Análise de Aplicativos no Android utilizando Traços de Execução

Renan Polisciuc[1], Luiz Albini[1], André Grégio[1], Luis Bona[1]

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

{rspolisciuc,luiz.albini,andre.gregio,luis.bona}@inf.ufpr.br

Abstract. *Android is the most used mobile operating system worldwide. This has increasingly been bringing more developers to the platform, as it is open-source and allows free app development. On the other hand, malicious apps have also been developed, which aim to harm the end user and are hard to identify, causing researchers to propose solutions to differentiate them from non-malicious apps. In this field of research, this paper presents the DroiDiagnosis, a proposal that uses machine learning and classifies* 80% *of the samples into non-malicious and malicious based their dynamic and static features.*

Resumo. *O Android é o sistema operacional mais utilizado por dispositivos móveis no mundo. Esse fato tem atraído cada vez mais desenvolvedores para a plataforma devido a sua característica opensource e desenvolvimento gratuito de aplicativos. Um problema que surgiu a partir disso são os aplicativos maliciosos, que visam prejudicar o usuário final e que muitas vezes são difíceis de identificar, o que tem levado autores a propor soluções para diferenciá-los dos benignos. Nesse sentido, neste trabalho será apresentado o DroiDiagnosis, uma solução que utiliza aprendizado de máquina e que classifica* 80% *das amostras entre benignas e maliciosas baseada em suas características dinâmicas e estáticas.*

1. Introdução

O Android é o sistema operacional para dispositivos móveis mais popular do mundo, representando uma parcela de mais de 85% do mercado de smartphones, [Statista 2018]. No Brasil, ele está presente em cerca de 95% dos smartphones, [Lavado 2019]. Essa popularidade tem atraído cada vez mais desenvolvedores para a plataforma, incluindo alguns mal-intencionados que desenvolvem aplicações maliciosas (*malware*) para roubar informações do usuário e/ou danificar o dispositivo. Alguns *malware* podem enviar SMS sem o consentimento do usuário, realizar chamadas telefônicas, roubar dados bancários ou até mesmo bloquear o dispositivo. Com base nisso, muitos trabalhos foram desenvolvidos para analisar aplicações do Android e identificar aquelas que são *malware*.

Trabalhos como Drebin, [Arp et al. 2014], DroidMat, [Wu et al. 2012], e DroidFusion, [Yerima and Sezer 2018] utilizam a técnica de análise estática, que consiste em descompilar a aplicação e analisar seu código fonte e recursos utilizados para encontrar características que indiquem comportamento malicioso. Para classificar uma aplicação como *malware*, eles utilizam algoritmos de aprendizado de máquina que têm lhes proporcionado taxas de acurácia de mais de 90%. Apesar de possuírem uma alta precisão e

serem mais rápidas que outras técnicas, elas têm um problema em comum que é não identificar *malware* que execute ações maliciosas em tempo de execução, já que sua análise é baseada apenas no código estático. Isso permite que muitos *malware* encriptem strings com comandos e os decriptem quando a aplicação está sendo executada, evadindo assim o *scan*.

A técnica de análise dinâmica é utilizada por trabalhos como EnDroid, [Feng et al. 2018], e DroidScribe, [Dash et al. 2016], e consiste em executar a aplicação para captura de ações maliciosas. A vantagem dessa abordagem é que ela identifica execução de código malicioso ofuscado pelo *malware*. Quanto às desvantagens, elas são mais lentas e complexas que as estáticas, pois exigem que uma VM *(virtual-machine)* seja levantada ou então um dispositivo real seja utilizado. Em ambos ambientes o dispositivo deve ser reiniciado e ter todos os seus dados removidos a cada simulação para evitar influências de uma simulação em outra. Uma desvantagem adicional de simulação em ambientes emulados é que alguns *malware* podem identificar que estão sendo executados nesses ambientes. Outra desvantagem comum da análise dinâmica é que, devido ao tempo limitado de simulação e a aleatoriedade das ações, algumas partes da aplicação podem não ser alcançadas.

Um problema de alguns trabalhos que realizam análise dinâmica é que eles utilizam apenas o rastreamento de ações dentro da *sandbox* das aplicações, enquanto que algumas atividades maliciosas podem ser realizadas fora desse espaço. Com base nisso, alguns trabalhos têm utilizado chamadas de sistemas, que são a forma de troca de informações entre as aplicações e o *kernel* e que podem ser facilmente acessadas através de ferramentas do próprio Linux. Trabalhos como [Burguera et al. 2011], [Jaiswal et al. 2018] e [Malik 2016] utilizam chamadas de sistema para diagnosticar e classificar aplicativos maliciosos. O problema principal é que eles se baseiam apenas na quantidade de chamadas de sistemas efetuadas pelas aplicações para determinar padrões de classificação, que, como mostrado neste trabalho, são insuficientes para determinar a maliciosidade de uma aplicação.

Sendo assim, neste trabalho será introduzido o DroiDiagnosis, uma ferramenta baseada em chamadas de sistemas e permissões de acesso para diagnosticar aplicações maliciosas. Serão apresentados resultados relacionados a quantidade de chamadas de sistema, diretórios acessados e URLs as quais houve trocas de informações pelas aplicações. Também será mostrado como identificar algumas operações ilegais através de chamadas de sistemas. Por fim serão utilizadas todas essas características para classificar as aplicações utilizando os classificadores de aprendizado de máquina SVM, KNN e Decision Tree.

O restante do artigo esta organizado da seguinte maneira. A Seção 2 apresenta os trabalhos relacionados. A Seção 3 os fundamentos. A Seção 4 a solução proposta. A Seção 5 apresenta os experimentos e os resultados obtidos. A Seção 6 e 7 concluem este artigo com as considerações finais e problemas conhecidos.

2. Trabalhos relacionados

Na literatura são encontrados trabalhos que propõem diagnóstico de *malware* utilizando chamadas de sistema em vários sistemas operacionais. Em [Asmitha and Vinod 2014] o autor introduz uma solução baseada em chamadas de sistema para detecção de *malware*

no Linux. Ele coletou 226 amostras de exemplares maliciosos e 442 benignos e utilizou aprendizado de máquina para classificação. Como características das amostras, o autor utilizou o Strace para coletar as chamadas de sistemas utilizadas por cada processo. As características consistiam de chamadas que só ocorriam em uma das classes e chamadas que ocorriam em ambas. Este trabalho obteve uma acurácia de 97% das amostras classificadas corretamente.

Em [Kolbitsch et al. 2009] o autor introduz um sistema para detecção de *malware* no sistema operacional Windows baseado em grafos de chamadas de sistema que se mostrou mais efetiva que as soluções tradicionais baseadas em assinatura. Para obtenção das chamadas de sistema, a solução utiliza o NTrace, que é um equivalente do Strace para o Windows. Uma limitação dessa estratégia é que uma mudança no algoritmo pode gerar um grafo diferente que não foi previsto pela solução. Em [Pham et al. 2019] é apresentada uma ferramenta para auxiliar na extração de características de programas no sistema operacional Mac OS X. A ferramenta permite que características estáticas e dinâmicas sejam extraídas. Ela analisa o comportamento das aplicações através de chamadas de sistema.

A utilização de chamadas de sistema para detecção de *malware* em dispositivos não se restringe apenas a programas utilizados por usuários finais. Em [Tran et al. 2017] o autor introduz um kit de ferramentas para detecção de atividade maliciosa em roteadores comerciais. A principal contribuição deste trabalho é uma *sandbox* que possibilita a extração de chamadas de sistemas executadas e detecção de vulnerabilidades. Para isso, ele utiliza as ferramentas Metasploit e o Strace.

Alguns trabalhos também utilizam as chamadas de sistema para detecção de *malware* no Android. Em [Ahsan-Ul-Haque et al. 2018], os autores introduzem um sistema para detecção de aplicativos maliciosos no Android baseado no modelo de Markov. A ideia chave deste trabalho é identificar as transições entre as chamadas de sistemas executadas e a probabilidade da ocorrência de uma chamada de sistema a partir de outra. A base de dados utilizada foi $1,215$ aplicações maliciosas e 319 benignas. Usando a técnica de K-Fold e o classificador Naive Bayes, a solução obteve 98% de acurácia na classificação das amostras.

Em [Jaiswal et al. 2018] o autor utiliza as chamadas de sistema para analisar o comportamento de jogos e suas versões maliciosas para o Android. A ideia foi contar as quantidades de chamadas de sistemas executadas e utilizá-las para diferenciar as aplicações benignas das maliciosas. A base de dados utilizada continha 20 aplicações benignas e 40 maliciosas. A conclusão apresentada pelo autor é que aplicações maliciosas executam mais chamadas de sistema que suas versões benignas.

Em [Bhatia and Kaushal 2017] o autor apresenta uma solução para classificação de aplicações no Android utilizando aprendizado de máquina. Como característica das amostras, o autor utiliza a quantidade de chamadas de sistema executadas por cada aplicação. A base de dados utilizada pelo trabalho era de 50 aplicações benignas e 50 maliciosas, sendo que 85 dessas 100 aplicações foram corretamente classificadas utilizando o método proposto.

Em [Hou et al. 2016] é apresentada uma solução para análise de *malware* no Android baseada nos grafos de interações entre chamadas de sistemas executadas por uma

aplicação. A ideia é que cada nó do grafo é uma chamada de sistema e cada aresta de um nó para outro significa que o nó de origem veio antes do nó de destino. As arestas contêm pesos, que indicam quantas vezes B ocorreu depois de A. A conclusão apresentada por este trabalho é que a estratégia apresentada é mais precisa que apenas contar as chamadas de sistema executadas. Usando K-fold e *Deep Learning*, em uma base com $3,000$ aplicações ($1,500$ benignas e $1,500$ maliciosas), o trabalho obteve precisão de $93,68\%$ das amostras classificadas corretamente.

Nos trabalhos baseados em chamadas de sistema encontrados na literatura, notou-se que muitos deles se atentam apenas à quantificação das chamadas e a relação entre elas, mas não à informação contida nas chamadas. Além disso, muitas das soluções que utilizam aprendizado de máquina utilizam bases de dados desbalanceadas ou pequenas, que podem gerar a falsa impressão de que o método proposto é efetivo. Acredita-se que as chamadas de sistemas, somente, não são suficientes para identificar que uma aplicação é maliciosa. Por isso, neste trabalho será proposto explorar mais a fundo as chamadas de sistema no Android e combiná-las com características estáticas. O objetivo é verificar o quão relevante essas características são para um modelo de aprendizado de máquina e que as chamadas de sistemas podem conter informações mais valiosas.

3. Fundamentos

Assim como outros tipos de sistemas operacionais, o Android possui aplicativos maliciosos e formas de identificá-los. Por ser um sistema aberto e de desenvolvimento gratuito, o Android atrai muitos desenvolvedores, incluindo aqueles mal-intencionados. Em contraste a isso, o fato de ser de código aberto possibilita que estruturas do sistema sejam estudadas pela comunidade para criação de soluções anti-*malware* através de técnicas já estudadas em outros sistemas operacionais. Sendo assim, nesta Seção serão apresentados os tipos de *malware* encontrados para o Android e as técnicas de análise e detecção encontradas na literatura.

3.1. Aplicativos maliciosos no Android

Os *malware* para Android são aplicações que aproveitam brechas da API para explorar falhas do sistema e realizar tarefas maliciosas. Os principais objetivos desse tipo de aplicação são roubar informações dos usuários ou danificar o dispositivo, sendo que alguns deles podem utilizar créditos telefônicos para enviar SMS e realizar chamadas, travar o *smartphone* com operações inúteis e até 'sequestrar' o dispositivo, como é o caso do *ransomware* [Kubovič 2018].

Apesar da evolução do Android e surgimento de novas técnicas de detecção cada vez mais precisas, os *malware* ainda estão em alta. Em dezembro de 2017, a Kaspersky identificou um *malware* (posteriormente chamado de Loapi) que esquenta tanto o dispositivo (fazendo operações de criptomoedas) que 'estufa' a bateria, [Markovskaya 2017]. A *Avast Threat Labs* identificou um *adware*, *malware* que mostra propagandas desnecessárias e em excesso, pré-instalado em dispositivos não certificados pela Google como ZTE e Archos [Dent 2018]. A *Researchers at ThreatFabric* descobriu um trojan para Android 7 e 8 nomeado MysteryBot que se disfarça de app bancária para realizar ações maliciosas como chamadas telefônicas, ler contatos da lista telefônica e encriptar arquivos pessoais do usuário [O'Donnell 2018].

Com os passar dos anos, os ataques a dispositivos Android foram se diversificando conforme novas falhas foram sendo descobertas. A categoria de um *malware* é definida de acordo com o tipo de ação maliciosa que ele efetua. As categorias de *malware* conhecidas são: *trojan*, *backdoor*, *worm*, *botnet*, *spyware*, *adware* e *ransomware*.

4. Proposta

Nesta Seção será introduzido o DroiDiagnosis, uma solução híbrida para diagnóstico de aplicações maliciosas no Android. A análise híbrida é uma junção da estática e da dinâmica que visa melhorar a precisão em relação às duas individualmente. Alguns trabalhos como [Arshad et al. 2018], [Lindorfer et al. 2014] e [Mas'ud et al. 2013] utilizam a análise híbrida como solução, mas deixam em aberto algumas questões como: a solução híbrida é realmente melhor que as outras duas soluções individualmente? Algumas outras adicionam *overhead* demasiado, como análise em três níveis (estática, dinâmica e rede) em que uma etapa é executada após a outra.

Com relação aos trabalhos relacionados, este trabalho se diferencia por propor a análise híbrida utilizando chamadas de sistemas e permissões, que são características simples de serem extraídas. As principais contribuições do DroidDiagnosis são:

- Utiliza características simples de serem extraídas, o que implica em menor *overhead*
- Explora elementos das chamadas de sistemas não explorados por outros trabalhos
- Diagnóstico a partir de traços totais e parciais de execução

Assim como muitos dos trabalhos da literatura optamos por utilizar aprendizado de máquina pois esse método tem se mostrado eficiente na classificação de aplicações maliciosas no Android, além de permitir relacionar características estáticas e dinâmicas e por ser mais flexível para mudanças e melhorias futuras.

4.1. Obtenção de Características Estáticas

O processo de obtenção de características estáticas ocorre em paralelo ao de características dinâmicas e utiliza a ferramenta Androguard, [Anthony Desnos 2018], para extrair os metadados da aplicação a ser analisada. O Androguard é uma biblioteca do Python que realiza engenharia reversa em arquivos de aplicações do Android e possibilita que diversas informações sejam extraídas, como dados do manifesto, *strings*, classes, métodos, etc. Foi criado um serviço cuja entrada é o *sha256* do arquivo da aplicação que consulta em um repositório de arquivos .apk para poder enviar para o Androguard. Utiliza-se o *sha256* para comprovar que a aplicação não foi alterada de alguma forma e para catalogar as aplicações mais facilmente.

As características estáticas extraídas consistem das quantidades de ocorrências de cada permissão e quantidades de permissões por tipo de permissão (perigosas, normais ou assinadas) utilizadas pela aplicação. Permissões normais são automaticamente garantidas pelo sistema por não oferecerem perigo ao usuário e permissões assinadas são utilizadas apenas por aplicativos do sistema. As permissões escolhidas como característica são as definidas pelo Android como perigosas, que são aquelas que podem afetar a privacidade do usuário e/ou comprometer o funcionamento do dispositivo. Com relação aos tipos de permissões, são consideradas todas as permissões requisitadas pelo dispositivo, sendo elas perigosas ou não.

As permissões perigosas escolhidas foram: READ_CALENDAR, WRITE_CALENDAR, CAMERA, READ_CONTACTS, WRITE_CONTACTS, GET_ACCOUNTS, ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION, RECORD_AUDIO, READ_PHONE_STATE, CALL_PHONE, READ_CALL_LOG, WRITE_CALL_LOG, ADD_VOICEMAIL, USE_SIP, PROCESS_OUTGOING_CALLS, BODY_SENSORS, SEND_SMS, RECEIVE_SMS, READ_SMS, RECEIVE_WAP_PUSH, RECEIVE_MMS, READ_EXTERNAL_STORAGE e WRITE_EXTERNAL_STORAGE.

4.2. Obtenção de Características Dinâmicas

O processo de extração de características dinâmicas se inicia com o recebimento do $sha256$ pelo serviço de extração. O serviço se encarrega de verificar a existência do arquivo .apk da aplicação no repositório e iniciar uma instância do emulador do Android, [AndroidStudio 2019], para simulação da execução. Após n segundos de simulação, o serviço é finalizado e o processo que o iniciou copia o arquivo do traço de execução, que está em uma pasta interna do dispositivo, para pasta onde ele será processado, fora do dispositivo, onde ele é convertido para JSON e rearmazenado. Além disso, é gerado um arquivo .pcap através do TCPDump que realiza a tarefa de monitorar e extrair os dados de redes que entraram e saíram do dispositivo durante a simulação.

A simulação da execução da aplicação consiste de uma sequência de passos desde preparar o ambiente até gerar comandos pseudo-aleatórios para o dispositivo. 1) Após a aplicação ser encontrada no repositório, utilizamos a ferramenta AAPT para obter o ponto de entrada da aplicação (o seu "main"); 2) Inicia-se uma instância do emulador do Android sem interface gráfica e sem nenhum aplicativo não-padrão em execução, além de associar o TCPDump para capturar os pacotes de rede; 3) Através da ferramenta ADB, [ADB 2019], a aplicação é instalada no emulador; 4) Após isso, a aplicação é iniciada utilizando a informação obtida em 1 sem executar nenhuma ação adicional; 5) Obtém-se o *process id* (PID) pertencente à aplicação; 6) Inicia-se o Strace e atrelamos ao PID da aplicação; 7) Inicia-se uma *thread* para geração de toques aleatórios na tela do dispositivo simulando a utilização do mesmo com a ferramenta Monkey; 8) Inicia-se mais uma *thread* para enviar comandos Telnet para o dispositivo para simular o envio/recebimento de mensagens de texto (SMS), envio/recebimento de chamadas telefônicas e rotações do dispositivo.

Muitos dos trabalhos na literatura simulam apenas ações dos usuários com toques na tela com o Monkey, mas alguns tipos de *malware* podem realizar ações maliciosas com a ocorrência de eventos paralelos à aplicação, como o recebimento de SMS. Com base nisso, utilizamos a ferramenta ADB para abrir um *socket* no dispositivo e o Telnet para conectar e enviar comandos simulando a ocorrência de eventos.

Após a obtenção dos traços de execução são extraídas características como: quantas vezes que uma chamada de sistema ocorreu e qual foi essa chamada, quais chamadas ocorreram, ocorrências de domínios de sites com propaganda, principais diretórios e arquivos acessados, assim como ocorrência de chamadas da função su e comandos suspeitos, como $chmod$ e $chown$ (esses comandos implicam em mais chamadas de sistemas em processos iniciados a partir da aplicação, mas que não são observados).

4.3. Seleção de Características

O processo de seleção de características é executado previamente, de maneira que durante o processo de análise ele seja apenas consultado para filtrar as características. Ele consiste em eliminar chamadas de sistemas que foram identificadas como comuns e que não servem para diferenciar aplicações benignas de maliciosas. Assim como as chamadas de sistemas, usando o mesmo critério, também são removidas permissões que ocorrem muitas vezes tanto em uma classe quanto em outra. Diretórios exclusivos das aplicações, como pastas internas e arquivos exclusivos também são removidos. URLs que não são maliciosas também são removidas, sendo que nesse caso, assumimos que as URLs maliciosas são aquelas encontradas em *blacklists* na internet, ou que são conhecidas por possuir propagandas (*adware*).

4.4. Vetorização

A vetorização consiste em analisar as características da aplicação e criar um vetor V de números. Em V, as características estáticas são as primeiras posições, sendo que cada posição pode ser 0, quando a característica não ocorre na aplicação e 1 quando ocorre. No caso das características dinâmicas, elas são as últimas posições do vetor, sendo que cada posição é a quantidade de vezes que uma certa chamada de sistema ocorreu, a ocorrência ou não de chamadas maliciosas, representadas por 0 ou 1, diretórios restritos, representados por 0 ou 1 quando acessados e a quantidade de vezes que cada diretório foi acessado. A última posição do vetor é a quantidade de *URLs* maliciosas acessadas.

4.5. Classificação

A classificação consiste em um processo de aprendizado de máquina em que os vetores V de amostras conhecidas e corretamente classificadas utilizando a ferramenta VirusTotal [VirusTotal 2018] são utilizadas como modelo para classificação de novas amostras. Neste trabalho utiliza-se três classificadores: *SVM*, [RAY 2017], *KNN*, [SRIVASTAVA 2018], e *Decision-Tree*, [Gupta 2017], pois foram os mais utilizados pelos trabalhos encontrados na literatura. Os classificadores foram treinados para identificar duas classes: benigna e maliciosa. A cada chamada do classificador, ele é retreinado e a partir daí ocorre a classificação.

4.6. Processo de Análise

A análise é composta por uma união dos serviços apresentados nas Seções 4.1 e 4.2. Combinados, os dois serviços são utilizados para a geração de um vetor de características que representa a aplicação sob análise. Após a geração do vetor de características completo, ele é enviado para um processo de classificação que decide se a aplicação é benigna ou não. A Figura 1 apresenta a arquitetura do funcionamento do DroiDiagnosis.

5. Experimentos e Resultados

Nesta Seção serão apresentados o experimento e resultados obtidos pela ferramenta. Será explicado o experimento realizado e mostradas as características extraídas das amostras durante a simulação, além das características extraídas de maneira estática.

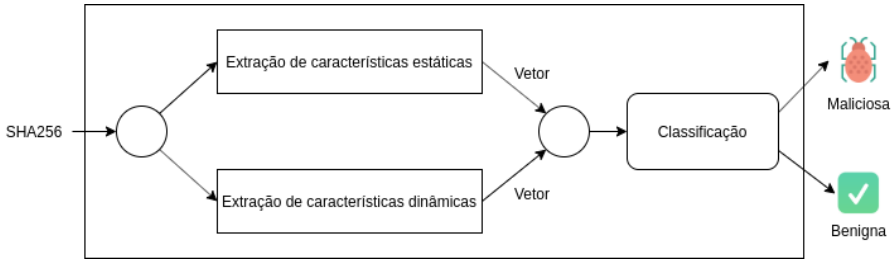

Figura 1. Modo de funcionamento de análise

5.1. Base de dados

Neste trabalho a base de dados escolhida foi a Androzoo, [Allix et al. 2016]. Esta base de dados é composta por $13,186,344$ amostras de aplicativos do Android divididas entre benignas e maliciosas. As aplicações foram escolhidas a partir de rótulos que identificam qual seu padrão de comportamento, como por exemplo *ransomware* ou *adware*. Foram escolhidas 100 aplicações publicadas entre 2015 e 2019 de 20 padrões diferentes, porém algumas delas falharam ao executar devido a problemas com a versão do Android e por problemas desconhecidos, sendo que apenas $1,431$ das aplicações maliciosas executaram efetivamente. Assim, para ter uma base balanceada, decidimos por igualar o número de aplicações por classe, ficando com $1,431$ aplicações maliciosas e $1,431$ benignas.

5.2. Experimentos

Foi feito um único experimento com todas as aplicações descritas na Seção 5.1. A máquina utilizada para as simulações possui uma CPU Intel(R) Core(TM) i5-7400 CPU @ $3,00$GHz com 4 núcleos, 8 GB de memória RAM de $2,133$ MHz. As aplicações foram executadas por um período de 20 segundos cada. Também foram coletados dados estáticos para análise de padrões. A versão do Android utilizada nas simulações foi a 7.0 (API 24) e arquitetura utiliza é a Intel X86_64.

5.3. Quantidade de Chamadas de sistema

Verificou-se que a quantidade de chamadas de *clock_gettime*, *gettimeofday* e *sched_yield* foram maiores para a classe dos *malware*. As funções *clock_gettime* e *gettimeofday* são relativas a tempo, enquanto que função *sched_yield* é utilizada para mover a *thread* para o fim da fila de execução da CPU. Um ponto que pode ser destacado dos *goodware* é a chamada da função *madvise* que é utilizada para obter informações de memória junto ao *kernel* (não é controlado pelo processo). Acredita-se que esse valor tenha sido maior para os *goodware* pois eles possuem uma execução mais linear e estável que os *malware*, que em muitos casos apresentaram erros após um certo tempo de execução (muitos terminaram antes dos 20s).

Pode-se perceber que os *malware* executaram mais chamadas de *readlinkat* e *exit* que os *goodware*. A função *readlinkat* tem a mesma função que *readlink*, e é utilizada para ler links simbólicos. A função *exit* é utilizada para finalizar um processo. Acredita-se que há uma relação entre essas duas chamadas, pois alguns *malware* costumam finalizar o processo quando há uma falha na leitura de um arquivo ou diretório.

Pode-se verificar também que os *malware* efetuam mais chamadas de *umask*, *rt_sigreturn*, *getrusage*, *getppid*, *getgid32*, *fnctl*, *exit_group*, *execve* e *brk*. A função *umask*

é utilizada para modificar a permissão de arquivos, *rt_sigreturn* é o retorno do *signal handler*, *getrusage* é utilizada para obter a utilização de um recurso, *getppid* obtém o PID do processo, *getgid32* obtém a identidade de um grupo, *fnctl* manipula *file descriptors*, *exit_group* finaliza todas as *threads* de um processo, *execve* executa um processo e *brk* altera a posição do *program break*. De maneira geral, a conclusão que se chegou é que os *malware* manipulam mais arquivos e buscam mais dados do processo que os *goodware*.

Uma característica pouco explorada por outros trabalhos é ocorrência exclusiva de algumas chamadas de sistemas. A não ocorrência de chamadas de sistemas em uma das classes pode indicar que elas podem ser utilizadas para diferenciá-las. As chamadas de sistema exclusivas no conjunto de aplicações estudadas são:

- **rt_sigsuspend**: espera por um sinal (apenas em *malware*)
- **chdir**: muda de diretório (apenas em *malware*)
- **pselect6**: monitora múltiplos *file descriptors* (apenas em *malware*)
- **mknodat**: cria um *node* em um diretório (apenas em *goodware*)
- **socket**: cria um ponto de comunicação (apenas em *goodware*)
- **fsync**: sincroniza dados em *buffers* com o dispositivo de memória permanente (apenas em *goodware*)

5.4. Diretórios e arquivos

Para diagnóstico de diretórios e arquivos acessados pelas aplicações, utilizou-se os parâmetros de algumas chamadas de sistemas. Os arquivos e pastas são obtidos através da análise de parâmetros das chamadas **unlinkat**, **fchmodat**, **faccessat** e **newfstatat**. Para facilitar a análise, foram divididos os arquivos/diretórios de acordo com a profundidade n, sendo que o n é definido como a **quantidade de diretórios entre a pasta principal do sistema e o arquivo/diretório acessado**.

Neste trabalho focou-se nos arquivos/diretórios de profundidade 0, 1 e 2. Como as aplicações acessam muitos arquivos, optamos por apresentar apenas os 10 diretórios mais acessados por cada classe de aplicação em cada profundidade.

Para a profundidade 0, a maioria dos diretórios acessados pelos *goodware* e pelos *malware* são os mesmos. No entanto, diretórios como **sdcard**, **sys** e **vendor** são mais acessados por *malware* do que por *goodware*. O diretório *sdcard* é mapeado para o cartão de memória do dispositivo, enquanto que os diretórios **sys** e **vendor** têm permissão permitida apenas para o usuário **root** do sistema.

Para profundidade 1, notou-se que a média de acessos ao diretório **storage/emulated** é maior para os *malware* do que para os *goodware*. Esse diretório representa a memória interna do dispositivo, o que indica que em média, aplicações maliciosas acessam mais a memória interna do que aplicações benignas.

Para a profundidade 2, não notou-se muitas diferenças entre os diretórios mais acessados por *goodware* e *malware*, porém verificou-se uma semelhança na quantidade média de acessos a alguns diretórios, como por exemplo **data/user/0**, **storage/emulated/0**, **data/misc/user** e **system/etc/security**. A média de acessos a esses diretórios pode indicar um padrão de acesso de aplicações no Android. Além disso, assim como nos diretórios de profundidade 1, pode-se notar que, em diretórios de profundidade 2, novamente os *malware* acessam mais a memória interna do que os *goodware*.

5.5. Permissões

Ambas classes de aplicações requisitam muitas permissões de acesso à rede, dados do dispositivo e memória externa. Algumas permissões requisitadas por *malware* podem ser destacadas, como: **GET_TASKS**, que garante acesso ao controle de *tasks* do Android, **CHANGE_WIFI_STATE**, que permite alterar o estado da conexão *Wifi*, **MOUNT_UNMOUNT_FILESYSTEMS**, que permite montar/desmontar arquivos de sistema para armazenamento removível e **READ_LOGS**, que permite ler *logs* do sistema. Observou-se também que as aplicações benigas requisitam tantas permissões perigosas quanto as aplicações maliciosas, no entanto a quantidade de permissões normais é maior nas aplicações benignas que nas maliciosas. Outro ponto é que permissões assinadas (*signature*) dificilmente são utilizadas por qualquer uma das classes.

5.6. Análise

Nesta seção serão avaliados os desempenhos das análises estática, dinâmica e híbrida através das métricas de acurácia, precisão, *recall* e *f1-score*. Utilizando apenas características estáticas, como permissões e tipos de permissões, obtiveram-se resultados que mostram que o classificador *SVM* é o que produz a maior precisão, com $74,7\%$, como mostrado na Tabela 1.

Classificador	Acurácia	Precisão	*Recall*	*F1-Score*
SVM	**0.738**	**0.747**	**0.738**	**0.736**
KNN	0.682	0.683	0.682	0.682
Decision Tree	0.662	0.662	0.662	0.662

Tabela 1. Métricas de desempenho da análise estática

Utilizando apenas características dinâmicas, como chamadas de sistemas, diretórios acessados e URLS acessadas, obtiveram-se resultados que mostram que o classificador *SVM* é o que produz a maior precisão, com 71.1%, como mostrado na Tabela 2.

Classificador	Acurácia	Precisão	*Recall*	*F1-Score*
SVM	**0.701**	**0.711**	**0.701**	**0.697**
KNN	0.683	0.684	0.683	0.683
Decision Tree	0.648	0.649	0.648	0.648

Tabela 2. Métricas de desempenho da análise dinâmica

Utilizando as características estáticas e dinâmicas obtiveram-se resultados que mostram que o classificador *SVM* é o que produz a maior precisão, com 80.0%, como mostrado na Tabela 3.

Classificador	Acurácia	Precisão	*Recall*	*F1-Score*
SVM	**0.800**	**0.800**	**0.800**	**0.800**
KNN	0.763	0.764	0.763	0.763
Decision Tree	0.673	0.673	0.673	0.673

Tabela 3. Métricas de desempenho da análise híbrida

6. Conclusão

Neste trabalho foi apresentado uma ferramenta baseada no rastreamento de chamadas de sistemas para identificar e diagnosticar aplicações maliciosas no Android. Pelo fato de o Android ser baseado no Linux, utilizou-se a ferramenta **strace** como base para a implementação.

Ao contrário de outros trabalhos, não focou-se apenas na quantidade de chamadas de sistema efetuadas pelas aplicações, pois supõe-se que essa métrica não garante a maliciosidade de uma aplicação. No entanto, verificou-se, através dos experimentos, que algumas chamadas ocorrem em média mais vezes nos *malware* do que nos *goodware*. Além disso, verificou-se que algumas chamadas de sistema ocorreram em uma classe e na outra não.

Além das chamadas de sistema, também verificou-se as URLs acessadas pelas aplicações assim como os diretórios. Através da análise dos diretórios, identificou-se que as aplicações maliciosas e não maliciosas tendem a acessar em média os mesmos diretórios, mas os *malware* acessam diretórios com acesso restrito mais vezes que os *goodware*. Identificou-se também alguns diretórios que são acessados de maneira padrão por aplicações do Android, como *data*, *system*, *storage* e *dev*. Foram detectadas também várias tentativas de utilização do comando **su** por aplicações maliciosas. Com relação às URLs acessadas, verificou-se que tanto os *malware* quanto os *goodware* mostram propagandas.

Em adição à análise das chamadas de sistemas, realizou-se engenharia-reversa no código das aplicações para verificar algumas outras características relevantes, em especial as permissões declaradas no manifesto. Verificou-se que tanto aplicações maliciosas quanto benignas tendem a utilizar permissões perigosas, no entanto aplicações benignas utilizam mais permissões ditas normais do que as maliciosas. Além disso, verificou-se que alguma permissões, como GET_TASKS, CHANGE_WIFI_STATE, MOUNT_UNMOUNT_FILESYSTEMS e READ_LOGS costumam ser mais utilizadas pelos *malware*.

Foram utilizados os dados obtidos nos experimentos para identificação de padrões de *malware* e *goodware* e classificação através de aprendizado de máquina. Foram criados três modelos baseados em características de análise dinâmica, estática e híbrida, uma combinação das duas. Utilizou-se os classificadores SVM, KNN e *Decision Tree*, sendo que o SVM foi melhor em todos os testes. Verificou-se também que o modelo estático se saiu melhor que o modelo dinâmico, sendo que o estático obteve o máximo de 73% de amostras classificadas corretamente, enquanto que o dinâmico obteve 70%. Melhor do que os dois, o modelo híbrido classificou 80% das amostras corretamente.

7. Problemas conhecidos

Algumas vulnerabilidades encontradas durante o desenvolvimento deste trabalho:

- Algumas aplicações podem executar atividade quando ocorre o *boot* do sistema. Nesse caso, nossa solução não detecta essa atividade maliciosa em específico pois assumimos que o *boot* já foi realizado
- Como citado por [Denney et al. 2018] alguns tipos de *rootkits* não utilizam chamadas de sistemas.

- Por utilizar o simulador, o DroiDiagnosis pode não identificar *malware* que detecta que está sendo executado em ambiente simulado e não efetua atividades maliciosas
- Ao utilizar a ferramenta Monkey para simular o usuário, algumas funcionalidades da aplicação podem não ser exploradas pois não foram acionadas

Em trabalhos futuros pretende-se contornar os problemas identificados e efetuar testes com bases de dados maiores que a utilizada.

Referências

ADB (2019). Android debug bridge. `https://developer.android.com`. Acessado em 05/01/2019.

Ahsan-Ul-Haque, A. S. M., Hossain, M. S., and Atiquzzaman, M. (2018). Sequencing system calls for effective malware detection in android. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7.

Allix, K., Bissyandé, T. F., Klein, J., and Traon, Y. L. (2016). Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471.

AndroidStudio (2019). Run apps on the android emulator. `https://developer.android.com/studio/run/emulator`. Acessado em 06/11/2019.

Anthony Desnos, Geoffroy Gueguen, S. B. (2018). Androguard. `https://androguard.readthedocs.io/en/latest/`. Acessado em 10/01/2020.

Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., and Rieck, K. (2014). Drebin: Effective and explainable detection of android malware in your pocket. In *Symposium on Network and Distributed System Security (NDSS)*.

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*, 6:4321–4339.

Asmitha, K. A. and Vinod, P. (2014). A machine learning approach for linux malware detection. In *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, pages 825–830.

Bhatia, T. and Kaushal, R. (2017). Malware detection in android based on dynamic analysis. In *2017 International Conference on Cyber Security And Protection Of Digital Services (Cyber Security)*, pages 1–6.

Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. (2011). Crowdroid: Behavior-based malware detection system for android. pages 15–26.

Dash, S. K., Suarez-Tangil, G., Khan, S., Tam, K., Ahmadi, M., Kinder, J., and Cavallaro, L. (2016). Droidscribe: Classifying android malware based on runtime behavior. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 252–261.

Denney, K., Kaygusuz, C., and Zuluaga, J. (2018). A survey of malware detection using system call tracing techniques.

Dent, S. (2018). Report finds android malware pre-installed on hundreds of phones. `https://www.engadget.com`. Acessado em 01/08/2018.

Feng, P., Ma, J., Sun, C., Xu, X., and Ma, Y. (2018). A novel dynamic android malware detection system with ensemble learning. *IEEE Access*, 6:30996–31011.

Gupta, P. (2017). Decision trees in machine learning. https://towardsdatascience.com/decision-trees-in-machine-learning-641b9c4e8052. Acessado em 09/09/2018.

Hou, S., Saas, A., Chen, L., and Ye, Y. (2016). Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs. In *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*, pages 104–111.

Jaiswal, M., Malik, Y., and Jaafar, F. (2018). Android gaming malware detection using system call analysis. In *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*, pages 1–5.

Kolbitsch, C., Comparetti, P., Kruegel, C., Kirda, E., Zhou, X.-y., and Wang, X. (2009). Effective and efficient malware detection at the end host. pages 351–366.

Kubovič, O. (2018). Ransomware para android em 2017: Novas infiltrações e extorsões mais graves. https://www.welivesecurity.com/br/2018/02/16/ransomware-para-android-em-2017/. Acessado em 20/08/2018.

Lavado, T. (2019). Em 10 anos no brasil, android foi de 2 smartphones para sistema operacional dominante do mercado. https://g1.globo.com. Acessado em 08/02/2020.

Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., F, Y., v. d. Veen, V., and Platzer, C. (2014). Andrubis – 1,000,000 apps later: A view on current android malware behaviors. In *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, pages 3–17.

Malik, S. (2016). System call analysis of android malware families. *Indian Journal of Science and Technology*, 9.

Markovskaya, A. (2017). Loapi — this trojan is hot! https://www.kaspersky.com/blog/loapi-trojan/20510/. Acessado em 01/09/2018.

Mas'ud, M. Z., Sahib, S., Abdollah, M. F., Selamat, S. R., Yusof, R., and Ahmad, R. (2013). Profiling mobile malware behaviour through hybrid malware analysis approach. In *2013 9th International Conference on Information Assurance and Security (IAS)*, pages 78–84.

O'Donnell, L. (2018). New banking trojan can launch overlay attacks on latest android versions. https://threatpost.com. Acessado em 02/08/2018.

Pham, D.-P., Vu, D.-L., and Massacci, F. (2019). Mac-a-mal: macos malware analysis framework resistant to anti evasion techniques. *Journal of Computer Virology and Hacking Techniques*, pages 1–9.

RAY, S. (2017). Understanding support vector machine algorithm from examples (along with code). https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/. Acessado em 08/09/2018.

SRIVASTAVA, T. (2018). Introduction to k-nearest neighbors: Simplified (with implementation in python). https://www.analyticsvidhya.com. Acessado em 08/09/2018.

Statista (2018). Global market share held by smartphone operating systems from 2009 to 2017. https://www.statista.com. Acessado em 02/10/2018.

Tran, N., Nguyen, N., Ngo, Q., and Le, V. (2017). Towards malware detection in routers with c500-toolkit. In *2017 5th International Conference on Information and Communication Technology (ICoIC7)*, pages 1–5.

VirusTotal (2018). Virustotal. https://www.virustotal.com/. Acessado em 01/09/2018.

Wu, D. J., Mao, C. H., Wei, T. E., Lee, H. M., and Wu, K. P. (2012). Droidmat: Android malware detection through manifest and api calls tracing. In *2012 Seventh Asia Joint Conference on Information Security*, pages 62–69.

Yerima, S. Y. and Sezer, S. (2018). Droidfusion: A novel multilevel classifier fusion approach for android malware detection. *IEEE Transactions on Cybernetics*, pages 1–14.