Utilizando Metadados de Aplicações e Comunicação entre Processos para Identificar Ameaças no Android

Rodrigo Lemos, Tiago Heinrich, Carlos Maziero

Departamento de Informática
Universidade Federal do Paraná (UFPR)
Curitiba – PR – Brasil

{rglemos,theinrich,maziero}@inf.ufpr.br

Abstract. *The leadership of Android in mobile device market share and the growth in the number and sophistication of cyber attacks highlight the need to study threat identification techniques in this environment. A popular approach for threat identification is the use of IDSs, which can explore multiple strategies to accomplish this. This work introduces a hybrid approach for detecting malware threats. This approach consists in using machine learning to detect malware based in static data and dynamic inter-process communication data from the analyzed apps. The results achieved shows that the proposed model has benefits in identifying threats and has an identification rate around 87%.*

Resumo. *A predominância do Android no mercado de dispositivos móveis e o aumento significativo na quantidade e na sofisticação dos ataques cibernéticos, destaca a necessidade do estudo de técnicas de identificação de ameaças no meio de dispositivos móveis. Uma estratégia popular para a identificação de ameças é a utilização de um sistema de detecção de intrusão, que pode explorar diferentes estratégias para realizar a identificação das ameaças. Este estudo apresenta uma estratégia para a identificação de ameaças utilizando uma base de dados híbrida, explorando dados extraídos de aplicações e da comunicação entre processos para treinar modelos de aprendizado de máquina para efetuar a identificação de ameças. Os resultados demonstram que um modelo híbrido traz benefícios para a identificação de ameaças em dispositivos móveis, com uma taxa de identificação em torno de 87%.*

1. Introdução

O Android é um sistema operacional *open source* baseado no núcleo do Linux, primeiramente desenvolvido para aparelhos móveis. Devido à sua ampla utilização, foi definida a *Open Handset Alliance (OHA)*, um consórcio de empresas que define padrões abertos para telefonia móvel [Rogers et al. 2009].

O Android é o sistema operacional mais utilizado no mundo [Golrang et al. 2021]. Devido a essa relevância de *market share*, o Android se tornou um alvo de desenvolvedores de *malware* [Kumar and Shukla 2020]. Essa evolução nas ameaças ao Android, juntamente ao fato do mesmo armazenar informações sensíveis dos usuários, como dados bancários e documentos confidenciais [Afonso et al. 2013], tornaram prioritário o desenvolvimento de mecanismos para proteger esses dispositivos.

Nos ambientes móveis, os atacantes constantemente se utilizam de técnicas de engenharia social para comprometer os dispositivos, normalmente oferecendo aplicativos maliciosos como aplicativos que estão em alta no momento [Chebyshev 2021]. A complexidade em se obter acesso aos dispositivos móveis remotamente implica nessa necessidade de ações do usuário (instalar um aplicativo, clicar em uma *Uniform Resource Locator (URL)*, conectar-se em uma rede sem fio, etc) para comprometer o mesmo. Dessa forma, é de grande importância para a segurança de um dispositivo Android avaliar as aplicações a serem instaladas no mesmo.

Essa característica de exploração de engenharia social para comprometer dispositivos móveis pode ser exemplificada pelo `covidLock`, um *ransomware* que promete fornecer acompanhamento em tempo real da pandemia de coronavirus. Ao enganar o usuário para ter permissão de mudar a senha do dispositivo, executa um ataque de bloqueio de tela no dispositivo e solicita um resgate do mesmo [CISA 2020]. De acordo com a Kaspersky, 2020 apresentou um aumento significativo nas ameaças financeiras em dispositivos móveis e manteve o número em outras famílias de malwares [Chebyshev 2021].

A detecção de *malwares* para Android tem cada vez mais relevância na segurança do sistema. Embora usualmente os aplicativos para Android sejam obtidos de lojas, onde são verificados antes de serem disponibilizados, existem aplicações maliciosas se passando por legítimas nas mesmas. Mesmo na loja oficial (*Google Play*) podem ser encontradas aplicações maliciosas não detectadas pelo mecanismo de detecção do *Google Bouncer* [Rahman et al. 2017].

Os modelos híbridos para detecção de *malware* propostos até o momento utilizam, em sua maioria, como dados dinâmicos chamadas de sistema e informações do uso de *hardware* dispositivo [Zachariah et al. 2017]. Além disso, também existem propostas de trabalhos que utilizam o tráfego de rede [Kato et al. 2020] ou chamada de *Intents* [Alzaylaee et al. 2020], que exige instrumentação para coleta dos dados, para detectar tipos específicos de ataques.

Dessa forma, este trabalho propõe uma nova abordagem para detecção de *malwares* no Android baseada em um modelo híbrido que combina dados referentes à comunicação entre processos realizadas pelos aplicativos analisados e informações estáticas dos mesmos. A proposta tem como diferencial a utilização da comunicação entre processos para caracterizar o comportamento dos aplicativos e sua combinação com dados estáticos dos mesmos, com o objetivo de criar um modelo eficiente e resiliente de detecção de *malwares*. As seguintes contribuições são apresentadas no trabalho:

- Uma discussão sobre uso de dados híbridos para a identificação de ameaças no Android;
- Uma nova técnica para identificação de aplicações maliciosas no Android através de um conjunto de dados híbridos;
- Uma estratégia para observação de comportamentos e identificação de ameaças utilizando técnicas de aprendizado de máquina; e
- Uma base de dados nomeada AndroBlend[1], contendo dados híbridos, com o intuito na identificação de ameaças no Android.

[1]O AndroBlend *dataset* pode ser encontrado em `https://github.com/Rodrigo-Lemos/AndroBlend`.

Este trabalho está estruturado em cinco seções. A Seção 2 descreve os conceitos relacionados a segurança e detecção de *malware* no Android. Na Seção 3 são apresentados os trabalhos relacionados a este estudo. A técnica proposta e a estratégia de avaliação são apresentadas na Seção 4. A Seção 5 apresenta os resultados dos experimentos e a discussão sobre os mesmos. Por fim, a Seção 6 traz as conclusões do trabalho.

2. Fundamentação Teórica

Esta seção apresenta os principais conceitos para o entendimento deste trabalho, descrevendo como a detecção de *malware* ocorre no Android, tipos de análises possíveis e processo de comunicação no Android.

2.1. Detecção de Malware no Android

O desenvolvimento de mecanismos de detecção para o ambiente Android é um tema em alta e, consequentemente, foi abordado em diversas pesquisas nos últimos anos. Um relatório da Avira em 2020 indicou que cresceu em 30% a preocupação de americanos com tópicos relacionados a privacidade, vazamento de dados e segurança de dispositivos móveis desde o início da pandemia de COVID-19 [Avira 2020].

Por outro lado, os *malwares* para Android também têm evoluído constantemente para burlar esses mecanismos de detecção. Por exemplo, o uso combinado de técnicas de ofuscação de código tem um grande impacto na eficiência em mecanismos de análise estática, diminuindo as suas taxas de detecção em mais de 50% na média [Ajiri et al. 2020].

O ambiente Android possui uma arquitetura em pilha contendo: aplicativos, APIs Java, bibliotecas nativas, camada de abstração de hardware e núcleo Linux. Essa arquitetura define mecanismos específicos para o funcionamento dos componentes no Android e, consequentemente, são utilizados mecanismos de detecção específicos para Android de modo a explorar essas diferenças. A segurança da plataforma Android se beneficia nos seguintes recursos [Android 2020]:

- Segurança robusta no nível do sistema operacional por meio do núcleo Linux, que permite que o Android faça uso de mecanismos de segurança do núcleo, como por exemplo o SELinux;
- Todos os aplicativos são executados isolados em *sandbox*;
- Comunicação segura entre processos;
- Assinatura de aplicativos, de modo a garantir a autenticidade dos mesmos;
- Uso de permissões definidas pelo aplicativo e concedidas pelo usuário.

Dentre os recursos de segurança do Android, o sistema de permissões e o conceito de *sandbox* implementam o princípio do privilégio mínimo para a execução das aplicações. Cada instância de um aplicativo é executada em uma máquina virtual *Android Runtime (ART)* isolada com um *user id* e *group id* únicos e que permitem acesso somente à pasta do próprio aplicativo. Com isso, uma aplicação só é capaz de solicitar serviços do dispositivo através dos mecanismos de comunicação entre processos do sistema Android [Rashidi and Fung 2015].

Além disso, apesar da alta capacidade de processamento que *smartphones* atuais possuem, ainda existem limitações de certos recursos, como a bateria. Devido a essas particularidades, o sistema Android precisa de mecanismos próprios para detecção de

malware. Três estratégias podem ser utilizadas para analisar os dados e utilizar um mecanismo de identificação, estas sendo: (I) análise estática, que visa analisar características inerentes à aplicação declaradas em seu código fonte; (II) dinâmica, que visa analisar o comportamento da aplicação em execução; e (III) híbrida, que consiste em combinar a análise estática e dinâmica [Kouliaridis et al. 2020].

2.2. 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 consiste de um arquivo compactado contendo 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` [Pan 2020].

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 cifrados, geração de código em tempo de execução, etc. Esse tipo de comportamento só pode ser detectado na análise dinâmica em tempo de execução [Nirumand et al. 2018].

Os algoritmos de análise estática podem ser baseados em: assinaturas, permissões e *bytecode* da máquina virtual. 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 ou o arquivo *.dex* para avaliar os fluxos de dados, chamadas de *Application Programming Interface (API)*, etc [Zachariah et al. 2017].

2.3. Análise dinâmica

A análise dinâmica consiste em monitorar e analisar o comportamento do aplicativo durante a sua execução para detectar alguma atividade suspeita e consequentemente se o mesmo é malicioso ou não. Um conjunto variado de atributos pode ser utilizado na análise dinâmica, como por exemplo: uso de recursos, comunicação com a Internet, chamadas de sistemas e comunicação entre aplicativos.

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 analisados, 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 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]. A emulação é um recurso explorado para monitorar a execução da aplicação fora do ambiente de execução, através do uso de *sandboxes*, mitigando assim o risco de um *malware* danificar o sistema antes de ser identificado.

3. Trabalhos relacionados

Esta seção apresenta os principais trabalhos encontrados na literatura, que propõem estratégias e técnicas de detecção de *malware* usando informações de comunicação entre processos no Android.

DroidCat [Cai et al. 2018] 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 *Software Development Kit (SDK)* por aplicações maliciosas e benignas: aplicações maliciosas utilizam *Intents* explicitas 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.

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.

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 Smali[2] coletados no servidor remoto onde o aplicativo é descompilado. O servidor remoto analisa em

[2]O formato *Android Application Package (APK)* com o arquivo *.dex* é o formato binário de uma apli-

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.

Droidiagnosis [de Souza Polisciuc et al. 2020] se trata de um detector de aplicações maliciosas com abordagem híbrida que utiliza permissões e chamadas de sistema como atributos. O trabalho explora a quantidade de chamadas de sistema em si, mas também informações contidas em seus argumentos como o acesso a diretórios, arquivos e URLs. Finalmente, ao analisar os resultados obtidos com o classificador o trabalho conclui que existe ganho de desempenho ao se utilizar o modelo híbrido; entretanto não são citados no trabalho a separação de bases de treino, validação e teste ou o uso de validação cruzada para impedir que os resultados comparados sejam tendenciosos ou afetados por *overfitting*.

4. Proposta

Nesta seção é apresentada uma proposta de detecção de aplicativos maliciosos baseada em identificadores utilizados na comunicação entre processos coletados de forma dinâmica e um conjunto de atributos estáticos contendo permissões, categorias, `class.package` e as versões máxima, mínima e alvo do SDK para o aplicativo analisado. Vale ressaltar que a presente proposta não se estende a ataques externos aos aplicativos, como *DNS spoofing*, ou a ataques em conluio; limitando-se assim à análise de aplicativos individualmente.

O método proposto consiste em um detector *offline*, onde a análise dos aplicativos é feita com dados previamente coletados e não em tempo real. Essa escolha permite que o método monitore cada aplicação de uma vez para a coleta dos atributos dinâmicos e não impõe limitações ao tempo de processamento dos dados e consumo de bateria do dispositivo, permitindo assim utilizar métodos mais custosos e robustos para analisar os mesmos. No Android, esse tipo de detecção é vantajosa pois pode ser realizada antes de se disponibilizar os aplicativos nas lojas ou até mesmo antes da instalação dos mesmos.

4.1. Atributos dinâmicos

De forma a caracterizar o comportamento dos aplicativos durante sua execução, foram utilizados identificadores de chamadas utilizadas durante a comunicação entre processos. Essa abordagem foi escolhida pois a arquitetura do Android força os aplicativos a utilizarem o seu mecanismo de comunicação entre processos para ter acesso a recursos do sistema e de *hardware*. Dessa forma, rastrear e entender as comunicações que utilizam esse canal é um ponto chave para entender comportamentos específicos do Android [Tam et al. 2015].

A comunicação entre processos na arquitetura do Android não é realizada diretamente, sendo intermediada pelo núcleo [Artenstein and Revivo 2014]. Dessa forma,

cação, que não é de fácil entendimento nem prático para efetuar modificações. O formato Smali vem para permitir o fácil entendimento destas informações.

como o Android possui um núcleo Linux, são utilizadas chamadas de sistema *ioctl* para comunicar com o mesmo, conforme mostra a Figura 1 que descreve o processo de solicitação de um serviço por um aplicativo.

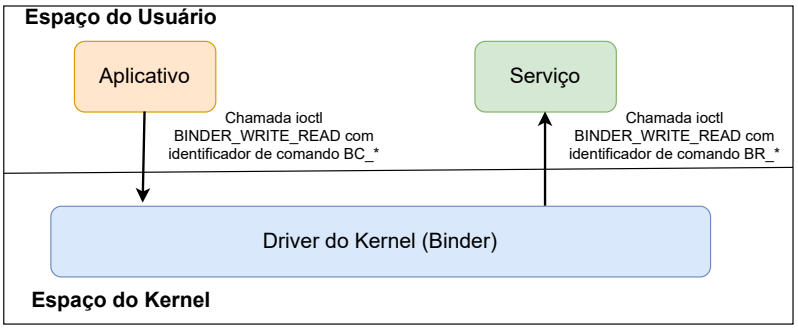

Figura 1. Mecanismo de comunicação entre processos no Android.

A chamada *ioctl* utilizada para fazer requisições tem como código de requisição `BINDER_WRITE_READ` e carrega parâmetros de escrita e leitura entre o espaço do usuário e o espaço do núcleo. Dentre esses parâmetros estão os identificadores de chamada, que começam com `BC_*` e `BR_*` e indicam os comandos da chamada *ioctl* enviados para o núcleo e recebidos pelo núcleo respectivamente.

Para serem utilizados como atributos para a detecção de *malware*, os traços de identificadores de chamada coletados durante a execução dos aplicativos analisados foram representados utilizando a abordagem *Bag-of-Words* [Zhang et al. 2010].

4.2. Atributos estáticos

Os atributos estáticos utilizados foram coletados utilizando o Androguard [Desnos et al. 2015], um conjunto de ferramentas utilizado para realizar engenharia reversa em aplicativos Android. Utilizando o Androguard foram coletadas informações contidas no arquivo *AndroidManifest.xml* dos aplicativos, em seu código fonte e referentes às autoridades certificadoras que assinam os certificados dos aplicativos.

Dentre as informações coletadas foram escolhidas como atributos para este trabalho as seguintes informações contidas no *AndroidManifest.xml*: permissões, categorias e versões mínima, alvo e máxima de SDK, e as classes contidas no código fonte do aplicativo. A Figura 2 apresenta esse processo de extração desses dados.

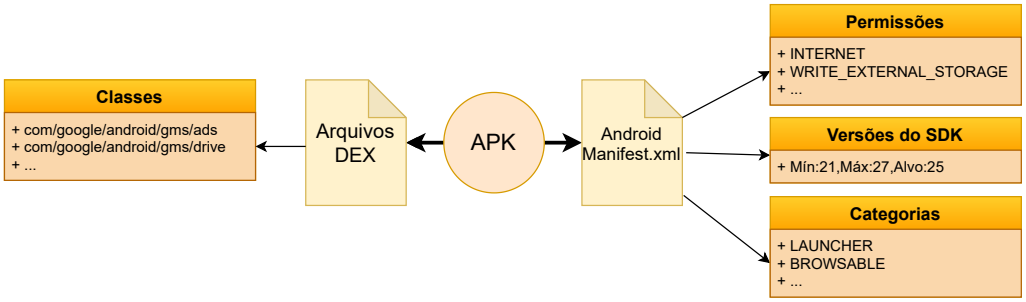

Figura 2. Processo de extração dos atributos estáticos.

No Android cada versão sucessiva da plataforma atualiza a lista de APIs disponíveis na versão anterior [Android 2021], e por isso a importância de se definir as versões compatíveis com o aplicativo analisado para analisar a sua segurança.

Os filtros de *Intents* são utilizados para definir os tipos de *Intents* que podem ser utilizados por outros aplicativos para iniciar uma atividade com o componente. As categorias são utilizadas para categorizar esses componentes, sendo assim uma *string* que contém informações adicionais sobre o tipo de componente que deve processar o *Intent* [Android 2019].

As permissões especificam as autorizações de acesso que precisam ser concedidas pelo usuário para que o aplicativo funcione. Finalmente, as classes coletadas incluem todas as classes chamadas no código fonte do aplicativo e logo contêm todas as APIs utilizadas no mesmo, identificando assim os recursos utilizados pelo desenvolvedor do aplicativo.

Por se tratarem de listas, os atributos referentes a permissões, categorias e classes foram vetorizados de forma a serem consumidos pelos algoritmos de aprendizado de máquina. Para isso, cada elemento dessas listas foi transformado em um atributo booleano que indica se o aplicativo em questão contém esse atributo ou não.

4.3. Metodologia proposta

O processo adotado para analisar um aplicativo consiste em simular a sua execução enquanto é monitorado para a extração das características dinâmicas e extrair as características estáticas do mesmo. Posteriormente, essas características são combinadas em um único vetor de características que é utilizado para classificar o aplicativo em malicioso ou não, de acordo com o modelo de aprendizagem de máquina treinado.

Para validar a proposta, foi utilizada a base de dados de aplicativos rotulada em maliciosos ou não Androzoo [Allix et al. 2016]. Essa base de dados foi selecionada por se tratar de uma base de dados pública e continuamente atualizada. Foram coletadas características de 622 aplicativos lançados entre 2018 e 2020, sendo 319 aplicativos benignos e 303 aplicativos maliciosos, se tratando portanto de uma base de dados balanceada.

Posteriormente essa base de dados foi normalizada e dividida na proporção de 80% pra treino/validação e 20% para testes. Os dados de treino foram utilizados para treino e validação dos modelos através de validação cruzada *k-fold*, com $k = 5$. Os modelos foram treinados utilizando seleção de atributos com objetivo de selecionar os 10% atributos mais relevantes e três algoritmos diferentes de aprendizagem de máquina: *Multilayer Perceptron*, *Random Forest* e *Ada Boost*. Estes algoritmos foram escolhidos devido à sua popularidade na literatura e seu baixo custo para o treinamento dos modelos de detecção. Finalmente, após o treino e validação os modelos foram testados com a base de testes, com objetivo de validar a proposta e garantir que não houve *overfitting* ou a inclusão de tendências nos mesmos. O fluxo descrito pode ser observado na Figura 3.

Para os testes foi utilizado o emulador do ambiente Android Studio em sua versão 3.6.3. O *Dispositivo Virtual Android (AVD)* 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. O Android 10.0 foi escolhido por ser a versão mais utilizada do Android no mundo desde

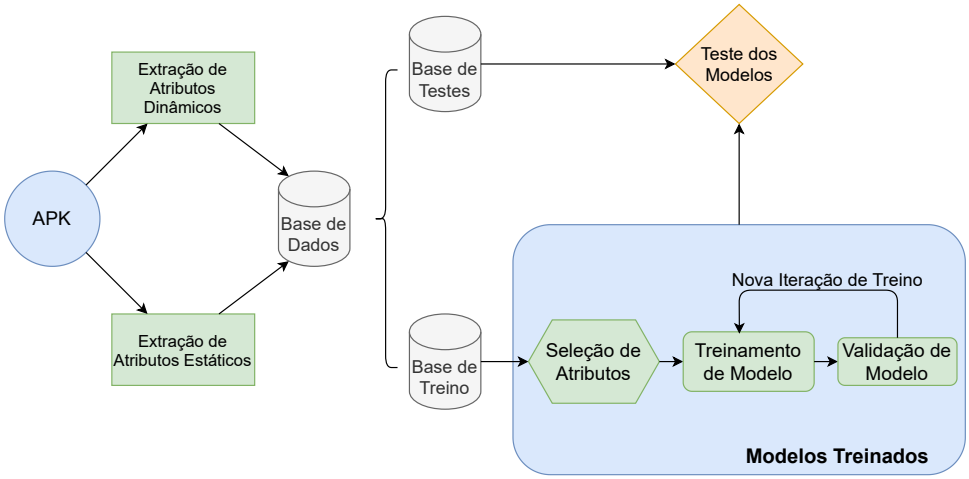

Figura 3. Fluxo de tarefas utilizado para validar a proposta.

agosto de 2020, com 41.17% do *market share* [StatCounter 2021].

5. Avaliação de Resultados

Ao dividir o conjunto de dados em bases de treino e teste, ambas as bases permaneceram com uma distribuição balanceada. A base de treino utilizada tem 257 aplicações benignas e 248 maliciosas, enquanto a base de testes tem 63 aplicações benignas e 64 maliciosas. Esse balanceamento nas bases é importante para não introduzir tendências nos algoritmos treinados.

A avaliação dos dados contidos na base de treino acabou utilizando três abordagens, com o enfoque na caracterização das diferenças entre aplicações maliciosas e benignas: (I) identificar atributos estáticos presentes somente em aplicações maliciosas; (II) atributos com a maior diferença entre as suas aparições em aplicativos maliciosos e benignos; e (III) identificar comportamentos característicos dessas aplicações representados nos dados dinâmicos.

A análise dos atributos estáticos contidos apenas em aplicações maliciosas foi realizada a partir da observação dos nomes das classes encontradas, concluindo que alguns dos aplicativos foram ofuscados. Este fator destaca a importância de utilizar um método que não dependa somente de atributos estáticos para detecção. Algumas das características relacionadas ao uso de permissões que se destacaram:

- Permissões com objetivo de alterar o funcionamento do serviço de localização (`BAIDU_LOCATION_SERVICE`, `LOCATION_MODE_HIGH_ACCURACY` e `LOCATION_MODE_OFF`); apesar de algumas dessas chamadas não serem compatíveis com versões recentes do Android, tendo assim versões desatualizadas do mesmo como alvo do atacante;
- `RUN_INSTRUMENTATION`, perigosa pois fornece ao aplicativo informações de instrumentação do aplicativo que supostamente estaria sendo testado e acesso para executar automaticamente ações no mesmo;
- `INTERACT_ACROSS_USERS`, que permite que o aplicativo execute ações transversalmente entre diferentes usuários, podendo assim violar a proteção entre usuários; e

- PERMISSION_RUN_TASKS, que permite que o aplicativo execute uma série de ações baseadas em contexto (hora, data, localização, gesto, entre outros).

Já na abordagem utilizada para atributos estáticos, foram observados os 30 atributos com a maior divergência de aparição entre aplicações maliciosas e benignas (isto é, os 30 atributos que possuem a maior discrepância entre aplicações maliciosas e benignas), em valores absolutos. Estes atributos foram selecionados dentre os 14.721 atributos estáticos coletados. O resultado é apresentado na Figura 4 destacando os respectivos atributos selecionados (eixo X) pela presença do mesmo em relação ao conjunto de aplicações (eixo Y).

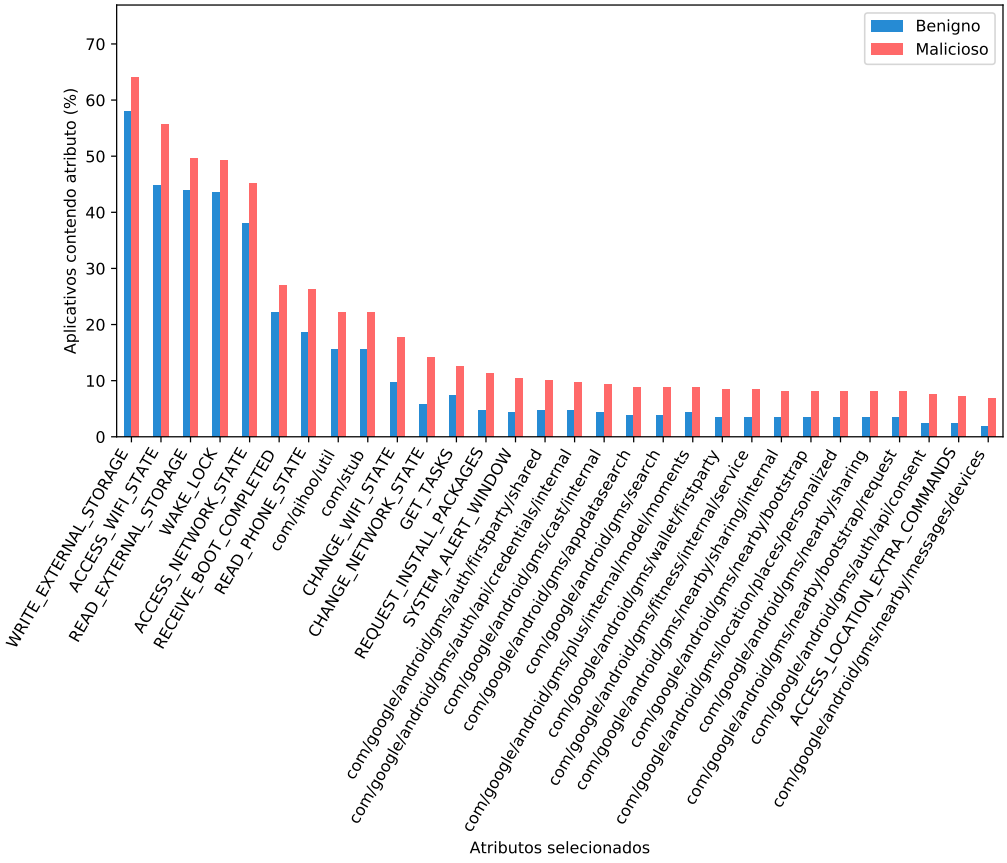

Figura 4. Diferença entre chamadas realizadas por aplicativos maliciosos e benignos.

O primeiro ponto que pode ser considerado consiste na diferença entre o número de permissões relacionadas a acesso e alteração de elementos de rede (ACCESS_WIFI_STATE, CHANGE_NETWORK_STATE, ACCESS_NETWORK_STATE, CHANGE_WIFI_STATE). Esta diferença entre aplicações benignas e maliciosas é explicada pela necessidade que aplicações maliciosas possuem de acessar a Internet, visando, por exemplo, requisitar anúncios e fazer *upload* de dados.

A permissão READ_PHONE_STATE, além de permitir acesso a informações de rede, permite acesso ao status de ligações e contas de usuários registradas no dispositivo.

Dessa forma, esse recurso permite monitorar ações do usuário do dispositivo e coletar informações privadas do mesmo, como quando e para quem o mesmo fez ligações.

Por fim, as permissões `ACCESS_LOCATION_EXTRA_COMMANDS` e `REQUEST_INSTALL_PACKAGES` têm destaque devido à sua relação com certas funcionalidades exercidas por aplicações maliciosas. `ACCESS_LOCATION_EXTRA_COMMANDS`, além de adquirir os dados encontrados com o `READ_PHONE_STATE` permite acesso da aplicação a recursos de localização. Já a permissão `REQUEST_INSTALL_PACKAGES` permite que a aplicação instale outras aplicações no dispositivo sem notificar o usuário nem solicitar a aprovação explícita dele.

Ao se analisar os identificadores de chamadas de sistema presentes na parte dinâmica da base de dados, observa-se que os aplicativos maliciosos realizam em média 17,4% mais chamadas por aplicativo do que os benignos, mostrando assim a diferença de comportamento entre as duas classes de aplicações. 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 Tabela 1 apresenta o desempenho médio para o primeiro conjunto de testes, que consiste na utilização da base de validação. Os valores médios representam as 5 execuções realizadas no processo de validação cruzada e destacam os valores de *F1Score*, *Recall*, *Precision*, *Brier Score* e *ROC AUC*. As métricas apresentadas foram escolhidas visando representar o desempenho da proposta.

Algoritmo	F1Score	Recall	Precision	Brier Score	ROC AUC
Random Forest	91,71%	90,17%	93,61%	7,92%	92,20%
AdaBoost	93,40%	91,29%	95,73%	6,33%	93,68%
Multilayer Perceptron	92,90%	93,70%	92,25%	6,93%	93,22%

Tabela 1. Desempenho médio na base de validação

Considerando os valores de *F1Score*, é possível afirmar que todos os três algoritmos apresentam uma boa adequação e têm uma alta taxa de identificação. Esses resultados demonstram que a estratégia e método de avaliação são adequados para a identificação de ameças no Android, além de demonstrar a adequação dos atributos escolhidos. Levando em consideração o valor alcançado pela *ROC AUC* e *Brier Score*, é possível observar o aprendizado e adequação dos dados em relação aos modelos, destacando esta estratégia para a identificação de ameaças.

Dentre as três abordagens, o *AdaBoost* é o algoritmo com o melhor resultado para o *F1Score* e com o menor valor para o *Brier Score*, que representa a taxa de perda e adequação ao modelo (quanto menor melhor). O *F1Score* e *Precision* são próximos para as três abordagens, evidenciando assim a qualidade dos dados utilizados para representar as classes de aplicativos analisadas.

A Figura 5 apresenta o resultado para a curva ROC, levando em consideração o melhor resultado obtido na base de teste, neste caso o *AdaBoost*. A curva demonstra que o seu desempenho se manteve consistente em todas etapas da validação cruzada, à medida em que o limiar de discriminação foi variado.

A Tabela 2 apresenta o desempenho médio utilizando os 20% selecionados para

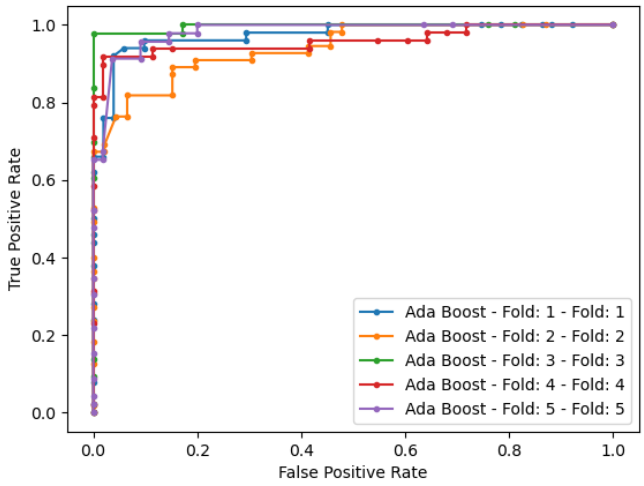

Figura 5. Curva ROC para o algoritmo *AdaBoost*.

a base de teste. É possível observar que apesar da queda no desempenho, os modelos treinados continuaram com um bom desempenho ao classificar amostras desconhecidas, mostrando assim a validade e resiliência do método adotado. A maior queda aconteceu na métrica de *Recall*, indicando um aumento no número de falsos negativos. Este fator é explicado pela possibilidade da base de dados utilizada para treinar os modelos de detecção não abranger todos os tipos de aplicações maliciosas existentes na base de testes.

Algoritmo	F1Score	Recall	Precision	Brier Score	ROC AUC
Random Forest	86,09%	76,90%	97,87%	13,85%	87,37%
AdaBoost	87,29%	77,46%	100,00%	12,59%	88,73%
Multilayer Perceptron	87,01%	81,97%	92,76%	13,70%	86,87%

Tabela 2. Desempenho médio na base de testes

Os experimentos demonstram a adequação dos modelos ao se utilizar esse conjunto de dados híbridos, validando assim a proposta deste trabalho. Os atributos selecionados são os principais fatores para este resultado, já que eles são responsáveis por refletir as informações dos aplicativos para os modelos de *Machine Learning (ML)*.

6. Conclusão

Devido à popularidade do Android, novas ameaças vêm sendo descobertas no decorrer dos anos. Para garantir a segurança dos usuários, novas estratégias estão sendo desenvolvidas visando otimizar a identificação de *malwares* no Android.

Visando a identificação de ameças no Android, este trabalho avaliou a adequação de uma estratégia com dados híbridos para identificar aplicações maliciosas no sistema. Dois conjuntos de atributos estão presentes no *dataset* AndroBlend, que representam características estáticas e dinâmicas de 622 APKs. Por fim, foi possível avaliar o quão adequado este conjunto de dados seria para representar aplicações benignas e malicio-

sas, utilizando três algoritmos de ML. Os resultados obtidos foram promissores, com um desempenho adequado para todas as estratégias testadas.

Como trabalho futuro, pode-se aprimorar o tratamento dos dados dinâmicos utilizados, de forma a representar de forma mais precisa o comportamento dos aplicativos analisados. Isso pode ser feito por exemplo ao se considerar os traços dos identificadores de chamada utilizados.

Agradecimentos

Este trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES). Os autores também agradecem o Programa de Pós-Graduação em Informática da UFPR.

Referências

Afonso, V. M., de Amorim, M. F., Ellery, E., Grégio, A. R., Junquera, G. B., Schick, G. A., Dahab, R., and de Geus, P. L. (2013). Um sistema para análise e detecção de aplicações maliciosas de android.

Ajiri, V., Butakov, S., and Zavarsky, P. (2020). Detection efficiency of static analyzers against obfuscated android malware. In *2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity)*.

Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. (2016). Androzoo: Collecting millions of android apps for the research community. In *13th International Conference on Mining Software Repositories*, MSR '16, New York, NY, USA. ACM.

Alzaylaee, M. K., Yerima, S. Y., and Sezer, S. (2020). DL-droid: Deep learning based android malware detection using real devices. *Computers & Security*, 89:101663.

Android, D. (2019). Intents e filtros de intents. `https://bit.ly/3gzwrTq`.

Android, D. (2020). Proteja um dispositivo android. `https://bit.ly/3wzmUSX`.

Android, D. (2021). Visão geral do manifesto do aplicativo. `https://bit.ly/35r1ifE`.

Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., and Rieck, K. (2014). Drebin: Effective and explainable detection of android malware in your pocket.

Arshad, S., Shah, M. A., Wahid, A., Mehmood, A., Song, H., and Yu, H. (2018). Samadroid: A novel 3-level hybrid malware detection model for android operating system. *IEEE Access*.

Artenstein, N. and Revivo, I. (2014). Man in the Binder: He Who Controls IPC, Controls the Droid. *BlackHat Europe 2014*.

Avira (2020). Americans are catching on. `https://bit.ly/3b4jJKm`.

Cai, H., Meng, N., Ryder, B., and Yao, D. (2018). Droidcat: Effective android malware detection and categorization via app-level profiling. *IEEE Transactions on Information Forensics and Security*, 14(6):1455–1470.

Chebyshev, V. (2021). Mobile malware evolution 2020.

CISA (2020). Alert (aa20-099a): COVID-19 exploited by malicious cyber actors.

de Souza Polisciuc, R., Albini, L. C., Grégio, A., and Bona, L. C. (2020). Análise de aplicativos no android utilizando traços de execução.

Desnos, A. et al. (2015). Androguard: Reverse engineering, malware and goodware analysis of android applications.

Fritz, C., Arzt, S., Rasthofer, S., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. (2013). Highly precise taint analysis for android applications.

Golrang, A., Yayilgan, S. Y., and Elezaj, O. (2021). The multi-objective feature selection in android malware detection system. In *Intelligent Technologies and Applications*. Springer International Publishing.

Hamed, Y. S. I., AbdulKader, S. N. A., and Mostafa, M. S. (2019). Mobile malware detection: A survey. *International Journal of Computer Science and Information Security*.

Kato, H., Haruta, S., and Sasase, I. (2020). Android malware detection scheme based on level of SSL server certificate. *IEICE Transactions on Information and Systems*.

Kouliaridis, V., Barmpatsalou, K., Kambourakis, G., and Chen, S. (2020). A survey on mobile malware detection techniques. *IEICE Transactions on Information and Systems*.

Kumar, S. and Shukla, S. K. (2020). The state of android security. In *Cyber Security in India*, pages 17–22. Springer Singapore.

Nirumand, A., Zamani, B., and 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.

Pan, B. (2020). Tools to work with android .dex and java .class files. `https://bit.ly/3pZOfv8`.

Qiu, J., Nepal, S., Luo, W., Pan, L., Tai, Y., Zhang, J., and Xiang, Y. (2019). Data-driven android malware intelligence: A survey. In *Machine Learning for Cyber Security*, pages 183–202. Springer International Publishing.

Rahman, M., Rahman, M., Carbunar, B., and Chau, D. H. (2017). Search rank fraud and malware detection in google play. *IEEE Transactions on Knowledge and Data Engineering*.

Rashidi, B. and Fung, C. J. (2015). A survey of android security threats and defenses. *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.*, 6(3):3–35.

Rogers, R., Lombardo, J., Mednieks, Z., and Meike, B. (2009). *Android application development: Programming with the Google SDK*. O'Reilly Media, Inc.

StatCounter (2021). *Mobile Android Version Market Share Worldwide*.

Tam, K., Khan, S. J., Fattori, A., and Cavallaro, L. (2015). CopperDroid: Automatic reconstruction of android malware behaviors. In *2015 Network and Distributed System Security Symposium*. Internet Society.

Zachariah, R., Akash, K., Yousef, M. S., and Chacko, A. M. (2017). Android malware detection a survey. In *2017 IEEE International Conference on Circuits and Systems*.

Zhang, Y., Jin, R., and Zhou, Z.-H. (2010). Understanding bag-of-words model: a statistical framework. *International Journal of Machine Learning and Cybernetics*.