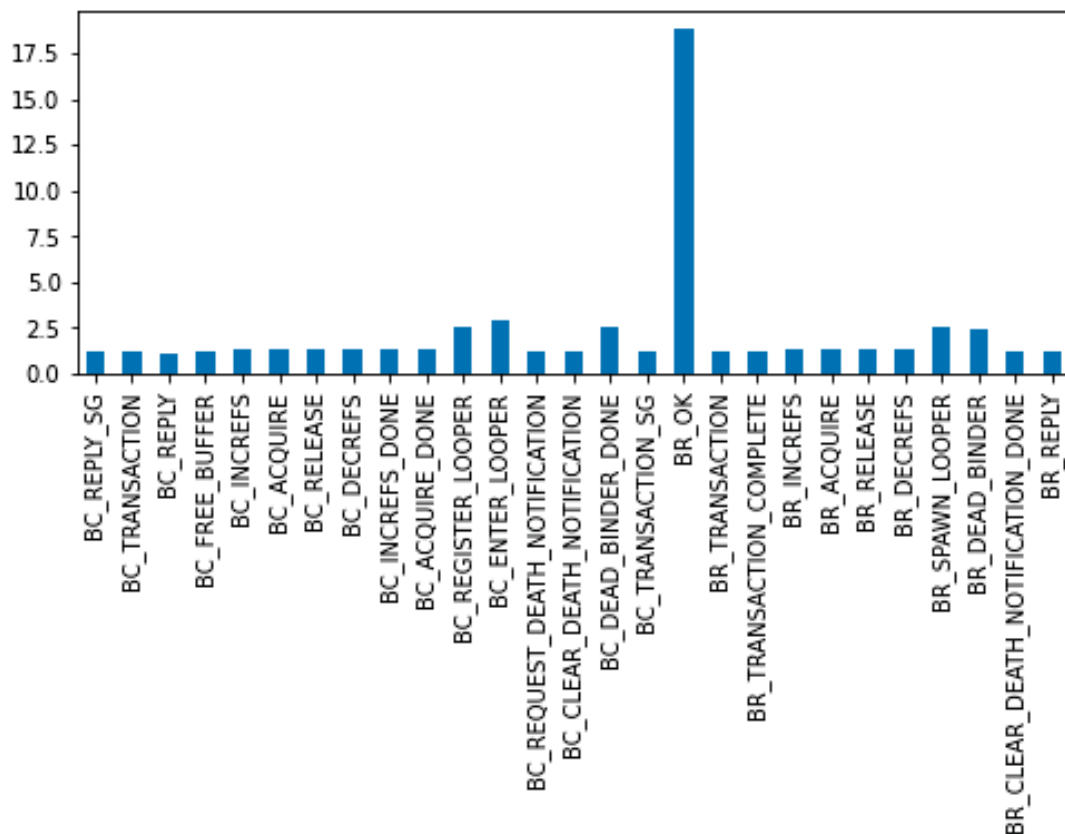


Figura 7.7: Divisão da média de chamadas por aplicativos maliciosos e benignos

Ao se analisar mais a fundo o gráfico da figura 7.7, percebe-se que os comandos `BC_REGISTER_LOOPER` e `BC_ENTER_LOOPER` são utilizados aproximadamente 3 vezes mais por aplicação em malwares; esses comandos são utilizados para registrar threads como threads do Binder, que são threads que podem ler e processar requisições vindas do Binder. Também percebe-se que o comando `BR_OK`, que é utilizado quando uma operação solicitada foi completa, é chamado aproximadamente 18 vezes mais em aplicações maliciosas.

Com isso, pode-se concluir que as aplicações maliciosas fazem mais requisições ao Binder. Essa diferença no uso médio do Binder entre aplicações maliciosas e benignas pode ser explicada pela necessidade dos malwares executarem suas ações maliciosas, além das ações normais utilizadas como fachada por esses aplicativos para enganar usuários, como requisitar acesso a arquivos, lançar anúncios na tela, enviar mensagens, etc.

7.4 BASE DE DADOS HÍBRIDA

Além da base de dados `AndroBind`, foi gerada uma base de dados híbrida que combina os dados da mesma com as permissões declaradas para cada um dos aplicativos analisados. Para representar as permissões nessa base de dados, cada permissão existente no conjunto de aplicativos analisado foi representada em uma coluna que possuía o valor 1 para os aplicativos que contém a mesma.

Essa base foi gerada com objetivo de avaliar a complementariedade dos dados coletados no `AndroBind` com dados clássicos em detectores de malware para Android e o impacto disso no desempenho do modelo de detecção. Além disso, esses dados possibilitam um melhor entendimento do processo de comunicação entre processos, que depende de permissões para

acontecer, e de como o Android combina diferentes mecanismos de segurança para mitigar vulnerabilidades no sistema.

7.5 LIMITAÇÕES

A metodologia adotada implica em algumas limitações ao projeto. Uma das limitações é a necessidade de ter acesso root ao sistema operacional para coletar as informações do Binder. Essa limitação não se trata de um problema, já que a aplicação funciona de maneira *offline* e logo é executada em um ambiente controlado.

Outra limitação é a necessidade de se iniciar o aplicativo antes de se iniciar a coleta de dados para saber o pid do mesmo; com isso ações executadas somente ao se iniciar o aplicativo não são refletidas no traço analisado.

A terceira limitação consiste na possibilidade de os eventos aleatórios gerados no *Monkey* não abrangerem todas as atividades existentes em um determinado aplicativo ou não reproduzirem o comportamento esperado para dar início a determinada ação; um exemplo onde isso ocorre são os aplicativos que exigem autenticação para executar atividades.

7.6 CONSIDERAÇÕES FINAIS

Neste capítulo foi abordada a metodologia para a obtenção dos traços de execução do Binder e técnicas de representação dos mesmos. Além disso, foi feita uma análise para identificar as diferenças de comportamento entre aplicativos benignos e maliciosos que podem ser observadas através da base de dados construída (AndroBind). Finalmente, algumas limitações inerentes à metodologia adotada foram identificadas.

8 AVALIAÇÃO DA PROPOSTA

Neste capítulo serão abordados os resultados obtidos com o modelo proposto. Inicialmente foram realizados testes sem adaptações na base de dados proposta, onde foi possível obter as primeiras conclusões sobre o modelo proposto e limitar o escopo de representações analisadas dos demais testes realizados para viabilizar a visualização e análise dos resultados.

Após isso, foram realizados testes com a base de treino e validação utilizando a base de dados proposta isoladamente e a base de dados híbrida contendo as informações referentes às permissões. Em ambos os casos a base de dados proposta foi testada em duas versões: original e considerando apenas os comandos BC_*. Ao se considerar apenas os comandos BC_* estão sendo desconsideradas as mensagens do Binder para o aplicativo analisado, considerando assim apenas as mensagens do aplicativo para o Binder.

Finalmente, após se avaliar os resultados na base de validação foi selecionado o melhor modelo avaliado para ser validado com a base de teste. Com esse resultado, foram verificadas a validade da proposta do presente trabalho e implicações desse resultado para aspectos relacionados à segurança do Android.

8.1 PIPELINE DO MODELO DE APRENDIZADO DE MÁQUINA

Com a base de dados preparada, deve-se definir o fluxo de execução das atividades a serem implementadas no modelo de aprendizado de máquina. O primeiro passo do *pipeline* consiste em normalizar os dados.

O próximo passo do fluxo adotado consiste em dividir a base de dados na proporção de 80% para treino e 20% para testes. Os dados de treino foram utilizados para treino e validação dos modelos através de validação cruzada k-Fold com k igual a 5, conforme adotado na literatura.

Na base de treino é então executado um algoritmo de seleção de características; foram realizados testes sem seleção de atributos e utilizando os seguintes algoritmos: *SelectPercentile*, *SelectKBest*, *TreeBased* e *RFECV*. Após a seleção de características é utilizado o algoritmo de aprendizado de máquina; foram testados os seguintes algoritmos: *Isolation Forest*, *Random Forest*, *Ada Boost* e *Multilayer Perceptron*. Os algoritmos de classificação e de seleção de características foram selecionados por estarem entre os mais populares, estando presentes em diversos trabalhos estado da arte, e possuírem um baixo tempo de treinamento, tornando assim os modelos de detecção menos custosos.

Finalmente, após a validação dos modelos testados foram selecionados os melhores modelos para serem testados com a base de testes. O fluxo descrito pode ser observado na figura 8.1.

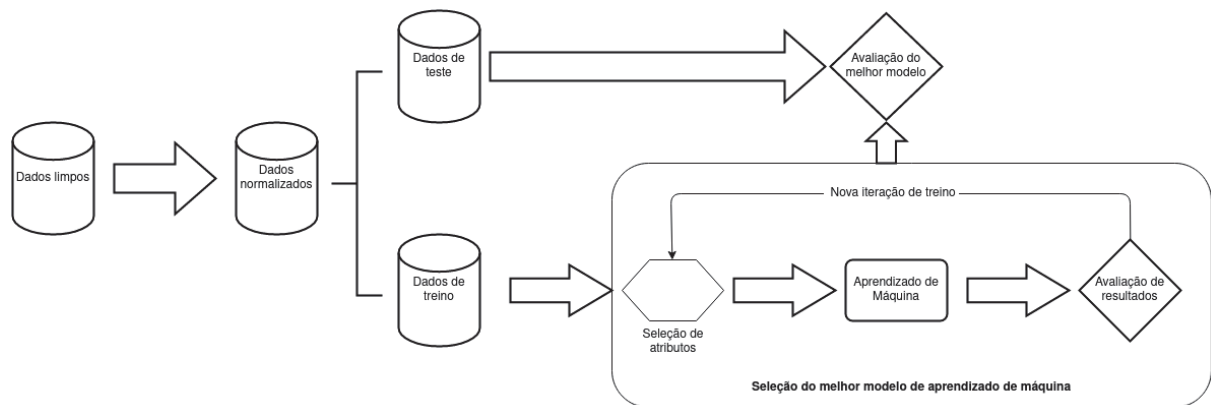
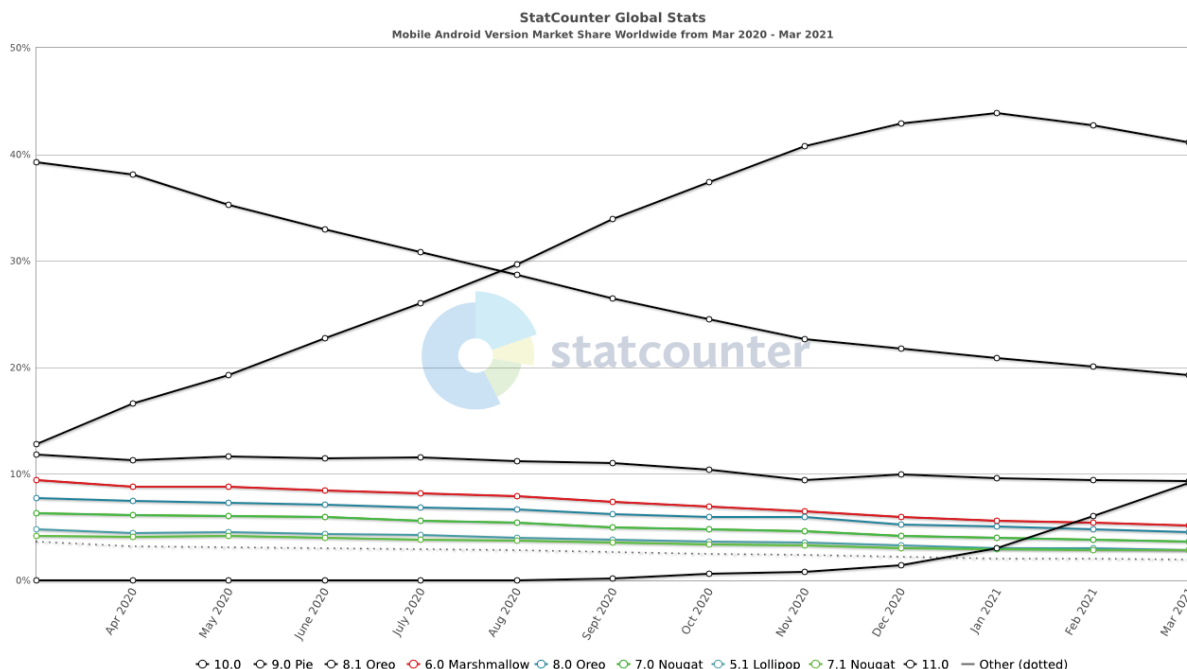


Figura 8.1: Pipeline de execução

8.2 PARÂMETROS DE IMPLEMENTAÇÃO

Para gerar os traços foi utilizado o emulador de Android do Android Studio em sua versão 3.6.3. O AVD (Dispositivo Virtual Android) utilizado possui perfil de hardware do Nexus 4.0 e imagem de sistema do Android Q (Android 10.0), com uma API de nível 29, com arquitetura x86. Além disso, o AVD foi configurado com 8 Gb de RAM e 50 Gb de disco.

Foi escolhido o Android 10 para a realização dos testes por ser a versão mais utilizada no mundo com 41.17% do *market share*, enquanto o Android 9 possui 19,31% no segundo lugar. O Android 10 é a versão mais utilizada desde agosto de 2020, conforme pode ser visto no gráfico 8.2 que mostra a evolução de *market share* das versões de Android nos últimos 12 meses.

Figura 8.2: Evolução de *market share* das versões de Android nos últimos 12 meses (StatCounter, 2021a)

Na fase de preparação dos dados foi utilizado o método `MinMaxScaler` na normalização dos mesmos. Para seleção de atributos: RFECV foi implementado com *Random Forest* como estimador, passo igual a 1% do número total de atributos originais e *10-fold cross-validation*; *selectPercentile* foi implementado com percentual de 10%; *selectKBest* foi implementado para

manter metade dos atributos utilizando qui-quadrado para avaliá-los; e *extraTreesClassifier* com 100 árvores.

Foram testados os algoritmos de aprendizado de máquina *random forest* e *isolation forest* com um número de árvores de 100, *ada Boost* com um estimador em 100 e *multilayer perceptron* com um número máximo de iterações de 300. Foi utilizado validação cruzada 5-fold nos testes para garantir a estabilidade dos resultados.

8.3 ESCOLHA DA MELHOR REPRESENTAÇÃO

Foram testadas duas formas de representação de dados: *bag-of-words* e *Tf-Idf*. Para se definir a representação mais adequada foi analisado o desempenho médio de cada uma das representações; dessa forma se diminui a influência de possíveis *outliers* utilizando algoritmos de aprendizado de máquina e seleção de atributos específicos.

Observou-se então que a representação *Tf-Idf* teve um resultado superior ao se analisar todas as métricas adotadas, conforme apresenta a tabela 8.1. Esse resultado mostra que ao se dar mais relevância ao comportamento anômalo dos aplicativos no uso do Binder a detecção de malware tem melhora de desempenho, indicando assim que realmente existem diferenças no uso do Binder por aplicações maliciosas e normais.

Representação	F1Score	Recall	Precision	Brier Score	ROC AUC
TF-IDF	0.688	0.753	0.656	0.350	0.643
Bag-of-Words	0.602	0.626	0.626	0.412	0.587

Tabela 8.1: Desempenho médio por representação dos dados

Outro parâmetro de representação testado foi a utilização de n-gramas como atributos. Para se avaliar esse parâmetro foi utilizada a mesma metodologia ao se considerar o desempenho médio de cada valor de n , porém foram consideradas apenas as representações com *Tf-Idf* por já terem sido identificadas como tendo desempenho superior. Foi observado então que o desempenho médio do modelo utilizando n igual a 2 e 3 foi muito superior ao utilizando unigramas, conforme pode ser observado na tabela 8.2. Esse resultado indica que a ordem em que os comandos são executados no Binder tem importância para detecção e não apenas a proporção com que cada um dos comandos é executado, apesar de ter sido identificado que aplicações maliciosas executam 20,8% mais comandos no Binder durante a exploração da base de dados.

n	F1Score	Recall	Precision	Brier Score	ROC AUC
3	0.717	0.807	0.667	0.335	0.660
2	0.707	0.783	0.667	0.341	0.654
1	0.640	0.670	0.634	0.383	0.615

Tabela 8.2: Desempenho médio por valor de n

Como foi concluído que as representações utilizando *Tf-Idf* e n-gramas são melhores para o tipo de dados adotado, as demais análises foram realizadas levando-se em consideração apenas a representação *Tf-Idf* com 3-gramas de modo a melhorar a visualização e avaliação de resultados.

8.4 RESULTADOS DOS TESTES

As tabelas 8.3, 8.5, 8.4 e 8.6 a seguir apresentam o desempenho obtido pelas representações propostas na fase de validação ordenados pela área abaixo da curva ROC. Para garantir a consistência do modelo selecionado foi utilizada a validação cruzada k-Fold com k igual a cinco, logo os resultados apresentados são as médias dos resultados obtidos ao se testar os cinco subconjuntos diferentes.

Seleção de Atributos	Aprendizado de Máquina	F1Score	Recall	Precisão	Brier Score	ROC AUC
<i>SelectKBest</i>	<i>Multilayer Perceptron</i>	0.841	0.834	0.855	0.153	0.844
<i>SelectPercentile</i>	<i>Multilayer Perceptron</i>	0.823	0.796	0.860	0.164	0.832
<i>TreeBased</i>	<i>Multilayer Perceptron</i>	0.735	0.743	0.735	0.257	0.743
<i>SelectPercentile</i>	<i>Ada Boost</i>	0.668	0.629	0.729	0.299	0.697
<i>SelectKBest</i>	<i>Random Forest</i>	0.664	0.631	0.725	0.304	0.694
<i>SelectPercentile</i>	<i>Random Forest</i>	0.662	0.620	0.722	0.304	0.693
<i>TreeBased</i>	<i>Random Forest</i>	0.620	0.583	0.673	0.340	0.658
<i>SelectKBest</i>	<i>Ada Boost</i>	0.619	0.587	0.667	0.347	0.652
-	<i>Ada Boost</i>	0.607	0.580	0.676	0.347	0.651
<i>RFECV</i>	<i>Ada Boost</i>	0.602	0.555	0.673	0.347	0.650
<i>TreeBased</i>	<i>Ada Boost</i>	0.604	0.555	0.676	0.347	0.650
-	<i>Multilayer Perceptron</i>	0.638	0.655	0.632	0.354	0.648
<i>RFECV</i>	<i>Random Forest</i>	0.611	0.585	0.647	0.353	0.644
-	<i>Random Forest</i>	0.584	0.559	0.625	0.373	0.626
<i>RFECV</i>	<i>Multilayer Perceptron</i>	0.604	0.618	0.604	0.379	0.621
<i>RFECV</i>	<i>Isolation Forest</i>	0.646	0.967	0.486	0.507	0.510
<i>SelectKBest</i>	<i>Isolation Forest</i>	0.646	0.991	0.480	0.521	0.497
-	<i>Isolation Forest</i>	0.646	0.991	0.480	0.521	0.497
<i>TreeBased</i>	<i>Isolation Forest</i>	0.642	0.980	0.479	0.523	0.496
<i>SelectPercentile</i>	<i>Isolation Forest</i>	0.641	0.979	0.477	0.525	0.493

Tabela 8.3: Resultados na base original

Seleção de Atributos	Aprendizado de Máquina	F1Score	Recall	Precisão	Brier Score	ROC AUC
<i>SelectPercentile</i>	<i>Multilayer Perceptron</i>	0.742	0.752	0.748	0.249	0.755
<i>SelectKBest</i>	<i>Multilayer Perceptron</i>	0.737	0.726	0.754	0.249	0.755
<i>TreeBased</i>	<i>Ada Boost</i>	0.712	0.682	0.754	0.265	0.736
<i>SelectPercentile</i>	<i>Random Forest</i>	0.701	0.667	0.748	0.274	0.727
<i>RFECV</i>	<i>Random Forest</i>	0.706	0.692	0.733	0.276	0.727
<i>SelectPercentile</i>	<i>Ada Boost</i>	0.698	0.650	0.759	0.272	0.726
<i>SelectKBest</i>	<i>Ada Boost</i>	0.682	0.648	0.737	0.288	0.715
<i>TreeBased</i>	<i>Random Forest</i>	0.680	0.643	0.734	0.292	0.710
<i>RFECV</i>	<i>Ada Boost</i>	0.682	0.661	0.711	0.296	0.706
<i>SelectKBest</i>	<i>Random Forest</i>	0.668	0.630	0.725	0.298	0.704
<i>TreeBased</i>	<i>Multilayer Perceptron</i>	0.687	0.688	0.700	0.300	0.703
-	<i>Ada Boost</i>	0.665	0.622	0.724	0.302	0.699
-	<i>Random Forest</i>	0.669	0.644	0.710	0.304	0.698
<i>RFECV</i>	<i>Multilayer Perceptron</i>	0.682	0.693	0.673	0.310	0.690
-	<i>Multilayer Perceptron</i>	0.649	0.672	0.635	0.349	0.655
<i>SelectPercentile</i>	<i>Isolation Forest</i>	0.646	0.980	0.483	0.519	0.496
<i>RFECV</i>	<i>Isolation Forest</i>	0.648	0.992	0.482	0.519	0.496
-	<i>Isolation Forest</i>	0.648	0.992	0.482	0.519	0.496
<i>TreeBased</i>	<i>Isolation Forest</i>	0.648	0.992	0.482	0.519	0.496
<i>SelectKBest</i>	<i>Isolation Forest</i>	0.634	0.961	0.475	0.534	0.482

Tabela 8.4: Resultados na base considerando apenas comandos BC_*

Seleção de Atributos	Aprendizado de Máquina	F1Score	Recall	Precisão	Brier Score	ROC AUC
<i>SelectPercentile</i>	<i>Multilayer Perceptron</i>	0.849	0.831	0.870	0.145	0.854
<i>SelectKBest</i>	<i>Multilayer Perceptron</i>	0.833	0.844	0.828	0.164	0.838
<i>SelectPercentile</i>	<i>Random Forest</i>	0.746	0.706	0.794	0.234	0.765
<i>TreeBased</i>	<i>Multilayer Perceptron</i>	0.749	0.754	0.747	0.245	0.757
<i>SelectPercentile</i>	<i>Ada Boost</i>	0.725	0.672	0.789	0.249	0.750
<i>RFECV</i>	<i>Random Forest</i>	0.713	0.706	0.732	0.277	0.727
<i>TreeBased</i>	<i>Random Forest</i>	0.698	0.664	0.746	0.281	0.722
<i>TreeBased</i>	<i>Ada Boost</i>	0.692	0.654	0.747	0.284	0.719
<i>SelectKBest</i>	<i>Random Forest</i>	0.692	0.649	0.752	0.283	0.718
<i>SelectKBest</i>	<i>Ada Boost</i>	0.685	0.643	0.738	0.288	0.713
<i>RFECV</i>	<i>Multilayer Perceptron</i>	0.705	0.736	0.682	0.298	0.704
<i>RFECV</i>	<i>Ada Boost</i>	0.675	0.653	0.713	0.307	0.698
-	<i>Random Forest</i>	0.668	0.627	0.721	0.305	0.696
-	<i>Multilayer Perceptron</i>	0.697	0.733	0.672	0.311	0.694
-	<i>Ada Boost</i>	0.641	0.622	0.672	0.337	0.666
<i>RFECV</i>	<i>Isolation Forest</i>	0.646	0.967	0.486	0.507	0.510
-	<i>Isolation Forest</i>	0.646	0.967	0.486	0.507	0.510
<i>TreeBased</i>	<i>Isolation Forest</i>	0.648	0.992	0.482	0.519	0.496
<i>SelectKBest</i>	<i>Isolation Forest</i>	0.634	0.961	0.475	0.534	0.482
<i>SelectPercentile</i>	<i>Isolation Forest</i>	0.634	0.961	0.475	0.534	0.482

Tabela 8.5: Resultados do modelo híbrido na base original

Seleção de Atributos	Aprendizado de Máquina	F1Score	Recall	Precisão	Brier Score	ROC AUC
<i>SelectPercentile</i>	<i>Multilayer Perceptron</i>	0.761	0.784	0.743	0.237	0.763
<i>SelectKBest</i>	<i>Multilayer Perceptron</i>	0.738	0.735	0.750	0.250	0.752
<i>RFECV</i>	<i>Random Forest</i>	0.724	0.682	0.776	0.252	0.747
<i>SelectPercentile</i>	<i>Random Forest</i>	0.704	0.652	0.775	0.265	0.735
<i>SelectPercentile</i>	<i>Ada Boost</i>	0.697	0.640	0.778	0.269	0.732
<i>RFECV</i>	<i>Multilayer Perceptron</i>	0.724	0.736	0.717	0.271	0.731
<i>RFECV</i>	<i>Ada Boost</i>	0.698	0.654	0.757	0.276	0.725
<i>SelectKBest</i>	<i>Ada Boost</i>	0.686	0.631	0.764	0.278	0.723
<i>TreeBased</i>	<i>Multilayer Perceptron</i>	0.711	0.705	0.718	0.278	0.721
<i>SelectKBest</i>	<i>Random Forest</i>	0.680	0.627	0.761	0.284	0.717
<i>TreeBased</i>	<i>Ada Boost</i>	0.669	0.611	0.753	0.291	0.708
<i>TreeBased</i>	<i>Random Forest</i>	0.672	0.625	0.739	0.295	0.706
-	<i>Ada Boost</i>	0.676	0.639	0.733	0.295	0.706
-	<i>Random Forest</i>	0.669	0.616	0.743	0.295	0.705
-	<i>Multilayer Perceptron</i>	0.644	0.661	0.630	0.356	0.646
<i>TreeBased</i>	<i>Isolation Forest</i>	0.646	0.967	0.486	0.507	0.510
-	<i>Isolation Forest</i>	0.646	0.991	0.480	0.521	0.497
<i>RFECV</i>	<i>Isolation Forest</i>	0.648	0.992	0.482	0.519	0.496
<i>SelectPercentile</i>	<i>Isolation Forest</i>	0.641	0.979	0.477	0.525	0.493
<i>SelectKBest</i>	<i>Isolation Forest</i>	0.634	0.961	0.475	0.534	0.482

Tabela 8.6: Resultados do modelo híbrido na base considerando apenas comandos BC_*

8.5 AVALIAÇÃO DOS RESULTADOS

Ao se analisar a taxa média de erro e *Brier Score* dos resultados percebeu-se que o algoritmo *Isolation Forest* não obteve um bom desempenho em nenhuma das variações da base de dados testadas; o algoritmo teve um *Brier Score* médio de 0.516 e um *recall* médio de 0.987 considerando todos os testes, indicando que o algoritmo não foi capaz de identificar as duas classes de aplicativos utilizando o método não supervisionado e considerou muito poucos aplicativos como anomalias. O *Brier Score* do *Isolation Forest* foi aproximadamente 75%, 72% e 94% superior aos do *Random Forest*, *Ada Boost* e *Multilayer Perceptron* respectivamente, dessa forma os resultados desse algoritmo não foram considerados nas demais análises para não poluir as conclusões.

Também é possível se observar que ao se combinar a base de dados proposta com os dados de permissões dos aplicativos analisados existe uma melhora no desempenho médio, indicando assim que esses dados são complementares na detecção de malware. A base híbrida aumentou o AUC médio em 3.94% e diminuiu o *Brier Score* médio em 8.77%.

Essa complementariedade pode ser explicada pelo mecanismo de segurança do Android que exige permissões do aplicativo para ter acesso a serviços; um exemplo que explica essa relação complementar entre o comportamento do Binder e as permissões de um aplicativo para atividades maliciosas é que caso um aplicativo tente fazer *upload* de dados para a internet, além de precisar passar pelo Binder para ter acesso a internet ele precisa ter a permissão para tal. Logo, ao incluir-se as permissões nos dados analisados o modelo passa a ter a informação sobre os possíveis destinos das transações intermediadas pelo Binder, diminuindo assim os erros na classificação.

Ao se comparar os resultados médios por representação utilizada também é observado que apesar dos testes utilizando apenas os comandos BC_* possuírem desempenho similar aos demais testes, o seu desempenho foi mais estável ao longo dos diferentes subconjuntos. Essa

diferença é apresentada na tabela 8.7, que apresenta as métricas médias e seus respectivos desvios padrão por representação utilizada.

Modelo	Base de Dados	F1Score		Recall		Precisão		Brier Score		ROC AUC	
		Média	Std	Média	Std	Média	Std	Média	Std	Média	Std
Dinâmico	BC_*	0.691	0.059	0.671	0.096	0.723	0.075	0.288	0.054	0.714	0.054
	BC_* e BR_*	0.659	0.099	0.634	0.131	0.699	0.097	0.311	0.079	0.687	0.079
Híbrido	BC_*	0.697	0.051	0.667	0.079	0.743	0.084	0.279	0.044	0.721	0.045
	BC_* e BR_*	0.718	0.073	0.699	0.090	0.747	0.093	0.267	0.069	0.735	0.067

Tabela 8.7: Desempenho e desvio padrão por representação

Essa maior estabilidade indica que os comandos enviados pelo Binder ao aplicativo geram ruído nos dados por não adicionarem informação relevante ao modelo. Essa conclusão pode ser explicada pelo fato de tanto o Binder quanto os demais serviços disponibilizados aos aplicativos através dele não alterarem seu comportamentos de acordo com o aplicativo de origem da requisição. Além disso, ao se utilizar a representação que considera apenas os comandos BC_* o treinamento do modelo é mais rápido, devido ao menor número de características utilizadas.

Concluí-se então que a melhor representação proposta para a base de dados analisada consiste no modelo híbrido com as 3-gramas de comandos BC_* feitos pelo aplicativo analisado ao Binder e as permissões do mesmo, cujos resultados com a base de validação estão apresentados na tabela 8.6.

Analisando mais a fundo os resultados apresentados na tabela 8.6, foi utilizado o AUC médio para selecionar os melhores modelos testados. Ao se comparar o AUC dos modelos, percebe-se que os dois modelos com melhor desempenho médio possuem um desvio padrão superior a 4% nos testes realizados durante a validação cruzada. Dessa forma foi escolhido o modelo que utiliza RFECV para seleção de atributos e *Random Forest* como modelo de classificação; o modelo escolhido teve AUC superior a 0.7 e *Brier Score* inferior a 0.3 em todos os subconjuntos testados durante a validação cruzada, indicando a consistência no desempenho do mesmo.

No modelo selecionado, o algoritmo RFECV selecionou 128 atributos para representar a base de dados. A sua curva ROC para cada um dos 5 subconjuntos utilizados na validação cruzada é apresentada na figura 8.3 e mostra que o seu desempenho se manteve consistente à medida que o limiar de discriminação foi variado.

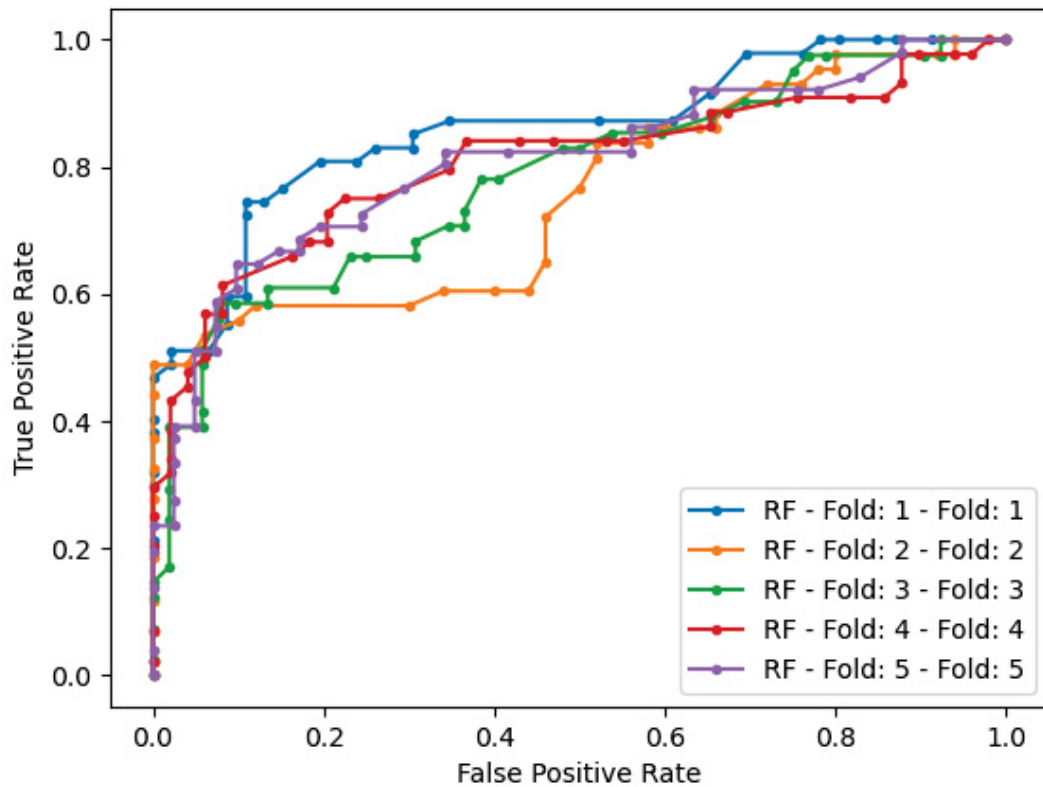


Figura 8.3: Curva ROC do modelo selecionado utilizando a base de validação

8.6 VALIDAÇÃO DO MODELO

Ao se utilizar a base de testes para validar o modelo escolhido foi encontrado o desempenho apresentado nas métricas da tabela 8.8 abaixo. A medida de *Brier Score* abaixo de 0.24, combinado com um AUC de 0.747 aproximadamente e valores altos de F1Score e precisão ao se avaliar dados inéditos para o modelo treinado mostram que os dados de execução do Binder são eficientes na detecção de aplicações maliciosas.

O *recall* aproximadamente 0.12 abaixo da precisão indica que o modelo teve um índice de falsos negativos superior ao de falsos positivos. Esse fenômeno também ocorreu na fase de treinamento com a maioria dos modelos testados, conforme pode ser observado na seção 8.4 e pode ser explicado pela limitação ao gerar a base de dados descrita na seção 7.5. Essa limitação consiste na possibilidade de em alguns aplicativos a simulação não abranger todas as atividades existentes e consequentemente não acionar a atividade maliciosa do mesmo. Nesse caso, como a atividade maliciosa não seria executada durante o monitoramento, ela não pode ser identificada nos traços.

Seleção de Atributos	Aprendizado de Máquina	F1Score	Recall	Precisão	Brier Score	ROC AUC
<i>RFECV</i>	<i>Random Forest</i>	0.702	0.647	0.767	0.239	0.748

Tabela 8.8: Desempenho do modelo selecionado na base de testes

Ao se comparar este resultado com o obtido na base de validação para o mesmo modelo percebe-se que o modelo não teve queda em seu desempenho, demonstrando que o mesmo se generalizou bem a novos dados e consequentemente não houve *overfitting* na fase de treinamento e validação. A curva ROC de cada um dos 5 subconjuntos treinados durante a validação cruzada foi traçada para os dados da base de testes e está apresentada na figura 8.4, abaixo para comprovar a semelhança de desempenho do modelo selecionado para a base de validação e para a base de testes, que até então era desconhecida para o mesmo.

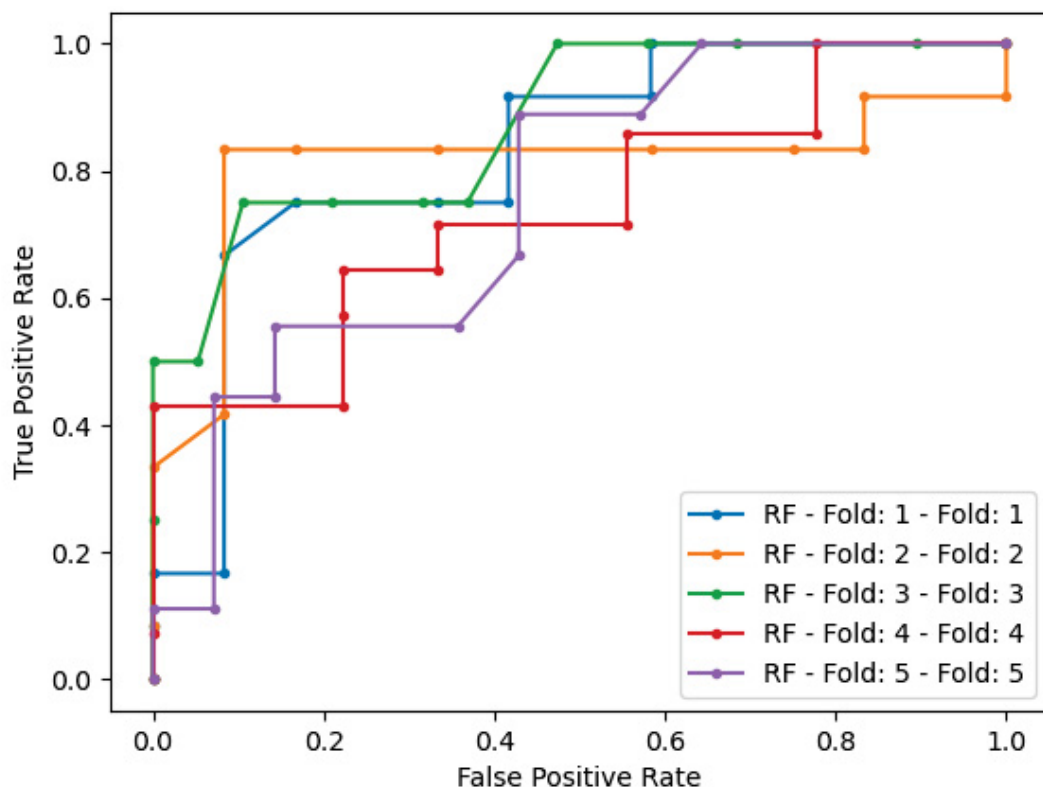


Figura 8.4: Curva ROC do modelo selecionado utilizando a base de testes

Os resultados apresentados mostraram a validade da prova de conceito realizada. Dessa forma, os dados de execução do Binder são eficientes em representar o comportamento de aplicativos e fornecem informações complementares aos atributos estáticos tradicionalmente utilizados.

Sendo assim, combinação de atributos dinâmicos do Binder com esses atributos pode ser a escolha para detecção de aplicações maliciosas no Android, uma vez que tal proposta combinaria o alto desempenho desses atributos estáticos e a robustez contra técnicas de ofuscação dos atributos dinâmicos, sem a necessidade de alterações nos aplicativos analisados ou nos dispositivos para coletar esses dados. A eficiência dessa proposta foi mostrada no trabalho (Lemos et al., 2021), onde esse modelo foi testado e obteve um desempenho de aproximadamente 0.88 de AUC.

8.7 CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentados a metodologia de aplicação das técnicas de aprendizagem de máquina para detecção de malware e os resultados alcançados para cada uma das representações testadas. Com isso, a proposta do trabalho foi validada e foram obtidas conclusões sobre a relação entre atividades maliciosas e os dados utilizados ao se analisar os resultados para a escolha da melhor representação dos mesmos.

9 CONCLUSÃO

O sistema operacional Android é, atualmente, o mais utilizado no mundo. Essa popularização dos dispositivos Android trouxe diversos desafios de segurança para a plataforma. Dentre esses desafios, está a disseminação de aplicativos maliciosos cada vez mais diversos e complexos. Entre os desafios impostos pela evolução nos malwares, está a necessidade de mecanismos de detecção eficientes e com a capacidade de identificar os mais diversos tipos de ações maliciosas.

Com esse intuito, foi proposta a utilização da comunicação entre processos no Android como fonte de dados para um mecanismo de detecção de aplicações maliciosas. Essa abordagem permite interceptar requisições de todo tipo de ações maliciosas executadas por aplicações, uma vez que no Android as aplicações são executadas em processos isolados e sem privilégios. Dessa forma, foi escolhido o Binder, que é o módulo do Kernel responsável pela comunicação entre processos, como o ponto de coleta desses dados; permitindo assim que a coleta seja realizada sem modificar as aplicações analisadas e o mecanismo de IPC do Android.

Como não foi encontrada na literatura proposta similar, foi gerada uma base de dados (AndroBind) para validar a mesma. Essa base foi gerada utilizando a ferramenta ftrace para coletar traços de funções executadas no Binder associadas ao processo da aplicação analisada, enquanto a mesma é estimulada com uma abordagem *stateless*. A base AndroBind possui traços de 622 aplicações, sendo 48,7% maliciosas.

Ao se analisar o AndroBind, percebe-se que aplicações maliciosas executam comandos 20,8% mais que aplicações benignas e registram mais threads como threads do Binder. Esse uso adicional de recursos do Binder, se deve às ações maliciosas executadas além de suas funções normais.

A base de dados foi então avaliada utilizando diferentes representações para os traços como prova de conceito para a proposta. Nessas avaliações foi validada a prova de conceito, mostrando assim que o uso do Binder como fonte de dados para detecção de malware é uma abordagem possível. Além disso, os resultados mostraram que o uso apenas das chamadas com origem no aplicativo analisado é mais eficiente que o uso de toda comunicação com o mesmo; mostrando assim que o Binder não altera seu comportamento ao se comunicar com aplicações maliciosas e assim as mensagens do Binder para as mesmas representam ruído nos dados. Outra conclusão dos resultados avaliados é que o uso de n-gramas para representação dos dados aprimorou os resultados observados, mostrando que a sequência de chamadas, além da proporção entre as mesmas, é importante para caracterizar ações maliciosas.

Foi avaliada também a eficácia de se combinar os dados do AndroBind com as permissões dos respectivos aplicativos, caracterizando assim um modelo híbrido. Esse modelo obteve um resultado um pouco superior ao uso do AndroBind sozinho, mostrando assim a complementariedade desses dados. Essa complementariedade pode ser explicada, pois combinando os dados é possível identificar, além das chamadas feitas a outros processos, os possíveis destinos dessas chamadas.

A validade dessa abordagem para detecção de malware representa um avanço na segurança do ambiente Android, já que se trata de um método imune à ofuscação de código e que é capaz de interceptar todo tipo de atividade maliciosa através de diferentes versões do Android, uma vez que o Binder como centralizador do IPC é comum a todas as implementações do Android.

Finalmente, como forma de continuar aprimorando a abordagem proposta, trabalhos futuros podem considerar diferentes formas de representar os traços coletados como o uso de mais informações, além dos identificadores das funções `binder_command` e `binder_return`, e diferentes métodos de representação como o `word2vec` (Rong, 2014); além disso pode ser considerada a adição de outros parâmetros estáticos utilizados na literatura no modelo híbrido, como forma de acrescentar mais informações sobre as aplicações analisadas. Também pode-se estudar as possíveis vantagens da utilização de um método *stateful* para estimular as aplicações, de modo a garantir uma maior cobertura da execução do código. Outra abordagem interessante para um trabalho futuro seria analisar as famílias de malware isoladamente, de forma a compreender melhor seus comportamentos específicos em relação a comunicação através do Binder.

REFERÊNCIAS

- Ahmadi, M., Ulyanov, D., Semenov, S., Trofimov, M. e Giacinto, G. (2016). Novel feature extraction, selection and fusion for effective malware family classification. Em *Proceedings of the sixth ACM conference on data and application security and privacy*, páginas 183–194.
- Ajiri, V., Butakov, S. e Zavarsky, P. (2020). Detection efficiency of static analyzers against obfuscated Android malware. Em *2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*. IEEE.
- Allix, K., Bissyandé, T. F., Klein, J. e Le Traon, Y. (2016). Androzoo: Collecting millions of Android apps for the research community. Em *Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16*, páginas 468–471, New York, NY, USA. ACM.
- Alves, A. (2019 (Acessado 21 de março de 2020)). *Brasil sofreu 15 bilhões de ataques cibernéticos em 3 meses*.
- Alzaylaee, M. K., Yerima, S. Y. e Sezer, S. (2020). DL-droid: Deep learning based Android malware detection using real devices. *Computers & Security*, 89:101663.
- Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H. e Rieck, K. (2014). Drebin: Effective and explainable detection of Android malware in your pocket. Em *Ndss*.
- Arshad, S., Shah, M. A., Wahid, A., Mehmood, A., Song, H. e Yu, H. (2018). Samadroid: A novel 3-level hybrid malware detection model for Android operating system. *IEEE Access*, 6:4321–4339.
- Artenstein, N. e Revivo, I. (2014). Man in the binder: He who controls IPC, controls the droid. *BlackHat Europe 2014*.
- Azeez, N. A., Bada, T. M., Misra, S., Adewumi, A., der Vyver, C. V. e Ahuja, R. (2019). Intrusion detection and prevention systems: An updated review. Em *Data Management, Analytics and Innovation*, páginas 685–696. Springer Singapore.
- Barbhuiya, S., Kilpatrick, P. e Nikolopoulos, D. S. (2020). DroidLight. Em *Proceedings of the 21st International Conference on Distributed Computing and Networking*. ACM.
- Bird, T. (2009). Measuring function duration with ftrace. *Proceedings of the Linux Symposium*.
- Boulis, C. e Ostendorf, M. (2005). Classification by augmenting the bag-of-words representation with redundancy-compensated bigrams. Em *Real-time bag of words, approximately*, páginas 9–16. Citeseer.
- Bridges, R. A., Glass-Vanderlan, T. R., Iannacone, M. D., Vincent, M. S. e Chen, Q. G. (2020). A survey of intrusion detection systems leveraging host data. *ACM Computing Surveys (CSUR)*, 52(6):1–35.

- Cai, H., Meng, N., Ryder, B. e Yao, D. (2019). Droidcat: Effective Android malware detection and categorization via app-level profiling. *IEEE Transactions on Information Forensics and Security*, 14(6):1455–1470.
- Ceschin, F., Oliveira, L. E. S. e Grégio, A. R. A. (2019). Aprendizado de máquina para segurança: Algoritmos e aplicações. *Capítulo 2 do Livro de Minicursos do XIX Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*.
- Debar, H., Curry, D. A. e Feinstein, B. (2007). The intrusion detection message exchange format (IDMEF). *RFC*, 4765:1–157.
- Desnos, A. et al. (2013). Androguard-reverse engineering, malware and goodware analysis of Android applications. *URL code.google.com/p/androguard*, 153.
- Fan, M., Liu, J., Luo, X., Chen, K., Chen, T., Tian, Z., Zhang, X., Zheng, Q. e Liu, T. (2016). Frequent subgraph based familial classification of Android malware. Em *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, páginas 24–35.
- Fritz, C., Arzt, S., Rasthofer, S., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Ocateau, D. e McDaniel, P. (2013). Highly precise taint analysis for Android applications. Relatório técnico, Technische Universität Darmstadt.
- Google (2019a). Monkey para testes de iu/apps. <https://developer.android.com/studio/test/monkey>.
- Google (2019b). Platform architecture: Android developers. <https://developer.android.com/guide/platform>.
- Google (2020). Usando binder ipc. <https://source.android.com/devices/architecture/hidl/binder-ipc>.
- Hamed, Y. S. I., AbdulKader, S. N. A. e Mostafa, M.-S. M. (2019). Mobile malware detection: A survey. *International Journal of Computer Science and Information Security (IJCSIS)*, January 2019.
- Idika, N. e Mathur, A. P. (2007). A survey of malware detection techniques. *Purdue University*, 48(2). Citeseer.
- Inc., P. (2005). *Open binder documentation*. <http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html>.
- Japkowicz, N. e Shah, M. (2011). *Evaluating Learning Algorithms: A Classification Perspective*. Cambridge University Press.
- Jyothsna, V., Prasad, V. R. e Prasad, K. M. (2011). A review of anomaly based intrusion detection systems. *International Journal of Computer Applications*, 28(7):26–35.
- Kabiri, P. e Ghorbani, A. (2005). Research on intrusion detection and response: A survey. *International Journal of Network Security*, 1:84–102.
- Kouliaridis, V., Barmapsalou, K., Kambourakis, G. e Chen, S. (2020). A survey on mobile malware detection techniques. *IEICE Transactions on Information and Systems*, E103.D(2):204–211.

- Kukunas, J. (2015). Ftrace. Em *Power and Performance in Enterprise Systems*, páginas 167–178. Elsevier.
- Kumar, M. e Hanumanthappa, M. (2015). Cloud based intrusion detection architecture for smartphones. Em *2015 International Conference on Innovations in Information, Embedded and Communication Systems (ICIECS)*, páginas 1–6.
- Kumar, S. e Shukla, S. K. (2020). The state of Android security. Em *Cyber Security in India*, páginas 17–22. Springer Singapore.
- Lemos, R., Heinrich, T. e Maziero, C. (2021). Utilizando metadados de aplicações e comunicação entre processos para identificar ameaças no android. Em *Anais do XXI Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, páginas 71–84, Porto Alegre, RS, Brasil. SBC.
- Liu, P., Wang, W., Luo, X., Wang, H. e Liu, C. (2020). Nsdroid: efficient multi-classification of Android malware using neighborhood signature in local function call graphs. *International Journal of Information Security*, páginas 1–13.
- Millar, S., McLaughlin, N., Martinez del Rincon, J., Miller, P. e Zhao, Z. (2020). DANdroid: A multi-view discriminative adversarial network for obfuscated Android malware detection. Em *Proceeding of the 10th ACM Conference on Data and Application Security and Privacy*.
- Namanya, A. P., Cullen, A., Awan, I. U. e Disso, J. P. (2018). The world of malware: An overview. Em *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*, páginas 420–427. IEEE.
- Nirumand, A., Zamani, B. e Ladani, B. T. (2018). VAnDroid: A framework for vulnerability analysis of Android applications using a model-driven reverse engineering technique. *Software: Practice and Experience*, 49(1):70–99.
- Nisioti, A., Mylonas, A., Yoo, P. D. e Katos, V. (2018). From intrusion detection to attacker attribution: A comprehensive survey of unsupervised methods. *IEEE Communications Surveys Tutorials*, 20(4):3369–3388.
- Onwuzurike, L., Mariconti, E., Andriotis, P., Cristofaro, E. D., Ross, G. e Stringhini, G. (2019). Mamadroid: Detecting Android malware by building markov chains of behavioral models (extended version). *ACM Transactions on Privacy and Security (TOPS)*, 22(2):1–34.
- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. e Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.
- Pharate, A., Bhat, H., Shilimkar, V. e Mhetre, N. (2015). Classification of intrusion detection system. *International Journal of Computer Applications*, 118(7).
- Qiu, J., Nepal, S., Luo, W., Pan, L., Tai, Y., Zhang, J. e Xiang, Y. (2019). Data-driven Android malware intelligence: A survey. Em *Machine Learning for Cyber Security*, páginas 183–202. Springer International Publishing.

- Ribeiro, J., Saghezchi, F. B., Mantas, G., Rodriguez, J., Shepherd, S. J. e Abd-Alhameed, R. A. (2019). An autonomous host-based intrusion detection system for Android mobile devices. *Mobile Networks and Applications*, 25(1):164–172.
- Rogers, R., Lombardo, J., Mednieks, Z. e Meike, B. (2009). *Android Application Development: Programming with the Google SDK*. O'Reilly Media, Incorporated. https://books.google.com.br/books?id=uU_3wAEACAAJ.
- Rong, X. (2014). word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*.
- Schreiber, T. (2011). Android binder. Dissertação de Mestrado, Ruhr University, Bochum, Germany. <http://www.nds.rub.de/media/attachments/files/2012/03/binder.pdf>.
- Shahzad, R. K. (2018). Android malware detection using feature fusion and artificial data. Em *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing*, páginas 702–709.
- Sharmeen, S., Huda, S., Abawajy, J. e Hassan, M. M. (2020). An adaptive framework against Android privilege escalation threats using deep learning and semi-supervised approaches. *Applied Soft Computing*, 89:106089.
- Shen, D., Zhang, Z., Ding, X., Li, Z. e Deng, R. (2017). H-binder: A hardened binder framework on Android systems. Em *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, páginas 24–43. Springer International Publishing.
- Shrivastava, G. e Kumar, P. (2019). Intent and permission modeling for privacy leakage detection in Android. *Energy Systems*, páginas 1–14. <https://doi.org/10.1007/s12667-019-00359-7>.
- Solorio-Fernández, S., Carrasco-Ochoa, J. A. e Martínez-Trinidad, J. F. (2019). A review of unsupervised feature selection methods. *Artificial Intelligence Review*, 53(2):907–948.
- StatCounter (2021a). *Mobile Android Version Market Share Worldwide*. <https://gs.statcounter.com/android-version-market-share/mobile/worldwide/#monthly-202003-202103>.
- StatCounter (2021b). *Operating System Market Share Worldwide*. <https://gs.statcounter.com/os-market-share#monthly-201103-202103>.
- Stephanie (2018). Capstone report. https://rstudio-pubs-static.s3.amazonaws.com/94682_5a41796c9e1a4345b6ba72275612768c.html.
- Stultz, J. (2017 (Acessado 30 de março de 2021)). *binder: Add support for scatter-gather*. <https://lore.kernel.org/patchwork/patch/757477/>.
- Tam, K., Khan, S. J., Fattori, A. e Cavallaro, L. (2015). CopperDroid: Automatic reconstruction of Android malware behaviors. Em *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society.
- Vasilomanolakis, E., Karuppayah, S., Mühlhäuser, M. e Fischer, M. (2015). Taxonomy and survey of collaborative intrusion detection. *ACM Computing Surveys*, 47(4):1–33.

- Wressnegger, C., Schwenk, G., Arp, D. e Rieck, K. (2013). A close look on n -grams in intrusion detection: Anomaly detection vs. classification. Em *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, AISEC '13, página 67–76, New York, NY, USA. Association for Computing Machinery.
- Xu, K., Li, Y., Deng, R., Chen, K. e Xu, J. (2019). Droidevolver: Self-evolving Android malware detection system. Em *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, páginas 47–62.
- Xu, K., Li, Y., Deng, R. H. e Chen, K. (2018). Deeprefiner: Multi-layer Android malware detection system applying deep neural networks. Em *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, páginas 473–487.
- Yeon, J., Im, J., Jeon, B., Lee, S., Lee, E. e Bahn, H. (2016). Design and implementation of kernel binder cache to accelerate Android IPC. Em *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, páginas 22–28.
- Yerima, S. Y. e Sezer, S. (2019). Droidfusion: A novel multilevel classifier fusion approach for Android malware detection. *IEEE Transactions on Cybernetics*, 49(2):453–466.
- Zachariah, R., Akash, K., Yousef, M. S. e Chacko, A. M. (2017). Android malware detection a survey. Em *2017 IEEE International Conference on Circuits and Systems (ICCS)*. IEEE.
- Zheng, A. e Casari, A. (2018). *Feature engineering for machine learning: principles and techniques for data scientists*. "O'Reilly Media, Inc."
- Zolotukhin, M. e Hämmäläinen, T. (2013). *Detection of Anomalous HTTP Requests Based on Advanced N-gram Model and Clustering Techniques*, páginas 371–382. Springer.