Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Adaptando JavaMOP para projetos Android

Gabriel dos Santos Martins

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2022

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Adaptando JavaMOP para projetos Android

Gabriel dos Santos Martins

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador)
CIC/UnB

Francisco Handrick Da Costa
CIC/UnB

Pedro Henrique Teixeira Costa
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
Coordenadora do Bacharelado em Ciência da Computação

Brasília, 07 de outubro de 2022

Dedicatória

Dedico esse trabalho aos meus pais que me proporcionaram o melhor para conseguir me dedicar aos estudos e sempre me apoiaram nos momentos mais difíceis dessa jornada. Dedico a minha noiva por sempre me apoiar, dar força e motivação para seguir em frente.

Agradecimentos

Agradeço primeiramente ao meu orientador, Professor Rodrigo Bonifácio que sempre me ajudou e me apoiou ao longo de todo o processo da construção desse Trabalho de Graduação. Agradeço ao Handrick pelas explicações e por me introduzir ao tema do presente trabalho e agradeço também ao Pedro que na fase final me ajudou na finalização do projeto apresentado aqui.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

O presente trabalho visa apresentar a implementação de um Sandbox para a plataforma Android utilizando JavaMOP. O Sandbox tem como objetivo prevenir o acesso a APIs confidenciais da plataforma Android a partir de um conjunto de métodos que especificam os métodos permitidos para um determinado aplicativo. O JavaMOP é utilizado para criar especificações que irão determinar se um método é permitido ou não para o aplicativo. Com as especificações criadas, se um aplicativo acessar um método do conjunto não especificado, o Sandbox irá realizar o bloqueio desse método, impedido que o aplicativo faça seu uso.

Palavras-chave: mop, android, malware, mineração de sandbox, javamop

Abstract

The present work aims to present the implementation of a Sandbox for the Android platform using JavaMOP. Sandbox aims to prevent access to sensitive Android platform APIs from a set of methods that specify the methods allowed for a given application. JavaMOP is used to create specifications that will determine whether or not a method is allowed for the application. With the specifications created, if an application accesses a method from the unspecified set, the Sandbox will block this method, preventing the application from using it.

Keywords: mop, android, malware, mining sandbox, javamop

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Com a crescente utilização de smartphones e aplicativos que estão presentes no dia a dia das pessoas, surge a necessidade de proteger esses usuários de possíveis ataques e invasão a seus aparelhos que muitas vezes ocorrem por meio dos aplicativos que estão usando, esses aplicativos são chamados de aplicativos *malignos*. Aplicativos malignos muitas vezes são aplicativos com **piggyback** que são criados por invasores, onde esses invasores descompactam aplicativos *benignos* e, em seguida, inserem algum código malicioso neles. A maioria dos malwares é baseada em aplicativos benignos, onde os aplicativos benignos representam o aplicativo original. [1].

Deste cenário surge a ideia de verificar em tempo de execução as *APIs* do Android que estão sendo acessadas de forma indevida, como iniciativa para a criação de um **Sandbox** para prevenir esses acessos. As APIs confidenciais em um aplicativo Android são um meio de acessar por exemplo a lista de contatos, a câmera, as fotos, informações sensíveis do usuário e vários outros recursos disponibilizados pelo Android para um aplicativo.

O uso de uma abordagem de Mineração de Sandbox é uma técnica eficaz para análise de malware do Android, como trabalhos de pesquisa anteriores revelaram [2]. O acesso a APIs no Android é permitido pelo usuário no momento em que é necessário acessar alguma parte do sistema Android para o funcionamento de alguma funcionalidade do aplicativo, porém em aplicativos com piggyback, APIs que não serão utilizadas para o correto funcionamento do aplicativo são utilizadas como um forma de roubar ou acessar alguma informação indevida do usuário.

JavaMOP é uma ferramenta de desenvolvimento no qual o monitoramento é seu princípio fundamental e nos permite verificar os acessos indevidos a APIs confidenciais em um aplicativo. Por meio de especificações, ele é capaz de verificar esses acessos e bloquear a chamada a eles executando alguma exceção já definida manualmente na especificação. JavaMOP é uma instância do framework MOP genérico específico para a linguagem de programação Java [3]. A criação do Sandbox com a utilização de JavaMOP permite ve-

rificar e bloquear os acessos a APIs não utilizadas pelo aplicativo Android benigno no aplicativo com piggyback.

1.1 Problema de Pesquisa

A mineração de sandbox, é uma técnica para confinar uma aplicação a recursos acessados durante testes automáticos [1]. O grupo de pesquisa da Universidade de Brasília já vem explorando o uso da técnica de mineração de sandbox para a identificação de malware, mas ainda não tem uma infraestrutura que receba como input a saída da atividade de mineração de sandbox e que gere um sandbox que realmente impeça que uma API não aceita pelo Sandbox, seja usada por um programa em execução.

1.2 Objetivo

O principal objetivo desse trabalho é explorar o uso da abordagem JavaMOP para implementar Sandbox em aplicativos Android. Para atingir esse objetivo, os seguintes objetivos específicos foram estabelecidos:

- **1**: Fazer um estudo sobre a implementação atual do JavaMOP para Android e identificar suas possíveis limitações.
- **2**: Realizar as adaptações necessárias para que o JavaMOP funcione nas versões mais novas da plataforma Android.
- **3**: Implementar um cenário de Sandbox para uma feature específica de Android utilizando JavaMOP.
- **4**: Avaliar se o protótipo demonstra viabilidade do uso de JavaMOP para implementação de Sandbox.

Capítulo 2

Fundamentação Teórica

Conforme mencionado no capítulo anterior, o principal objetivo desse trabalho de graduação é fornecer uma estrutura que permita a criação de um Sandbox para dispositivos Android, capaz de gerar especificações automaticamente a partir da atividade de mineração de sandbox. Com as especificações geradas automaticamente, espera-se que seja possível utiliza-las como **input** em um sandbox que realmente impeça que uma API não aceita pelo Sandbox seja usada por um aplicativo em execução.

2.1 Sandbox

A partir do uso de um Sandbox, podemos restringir um programa que usa bibliotecas ou ferramentas externas, bloqueando o uso quando o usuário deseja fazer alguma ação que seja necessário algum serviço externo ao programa. Aplicado ao contexto Android, podemos ver esse comportamento quando um aplicativo deseja acessar a Galeria de fotos do usuário e para isso solicita a uma API fornecida pelo sistema Android para conceder permissão para esse acesso. Nesse cenário, temos um aplicativo querendo acessar um serviço que está fora dele, nesse caso a Galeria do usuário.

Esse trabalho de graduação tem o objetivo de fornecer uma estrutura que permita a criação de um Sandbox para dispositivos Android que possa realizar o bloqueio de chamada a API‘s do Android por um aplicativo, que não foi permitida pelo usuário. Com o RV-Android e o JavaMOP é possível criar especificações e formalismos de forma individualizada, mapeando o que é permitido e realizando o bloqueio através da verificação em tempo de execução de alguma API que não foi permitido pelo usuário, tendo assim um Sandbox. Para ter as especificações e formalismos do que é permitido, é usado a técnica de mineração de sandbox.

A mineração de sandbox é uma técnica para confinar uma aplicação e recursos acessados durante testes automáticos [4]. Esse conceito aplicado a aplicativos Android, que são

descritas na seção 2.1.1, funcionaria com o auxílio de ferramentas geradoras de casos de teste [4], para identificar recursos que são acessados durante testes da aplicação. O uso de Sandbox é mais uma maneira para proteger os usuários de aplicativos maliciosos, pois ele restringe seu acesso a recursos e serviços do Android que são mais sensíveis, como por exemplo acesso a Câmera, Galeria de Fotos, Storage do aparelho e vários outros recursos que um smartphone possui hoje.

O acesso a recursos de usuários compartilhados como a localização e contatos, está disponível apenas por meio de APIs dedicadas, que são protegidas por permissões [4].

Uma das formas utilizadas para a criação de Sandbox é utilizar um aplicativo benigno em uma ferramenta geradora de casos de teste para explorar o comportamento do aplicativo, extraindo um conjunto de APIs confidenciais que foram chamadas durante a execução. Com base nas APIs confidenciais que foram chamadas, é criado o Sandbox que bloqueia as demais APIs não chamadas durante a execução desses testes. Aplicativos com **piggyback** são criados por invasores descompactando aplicativos benignos e, em seguida, inserindo algum código malicioso neles. A maioria dos **malwares** é baseada em aplicativos benignos [5]. Aplicativos com **piggyback** são bloqueados durante a sua execução em um Sandbox, pois se for realizada chamadas a APIs não mapeadas na fase de exploração, o Sandbox será capaz de identificá-las.

2.1.1 Ferramentas geradoras de casos de teste

Ao invés de escrever testes ou coletar execuções durante a produção, pode-se também gerá-los. No domínio da segurança, o objetivo principal dessas execuções geradas é encontrar **bugs** [4]. No mercado, existem algumas ferramentas com essa finalidade, entre elas podemos citar o *Monkey*, *Droidmate* e *Boxmate*.

1. **Monkey:** Essa ferramenta, gera fluxos aleatórios de eventos que o usuário poderia ter como por exemplo clique, toques ou gestos. Ele é normalmente usado como testador de robustez, mas também para encontrar **bugs** de interface gráfica e `bugs` de segurança.

2. **Droidmate** Gera testes automatizados para explorar comportamentos de um aplicativo Android.

3. **Boxmate** Identifica APIs sensíveis do Android chamadas durante a execução de casos de teste e as usa para formar um sandbox [4].

2.2 Monitoring Oriented Programing (MOP)

No desenvolvimento de software é comum realizar testes para verificar o correto funcionamento do software, desde as primeiras **features** desenvolvidas até a versão final do produto. Existem fases durante o desenvolvimento de software em geral e para o desenvolvimento de aplicativos Android não é muito diferente, visto que há a necessidade de se realizar testes durante o desenvolvimento de forma que aumente a confiabilidade não só do produto mas do código em si além da segurança do sistema. O monitoramento das execuções de um sistema em relação às propriedades esperadas, desempenha um papel importante não apenas nos estágios de desenvolvimento do sistema, por exemplo, depuração e teste, mas também no sistema implantado como um mecanismo para aumentar a confiabilidade ou a segurança do sistema [3].

Tal monitoramento é feito através das especificações de propriedades, onde durante a execução de um programa são monitorado partes específicas dele com especificações que irá resultar em ações de acordo com o que está sendo monitorado. Quando temos uma propriedade violada ou válida, são tomado ações de acordo com a finalidade do evento executado.

Monitoramento de tempo de execução é uma técnica utilizável em todas as fases do ciclo de desenvolvimento de software, desde o teste inicial até a depuração e a manutenção da função adequada no código de produção [3]. O MOP separa geração e integração de monitores e fornece uma estrutura genérica e extensível para monitoramento em tempo de execução, permitindo instanciar o MOP com linguagens de programação específicas e formalismos de especificação para suportar diferentes domínios [6]. Neste trabalho foi utilizado framework MOP que é utilizado especificamente para a linguagem de programação Java, chamado JavaMOP.

2.2.1 JavaMOP

A ferramenta central para a criação de um Sandbox que irá permitir a criação de propriedades que de certa forma delimitam o uso de um aplicativo é o JavaMOP, que junto com o RV-Android permite definir regras para um determinado aplicativo. JavaMOP é o único sistema de monitoramento paramétrico, que permite a especificação de propriedades que relacionam objetos em um programa, ao invés de apenas propriedades globais [3]. As propriedades paramétricas são propriedades com variáveis livres que permitem descrever comportamentos globais e comportamentos de objetos de um programa ou aplicativo.

Uma outra definição de JavaMOP, consiste em uma coleção integrada de programas especializados, permitindo processar anotações em programas Java de maneira fácil, natural e automática [7].

Usando o JavaMOP, conseguimos monitorar a execução de um programa através de especificações de eventos como mostra o trecho de código a seguir, no qual especifica que depois de qualquer chamada a um **Objeto** que for realizada por um aplicativo, mostrar nos `logs` da execução do aplicativo a frase *JavaMOP: Chamada a objeto realizada!*.

Código 2.1: Especificação de evento JavaMOP

```
event g1 after (Object o): (call(* Object+.*(..)) && target(o)) {
    Log.v("JavaMOP: Chamada a objeto realizada!");
}
```

Para definir se uma especificação é validada ou violada, o JavaMOP utiliza formalismos lógicos que podem ser *expressões regulares* (`ere`), *gramática livre de contexto* (`cfg`), *máquina de estado finito* (fsm), *lógica temporal linear de tempo passado* (`ptltl`), *lógica temporal linear de tempo futuro* (`ftltl`) e *lógica temporal linear de tempo passado lógica com chamadas e retornos* (`ptCaRet`) [6].

O trecho do código 2.2, ilustra o uso de formalismos lógicos usando *expressões regulares* (`ere`) onde *g1* é o evento do código 2.1 e o `*` significa que se houver uma mais chamadas desse evento. Após a verificação do formalismo lógico utilizado, sabemos se uma especificação é validada ou violada e a partir desse ponto, ações previamente definidas por quem escreveu a especificação são executadas. Essas ações definidas são chamadas de manipuladores e podem ser qualquer código Java. No trecho de código onde tem a palavra `@match`, representa o manipulador que nesse caso dispara somente um Log.

Como o JavaMOP depende de um compilador AspectJ externo para tecer o código de monitoramento, o JavaMOP só permite eventos que podem ser definidos no AspectJ padrão [3].

Código 2.2: Formalismo usando `ere`

```
ere : g1*

@match {
    Log.v("JavaMOP:", "match: Evento g1 realizado");
}
```

2.3 Programação Orientada a Aspecto (AOP)

O processo de compilação do JavaMOP envolve o uso do **AspectJ**, que é uma linguagem de programação pertencente ao paradigma de programação orientada a aspectos **AOP**.

AOP é uma técnica de desenvolvimento de software que visa a separação de interesse [8]. O comportamento desses aspectos são definidos pelo programador, da mesma forma que as especificações são definidas no JavaMOP. Para diferenciar melhor o uso de MOP e AOP, podemos dizer que MOP é ajustado e otimizado para mesclar especificação e implementação via monitoramento, enquanto o AOP visa a separação de interesse [7].

Os aspectos dependem de três conceitos: *joinpoints*, *pointcuts* e *advices*. Um `joinpoint` é um ponto identificável na execução de um programa alvo. Um `pointcut` seleciona um conjunto de joinpoints. Então, um `advice` é um pedaço de código associado a alguns pointcuts [9]. Quando a execução de um programa atinge um joinpoint selecionado por um pointcut, eles alteram ou aumentam a execução desse programa. Isso pode ser visto de acordo com as ações especificadas pelo usuário e exemplificados com os códigos gerados a partir do JavaMOP. Usando como referência o código 2.1 que especifica um evento g1, após a compilação desse código é gerado um arquivo AspectJ, listado no código 2.3.

2.3.1 AspectJ

Após fazer as especificações para realizar o monitoramento em tempo de execução, o JavaMOP gera código AspectJ para monitoramento, que é inserido no programa por meio de um compilador AspectJ como por exemplo seu compilador padrão `ajc`. **AspectJ** é o compilador de aspecto mais popular e eficiente atualmente disponível [9].

Código 2.3: Resultado em AspectJ da compilação do código MOP listado em 2.1

```
pointcut MOP_CommonPointCut() :
    !within(com.runtimeverification.rvmonitor.java.rt.RVMObject+) &&
    !adviceexecution() &&
    BaseAspect.notwithin();

pointcut Logger_g1(Object o) : (call(* Object+.*(..))
    && target(o))
    && MOP_CommonPointCut();

after (Object o) : Logger_g1(o) {
        LoggerRuntimeMonitor.Logger_g1Event(o);
}
```

Nesse código é possível observar os *pointcuts* gerados automaticamente e que serão usados na instrumentação do programa Java, no caso mencionado nesse trabalho, no aplicativo Android. De maneira mais clara, pode-se notar o *pointcut* adicionado referente ao evento *g1*, no código 2.3 identificado como `Logger_g1`, que observa no aplicativo todas as chamadas a qualquer método de qualquer *Objeto*.

2.4 RunTime Verification

Outro conceito essencial no processo de criação de um *Sandbox* e de verificação da *APIs* sensíveis no Android é o de **RunTime Verification (RV)**, que é uma área que dá mais rigor aos testes. Em **RV**, os monitores são sintetizados automaticamente a partir de especificações formais, ou seja, após a compilação de especificações JavaMOP, também são gerado arquivos *.rvm*. Ainda utilizando o trecho de código em JavaMOP listado em 2.1, após a compilação desse arquivo *mop*, além do arquivo em AspectJ gerado, o trecho de código abaixo (2.4) também é gerado e usado para a verificação em tempo de execução do aplicativo Android.

Código 2.4: Resultado .rvm da compilação do código MOP listado em 2.1

```
Logger() {
        event g1(Object o){
                Log.v("JavaMOP: Chamada a objeto realizada!");
        }
        ere:  g1*

        @match
        {
                Log.v("JavaMOP:", "match: Evento g1 realizado");
        }
}
```

2.4.1 RV-Monitor

O **RV-Monitor** será usado para compilar as especificações em AspectJ e **RVM**. Apesar de o RV-Monitor representar uma evolução do JavaMOP [5], é a partir dele que geramos código RVM e AspectJ para em seguida compilá-los e realizar o monitoramento em tempo de execução. Assim como o JavaMOP, o RV-Monitor suporta *plugins* de lógica, permitindo a especificação de propriedades em vários formalismos, incluindo *expressões regulares, gramáticas livres de contexto, autômatos e lógica temporal linear de tempo passado* [5].

RV-Monitor permite a especificação de propriedades formais sobre eventos e os pontos de instrumentação para esses eventos em um único arquivo orientado a monitor. O RV-Monitor é compatível com qualquer método de instrumentação Java, e o projeto JavaMOP pode ser estendido para gerar a entrada necessária para outras ferramentas de instrumentação, se necessário [5].

2.4.2 RV-Android

A ferramenta que irá de certa forma orquestrar o processo de instrumentação de um aplicativo Android e adicionar o monitoramento em tempo de execução é o **RV-Android**. O RV-Android consiste em dois componentes, uma ferramenta de geração de bibliotecas de monitoramento e um ambiente de tempo de execução usado na geração dessas bibliotecas para monitoramento dinâmico de propriedades no dispositivo e recuperação de violações [5].

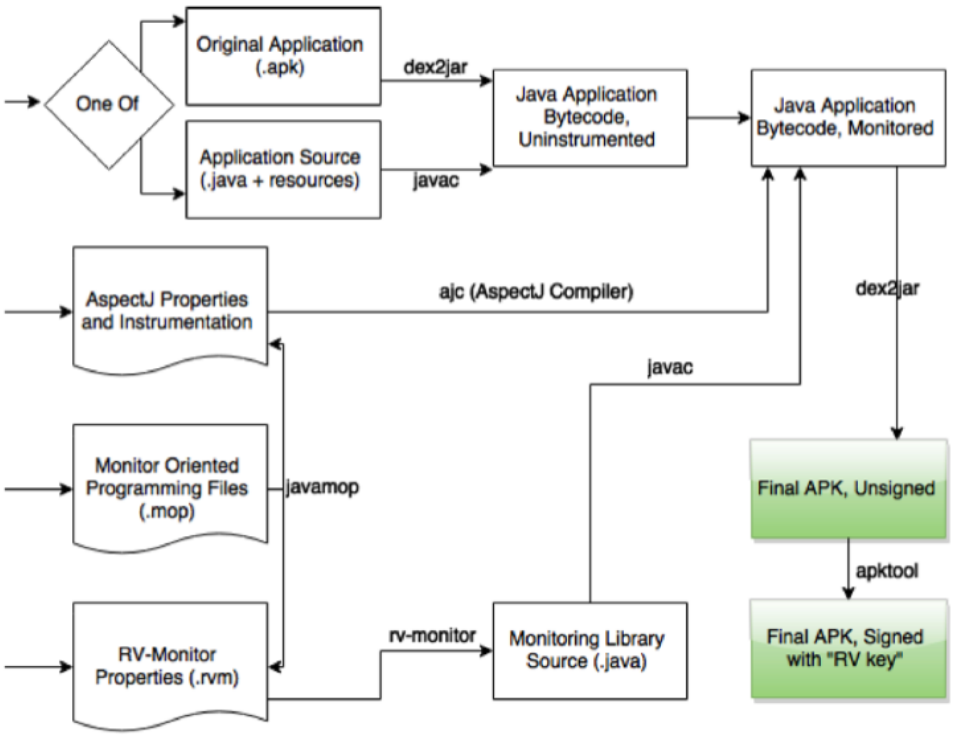

Figura 2.1: Fluxo de compilação RV-Android (Fonte: [5]).

A Figura 2.1 mostra o fluxo de execução do RV-Android, onde mostra no início do fluxo, os arquivos de entrada para o processo de instrumentação, um arquivo Android empacotado *.apk* ou um conjunto de arquivos *.java* como entrada. Além do aplicativo Android como entrada, seja ele a .apk ou o código .java, o RV-Android aceita arquivos de aspecto, propriedades do RV-Monitor *(.rvm)* e arquivos compatíveis com programação orientada a monitoramento, no nosso caso é utilizado o JavaMOP. Durante o processo do RV-Android, ferramentas terceiras são utilizadas para o processo de *descompilação* da apk, para compilar uma nova apk e assinar a nova apk com os monitores junto. Para essas finalidades, o **dex2jar** é utilizado.

O dex2jar possui outras ferramentas dentro de seu conjunto, onde cada uma tem uma utilidade diferente. Durante a execução do RV-Android, é utilizado as seguintes ferramentas do dex2jar:

- **d2j-dex2jar**: No início do processo onde uma apk é usado como entrada, é necessário extrair as classes desse aplicativo, transformando-o primeiro em um .jar.
- **2j-asm-verify**: Após converter o apk em .jar, essa ferramenta do dex2jar verifica o .jar extraído.
- **d2j-jar2dex**: Após ter as classes do .jar, é necessário transformar essas classes no .jar novamente, e para isso o d2x-jar2dex é utilizado.
- **d2j-apk-sign**: Depois de ter realizado o processo de instrumentação, uma nova apk é gerada e com o d2j-apk-sign é realizado a assinatura dessa nova apk.

No final do processo do RV-Android, uma apk é gerada e nela é empacotado os arquivos de monitoramento gerados a partir do JavaMOP, AspectJ e RV-Monitor. Essa etapa pode ser observada no final do processo ilustrado na Figura 2.1.

Capítulo 3

Relato de Experiência

O objetivo geral desse estudo conforme discutido na introdução, é identificar se JavaMOP é uma técnica adequada, para identificar a viabilidade do uso de JavaMOP para implementação de Sandbox para Android. A metodologia de pesquisa utilizada nesse estudo, é um estudo de caso, cujo o objetivo é responder as seguintes questões de pesquisa:

- **Questão de pesquisa 1**: A implementação atual de JavaMOP para Android, é funcional?
- **Questão de pesquisa 2**: Como adaptar a implementação atual de JavaMOP para Android para as versões mais atuais do SDK.
- **Questão de pesquisa 3**: O quanto é viável a implementação de JavaMOP para a utilização de Sandbox?

O processo inicial para a execução do RV-Android, consiste na configuração do ambiente com algumas bibliotecas e ferramentas necessárias durante a execução, como é o exemplo do JavaMOP. RV-Android aceita como entrada arquivos MOP, como mostrado na Figura 2.1, onde especifica o fluxo de execução e os arquivos de entrada que aceita. Foi necessário baixar o JavaMOP e configurá-lo nas variáveis de ambiente do sistema, tornando-o disponível para o RV-Android executa-lo. O RV-Android utiliza o RV-Monitor como sua principal tecnologia de geração de biblioteca de monitoramento, permitindo a verificação das propriedades de segurança durante a execução e operando inteiramente no espaço do usuário sem a necessidade de modificações no kernel ou no sistema operacional [5]. Dessa forma, foi necessário realizar a configuração do RV-Monitor nas variáveis de ambiente para ser utilizado no momento da compilação dos monitores que serão utilizados no momento da instrumentação do aplicativo. Por último, outro requisito fundamental é o AspectJ e o compilador padrão ajc, usado para gerar os pointcuts necessários para a instrumentação. Foi preciso usar uma versão igual ou superior a 1.8 e definir essa versão

na linha de comando do processo de execução do ajc, responsável por compilar e gerar os pointcuts.

Com esses pré-requisitos instalados e configurados no ambiente, foi possível realizar o estudo de caso e responder às questões de pesquisa.

3.1 Problemas da Versão Atual

3.1.1 Verificação de Classes do .jar

No processo de execução do RV-Android são necessárias algumas tratativas quanto à APK que será instrumentada, seja ela de converter a apk para .jar, verificar as classes do .jar e verificar e realizar assinatura da apk. Para executar esses processos, é usado o *dex2jar* no qual já vem com uma versão nas bibliotecas utilizadas no RV-Android atual. Como RV-Android foi desenvolvido a cerca de 8 anos, os aplicativos desenvolvidos naquela época são definitivamente diferentes dos desenvolvidos atualmente e isso é percebido quando tentamos utilizar a versão do dex2jar de 8 anos atrás em aplicativos mais recentes, mais especificamente quando usamos a biblioteca do dex2jar para verificar as classes contidas de um .jar que foi extraído de uma apk atual. Ao executarmos essa mesma biblioteca na apk padrão que vem junto com o RV-Android desenvolvido originalmente 8 anos atrás, a verificação ocorre sem nenhuma falha, mas quando tentamos realizar a mesma verificação em uma apk atual, são apontados vários erros na verificação dos métodos das classes presentes na apk atual.

3.1.2 Conversão de .jar para .dex

Outro papel desempenhado por uma das bibliotecas do dex2jar é na conversão do arquivo .jar gerado após toda a instrumentação feita pelo RV-Android. Depois de realizadas todas as etapas para instrumentação do aplicativo, é necessário convertê-lo novamente para formato apk que permita o usuário instala-lo em um dispositivo Android e que esteja funcionando corretamente. Antes disso é preciso converter o .jar gerado no processo de instrumentação com os arquivos de monitoramento para o formato .dex e posteriormente copiar o .dex para a apk já empacotada. Para esse processo em questão, o dex2jar possui uma biblioteca que se chama jar2dex e realiza esse procedimento, porém mesmo após realizar a atualização do dex2jar para uma versão mais recente, não estava sendo possível fazer a conversão com sucesso. O seguinte erro era mostrado durante a execução do RV-Android nessa etapa de conversão:

```
Uncaught translation error: com.android.dx.cf.code.SimException: local
```

```
variable type mismatch: attempt to set or access a value of type int[]
using a local variable of type androidx.transition.TransitionValues.
This is symptomatic of .class transformation tools that ignore
local variable information.
```

Após diversas pesquisas, não ficou claro o motivo exato do que levava a acontecer esse problema, pois o mesmo acontecia tanto no aplicativo de exemplo padrão de 8 anos atrás e no aplicativo mais atual que foi desenvolvido durante a execução deste trabalho de graduação.

3.1.3 Instrumentação

O repositório original do RV-Android veio acompanhado de um aplicativo feito na época como já foi relatado em uns do problemas anteriores e com alguns arquivos de monitoramento, que realizava a monitoração por exemplo de quando era acessado a API de localização. Executando o RV-Android para esse aplicativo que já veio no projeto com os arquivos de monitoração, notei que a verificação em tempo de execução conseguia capturar os eventos relacionados às APIs que foram especificadas nos arquivos de monitoramento. O próximo passo foi adicionar um aplicativo novo e atual com um novo arquivo de monitoramento. Esse arquivo de monitoramento novo, capturava todas as chamadas de métodos realizadas durante a execução do novo aplicativo. Após a execução do RV-Android no novo aplicativo, o mesmo não capturava os eventos e não apresentava nenhum sinal da instrumentação feita. Executando o RV-Android com esse novo arquivo de monitoramento no aplicativo antigo que veio por padrão no RV-Android, ele também não apresentava nenhum sinal de que estava sendo monitorado todos os eventos, mesmo continuando a captação dos eventos de acesso a API, Storage, etc.

Resumidamente, não estava sendo feito a instrumentação dos aplicativos com qualquer que seja a especificação utilizada ou aplicativo utilizado para teste. Funcionava somente o aplicativo padrão com as especificações padrões que tinha de exemplo com o RV-Android atual.

A partir dos problemas mencionados, é possível responder a primeira questão de pesquisa, no qual fica claro que a versão atual necessita de diversas atualizações de bibliotecas terceiras para poder funcionar.

3.2 Correções realizadas

3.2.1 Atualização do dex2jar

Partindo para a segunda questão de pesquisa, torna-se necessário a atualização de algumas bibliotecas terceiras, a primeira delas a ser atualizada foi a dex2jar. Após realizar a atualização do dex2jar para a versão mais recente, foi possível realizar as verificações tanto em aplicativos mais antigos como por exemplo na apk padrão que vem com o RV-Android, como nos aplicativos mais novos, como por exemplo a apk atual que estava dando erro na versão mais antiga do dex2jar.

Para a solução no processo de conversão do arquivo .jar para .dex foi um pouco mais trabalhoso, pois eu estava sem entender claramente porque estava acontecendo esse problema e não conseguia soluciona-lo. Então procurei outras ferramentas que fizessem o mesmo procedimento de converter o .jar para .dex e que fosse mais atual com atualizações mais recentes.

O Android possui algumas ferramentas de linha de comando que podem ser encontradas em `https://developer.android.com/studio/command-line`, e entre elas a **d8**. Essa ferramenta d8 é encontrada a partir do *Android SDK Build Tools 28.0.1* e é usada para compilar *bytecode* java para o formato *dex* que é executado em dispositivos Android. Após atualizar o RV-Android para utilizar essa ferramenta, o resultado esperado dessa conversão foi obtido e agora era possível ter o arquivo .dex com os arquivos de monitoramento que foram compilados e gerados durante o processo de instrumentação do RV-Android.

Após as correções e atualizações necessárias descritas, a execução do RV-Android estava sendo finalizada sem erros e a próxima etapa era realizar a instrumentação de alguns aplicativos para ver o funcionamento completo da ferramenta.

3.2.2 Resolução da Instrumentação

Com as bibliotecas atualizadas e a execução do *script* de instrumentação sendo finalizada sem erros aparentes, iniciou-se a fase de testes das instrumentações, e foi onde surgiu o problema que qualquer aplicativo e qualquer especificação não estavam funcionando.

Investigando mais a fundo esse problema, precisei ver o resultado da apk que funcionava, ou seja, a que já veio com o RV-Android e o resultado da nova APK instrumentada que não funcionava. Para isso, extraí manualmente os arquivos da APK apenas renomeando para *.zip* e usei o arquivo *classes.dex* presente na apk para extrair o *.jar* contendo os arquivos *.class* do aplicativo. Para obter o *.jar* do *classes.dex* utilizei a fer-

ramenta dex2jar que também foi utilizada no processo do RV-Android. Dessa forma, com o .jar contendo os arquivos *.class* em mãos, usei uma ferramenta de interface que permite visualizar código fonte Java desses arquivos .class chamada `JD-GUI`, presente em `http://java-decompiler.github.io`.

A partir desse momento pude investigar melhor o código fonte de cada aplicativo, tanto do antigo que estava funcionando o monitoramento quanto do novo aplicativo no qual o monitoramento não estava funcionando. A apk resultante da execução do RV-Android no aplicativo antigo, possuía alguns arquivos relacionados ao monitoramento e de especificações dos eventos para capturar as chamadas APIs, como pode ser visualizado na Figura 3.1. O pacote `monitors` possui arquivos relacionados às especificações de eventos que terá ações ao acessar alguma API do sistema Android, como por exemplo Localização, Notificação e Armazenamento e a monitores que quando a classe `Closeable` for chamada mais de uma vez será disparado uma mensagem de *log*. Esse é o resultado após a execução do RV-Android também com o arquivo de monitoramento adicionado como teste, que captura todos os eventos disparados pelo aplicativo, e como pode observar, não possui nenhum arquivo de monitoramento para essa finalidade empacotada na apk.

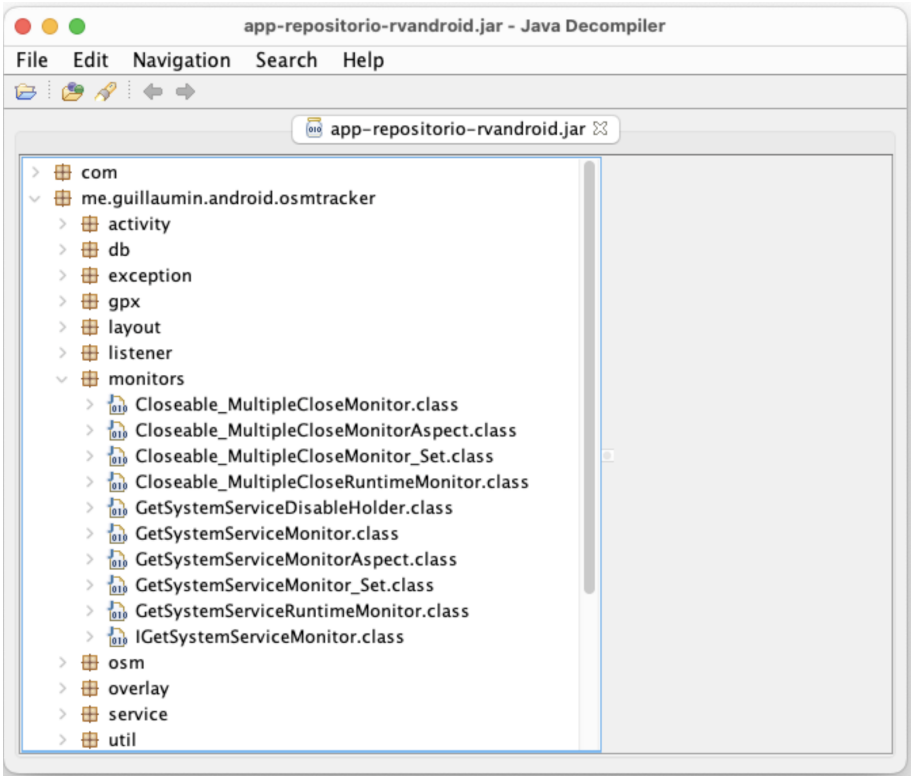

Figura 3.1: Interface com arquivos do APK padrão do RV-Android.

Fazendo o mesmo processo no aplicativo novo, não foi encontrado nem mesmo os monitores presentes na Figura 3.1. Investigando também o aplicativo que veio na ferramenta

RV-Android sem executar o processo de instrumentação e visualizando seu código fonte Java, foi percebido que o mesmo já vinha instrumentado com os arquivos de monitoração, o que explicava porque esse aplicativo estava apresentando os logs quando um evento era capturado em tempo de execução, ele já veio instrumentado. Percebi que por algum motivo a instrumentação após rodar RV-Android não estava funcionando, mesmo sem apontar erros na execução.

Após investigar passo a passo da execução do RV-Android, notei que no processo em que o novo classes.dex com os arquivos de monitoramento é substituído na apk, o mesmo não estava sendo encontrado e logo não era substituído. O aplicativo que tínhamos na saída do RV-Android, permanecia o mesmo que colocamos na entrada, sem nenhum arquivo de monitoramento. O que acontecia é que no momento de substituir o classes.dex, o mesmo não estava com o caminho especificado corretamente e não era encontrado, dessa forma nada era substituído no apk original. Após corrigida essa etapa, o apk gerado como saída do RV-Android passou a ter todos os arquivos de monitoramento e passou a exibir os logs de monitoramento em tempo de execução.

Capítulo 4

Prova de Conceito

Nesse capítulo será apresentada a solução proposta para a implementação do Sandbox utilizando JavaMOP. A apresentação ocorrerá em quatro partes, onde em cada parte são apresentadas informações com detalhes que levaram ao resultado obtido que é discutido na parte parte do funcionamento do Sandbox. Primeiro são apresentados os conjuntos de métodos das APIs, os que são sensíveis, os que são permitidos por um aplicativo e os que não são permitidos e como serão utilizado nos testes. Em seguida, o aplicativo Android desenvolvido para os testes é apresentado, bem como informações do desenvolvimento. Na terceira parte, o funcionamento do Sandbox usando os aplicativos desenvolvidos e os conjuntos de métodos mencionados na primeira parte com suas especificações.

4.1 Conjunto de Métodos das APIs

Como mencionado nos capítulos anteriores, a maioria dos aplicativos maliciosos são baseados em aplicativos benignos [1], ou seja, aplicativos que foram modificados inserindo algum código malicioso nele, o chamados aplicativos com **piggyback**. Esses aplicativos com piggyback, ou também conhecidos como aplicativos malignos, geralmente fazem o uso de alguma API externa do Android, como de localização do usuário, galeria de fotos, dentro várias outras que permite ter informações pessoais do usuário. Com esse problema, surge a ideia da implementação do Sandbox, que permite bloquear os acessos a essas APIs, chamadas APIs sensíveis. A construção do Sandbox irá fazer o uso de um conjunto de métodos que não são permitidos por um aplicativo Android. Esse conjunto de métodos é extraído a partir do conjunto de métodos das APIs sensíveis e do conjunto de métodos das APIs permitidas por um aplicativo. O conjunto de métodos da diferença entre esses dois conjunto, será usado para fazer as especificações do Sandbox, que serão os métodos a serem bloqueados.

Tabela 4.1: Lista de algumas APIs sensíveis.

Pacote	**Classes**	**Métodos**
android.app	ActivityManager	getRecentTasks(int,int)
android.app	ActivityManager	getRunningTasks(int)
android.speech	SpeechRecognizer	startListening()
android.media	MediaRecorder	setAudioSource(int)
android.media	AudioManager	isWiredHeadsetOn()
android.media	AudioManager	setMicrophoneMute(boolean)
android.hardware	Camera	open()
android.hardware	Camera	open(int)
android.telephony	TelephonyManager	getPhoneType()
android.telephony	TelephonyManager	getImei()
android.telephony	TelephonyManager	getSimSerialNumber()
android.netwifi	WifiManager	disconnect()
android.netwifi	WifiManager	reconnect()

Tabela 4.2: Identificação de API Sensível.

Método Android
android.app.ActivityManager: java.util.List getRecentTasks(int,int)
android.app.ActivityManager: java.util.List getRunningTasks(int))
android.speech.SpeechRecognizer: void startListening(android.content.Intent)
android.media.MediaRecorder: void setAudioSource(int)
android.media.MediaRecorder: void setAudioSource(int)
android.media.AudioManager: void setMicrophoneMute(boolean)
android.hardware.Camera: android.hardware.Camera open()

4.1.1 APIs Sensíveis

Com o uso de uma ferramenta geradora de caso de teste como em [2], é possível extrair de um aplicativo Android, todas as chamadas a APIs que aquele aplicativo faz e dessa forma identificar os que são permitidos por ele. Essas APIs sensíveis do Android são usadas através de métodos das Classes de cada API disponibilizada pelo Android. Para ilustrar melhor, a Tabela 4.1 descreve as classes e métodos existentes que podem ser usadas por um aplicativo e o que ela faz, e também mostra a qual pacote pertence.

Para exemplificar melhor, a Tabela 4.2 mostra o código de como o método que é usado pelo aplicativo é identificado e listado. Esse código é usado posteriormente para poder capturar os métodos acessado por um aplicativo e fazer a composição do Sandbox, usando também a Classe a qual esse método pertence.

Esse primeiro conjunto de APIs sensíveis, é a base para iniciar o processo da construção do Sandbox, é a partir dele que é extraído as APIs e Métodos que são permitidos e os que não são permitidos.

Tabela 4.3: Lista APIs permitidas.

Pacote	Classes	Métodos
android.app	ActivityManager	getRecentTasks(int,int)
android.app	ActivityManager	getRunningTasks(int)
android.speech	SpeechRecognizer	startListening()
android.media	MediaRecorder	setAudioSource(int)
android.media	AudioManager	isWiredHeadsetOn()
android.media	AudioManager	setMicrophoneMute(boolean)
android.hardware	Camera	open()
android.hardware	Camera	open(int)
android.telephony	TelephonyManager	getSimSerialNumber()
android.netwifi	WifiManager	disconnect()
android.netwifi	WifiManager	reconnect()

Tabela 4.4: Lista de APIs bloqueadas.

Pacote	Classes	Métodos
android.telephony	TelephonyManager	getPhoneType()
android.telephony	TelephonyManager	getImei()

4.1.2 APIs Permitidas

Com o conjunto de APIs sensíveis obtidas, o próximo passo é extrair as que são permitidas pelo Aplicativo, para em seguida trabalhar no conjunto que interessa ao Sandbox, o conjunto de APIs a serem bloqueadas. Para exemplificar aqui, vamos pegar as seguintes APIs sensíveis que são permitidas. Usando a Tabela 4.1 como base, teremos na Tabela 4.3 as APIs que são permitidas pelo aplicativo.

4.1.3 APIs Bloqueadas

Após ter o conjunto de APIs Sensíveis usadas por um aplicativo e ter obtido as APIs que são permitidas dentro desse conjunto, ao fazer a diferença, podemos ter aquelas que não foram permitidas, ou seja, as que devem ser bloqueadas ao serem usadas no Sandbox.

Ainda usando como referência a Tabela 4.1 e agora a Tabela 4.3 para fazer a diferença entre as duas, obtemos o terceiro conjunto, mostrado na Tabela 4.4 e que será usado para a composição do Sandbox.

4.2 Aplicativos de teste

Como mencionado na Seção 4, o aplicativo desenvolvido e ilustrado na Figura 4.1, terá como objetivo testar a criação do Sandbox para os métodos mencionados, que no caso são os métodos `getPhoneType()` e `getImei()` da classe `TelephonyManager`.

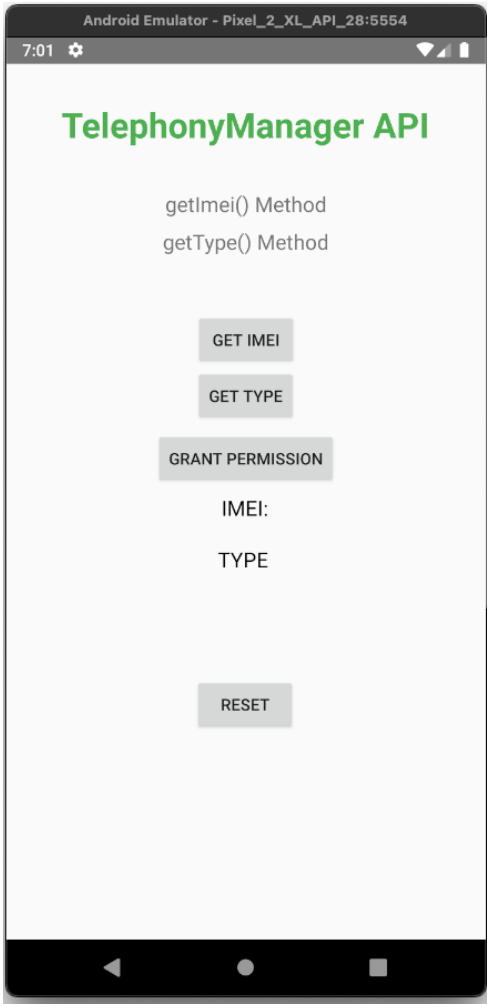

Figura 4.1: Aplicativo de teste para o Sandbox.

Descrevendo o aplicativo, os botões com o nome **GET IMEI** e **GET TYPE** irão fazer a chamada aos seus respectivos métodos, que devem ser bloqueados pelo Sandbox. A Figura 4.1 e Figura 4.2 mostra o código do aplicativo referente aos métodos que devem ser bloqueados, onde é possível identificar a chamada a eles bem como a classe no qual eles pertencem. Nota-se que segue a mesma assinatura presente na tabela 4.4. Foi necessário importar a API do TelephonyManager para isso: `import android.telephony.TelephonyManager`.

Após obter as informações do tipo de telefone, uma estrutura `Switch/Case` foi usada somente para obter um valor como `String` para apresentar a tela.

Código 4.1: Código da chamada a API que pega o tipo do telefone

```
int phoneType = telephony.getPhoneType();
String telephonyType;

switch (phoneType) {
    case (TelephonyManager.PHONE_TYPE_CDMA):
        telephonyType = "CDMA";
        break;
    case (TelephonyManager.PHONE_TYPE_GSM):
        telephonyType = "GSM";
        break;
    case (TelephonyManager.PHONE_TYPE_NONE):
        telephonyType = "NONE";
        break;
    default:
        throw new IllegalStateException("Unexpected value");
}

telephoneTypeView.setText("Type: " + telephonyType);
```

Código 4.2: Código da chamada a API que pega o IMEI

```
String telephonyImei = telephony.getImei();

phoneImeView.setText("IMEI: " + telephonyImei);
```

A versão do SDK que foi utilizada para o desenvolvimento e compilação do Aplicativo foi a versão 26.

4.3 Funcionamento do Sandbox

A versão atual do Sandbox não é mantido a mais de 8 anos e precisou receber algumas atualizações, pois foram encontradas diversas dificuldades que tomaram bastante tempo do desenvolvimento desse trabalho para ser resolvido. Todos os problemas encontrados e as soluções utilizadas, podem ser conferidas em 3.

No geral, o fluxo do Sandbox atualizado e adaptado para JavaMOP ficou como ilustrado na Figura 4.2, onde é necessário somente a APK a ser instrumentada e especificações JavaMOP. Na nova forma de execução por linha de comando do RV-Android, torna-se necessário informar somente a pasta onde as especificações JavaMOP se encontram, como por exemplo: `./instrument_apk.sh [apk] [keystore] [keystore password] [signing key alias] [mop_directory]`.

As demais etapas do fluxo são descritos em 3.

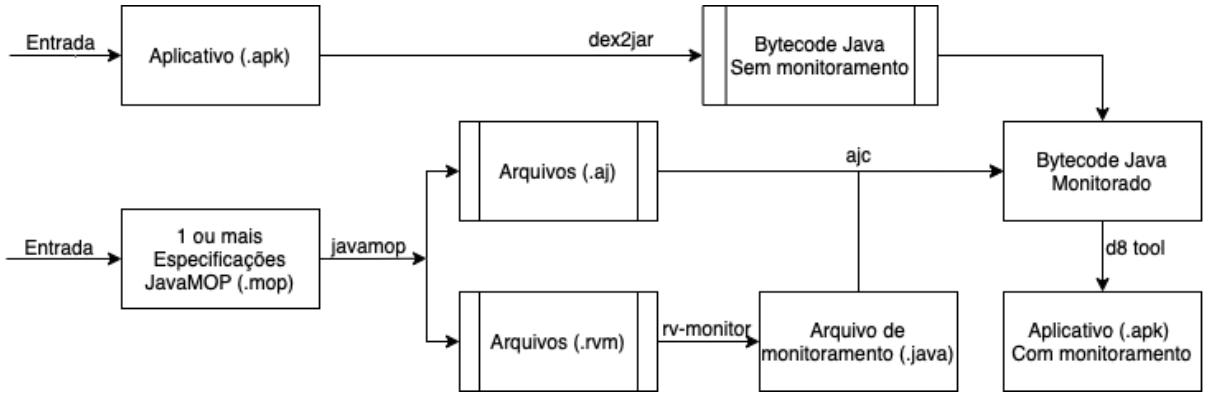

Figura 4.2: Fluxo da versão atual do RV-Android atualizado.

Com o conjunto de APIs a serem bloqueadas e o aplicativo de teste pronto, o próximo passo é fazer as especificações em JavaMOP que irão realizar o bloqueio no momento em que fosse chamado as APIs da Tabela 4.4. Seguindo o mesmo padrão do exemplo do trecho de código em 2.2, foram feitos os formalismos com as regras e modificações necessárias. No trecho de código 4.3, podemos notar que agora o evento é executado antes da chamada do método (`before`), pois nesse caso não é interessante para o usuário que o Sandbox permita a chamada de método para somente depois bloquear essa API.

Outra diferença é dentro do `call`, onde é especificado exatamente o escopo e o tipo de retorno do método respectivamente no primeiro evento **gt** (`public int`) e no segundo evento **gi** (`public String`).

O formalismo lógico utilizado (`ere`) também mudou um pouco, agora verificando se o evento **gt** ou **gi** foi chamada 1 ou mais vezes, para poder fazer o `@match`.

A grande mudança em relação a 2.2, é que agora um código Java foi "acoplado"com a função de lançar uma exceção caso o formalismo lógico seja verdadeiro e assim encerrar o aplicativo. Até o momento, encerrar a execução do aplicativo foi a única forma encontrada para bloquear a chamada a API. O código Java é descrito em 4.4.

Código 4.3: Especificação JavaMOP que bloqueia getImei e getPhoneType

```
import android.telephony.TelephonyManager;

TelephonyManagerSpec() {

   event gt before(TelephonyManager tm) : (call(
     public int android.telephony.TelephonyManager.getPhoneType())
         && target(tm)) {
         Log.v("RV-MONITOR: ", "Calling Object: ");
   }

   event gi before(TelephonyManager tm) : (call(
         public String android.telephony.TelephonyManager.getImei())
         && target(tm)) {
         Log.v("RV-MONITOR: ", "Calling Object: ");
   }

   ere : (gt | gi)*

   @match {
         Log.v("RV-ANDROID: ", "Method not Allowed");

         throw new SandboxViolationException(
            "TelephonyManager.getImei()"
         );
   }
}
```

Código 4.4: Código que lança a exceção e encerra o App

```
 package mop;

 public class SandboxViolationException extends RuntimeException{
   public SandboxViolationException(String mn){
     super("Invalid call to method: " + mn);
   }
 }
```

Após executar o RV-Android, o novo aplicativo pronto para ser instalado, ficará disponível na pasta **out**. Na seção seguinte, é discutido os resultados obtidos com a construção desse Sandbox.

4.3.1 Resultados Obtidos

Após o processo de criação do Sandbox, feito todas as especificações e a instrumentação, esse novo aplicativo gerado deverá bloquear toda e qualquer chamada a API que não seja permitida, no nosso caso os métodos `getImei()` e `getPhoneType()` da API de `TelephonyManager`.

Quando clica nos botões que realizam a chamada a esses métodos, o aplicativo é fechado na mesma hora, refletindo o que foi especificado em JavaMOP com a classe feita em Java. Se visualizado os logs do Android no momento da execução, é possível ver a mensagem especificada em `Log.v` no trecho de código 4.3 antes da exceção ser lançada.

Respondendo ao quarto item do objetivo de pesquisa, o protótipo apresentado nesse trabalho demonstra viabilidade do uso dessa nova versão, pois o resultado esperado foi alcançado, onde um Sandbox que bloqueia chamadas a API funcionou, porém ainda torna-se necessário uma forma melhor de tratar como a exceção é lançada e bloquear a chamada API.

Capítulo 5

Conclusão

Foi possível ter uma versão nova do RV-Android que funcione para as versões mais recentes do Android junto com JavaMOP, e para isso foram necessárias diversas adaptações na versão que tinha atualmente e desatualizada. Durante o processo de exploração foi demandada diversas horas de pesquisa e estudo para atualizar as ferramentas terceiras que o RV-Android faz o uso e surgiu a necessidade de substituir ferramentas terceiras no momento final do processo de instrumentação, como foi o caso do uso da biblioteca d8 para transformar os arquivos instrumentados em .dex novamente antes de inserir na nova apk.

Dessa forma, torna-se viável o uso dessa nova implementação e podem ser necessárias algumas adaptações que talvez possam ser definidas de forma automática, como por exemplo no momento de adicionar o .jar da versão do Android que possua um método específico de uma API do Android no momento da execução do compilador ajc do AspectJ, pois por exemplo o método usado aqui como exemplo `getImei()` não tem disponível no .jar do android-17, e nesse caso seria necessário trocar. A atual implementação também pode servir de input para trabalhos no qual pesquisadores da Universidade de Brasília vem trabalhando, na implementação de Sandbox adaptado para Android.

Essa nova versão pode ser obtida através do repositório do Github `https://github.com/gsmartins96/rv-android` e no seguinte repositório está disponível também a versão original dessa versão atualizada `https://github.com/runtimeverification/rv-android`.

Referências

[1] Bao, Lingfeng, Tien-Duy B. Le e David Lo: *Mining sandboxes: Are we there yet?* Em *SANER*, páginas 445–455. IEEE Computer Society, 2018. 1, 2, 17

[2] Costa, Francisco Handrick da, Ismael Medeiros, Thales Menezes, João Victor da Silva, Ingrid Lorraine da Silva, Rodrigo Bonifácio, Krishna Narasimhan e Márcio Ribeiro: *Exploring the use of static and dynamic analysis to improve the performance of the mining sandbox approach for android malware identification.* J. Syst. Softw., 183:111092, 2022. 1, 18

[3] Jin, Dongyun, Patrick O'Neil Meredith, Choonghwan Lee e Grigore Rosu: *Javamop: Efficient parametric runtime monitoring framework.* Em *ICSE*, páginas 1427–1430. IEEE Computer Society, 2012. 1, 5, 6

[4] Jamrozik, Konrad, Philipp von Styp-Rekowsky e Andreas Zeller: *Mining sandboxes.* Em *ICSE*, páginas 37–48. ACM, 2016. 3, 4

[5] Daian, Philip, Yliès Falcone, Patrick Meredith, Traian Florin Şerbănuţă, Shin'ichi Shiriashi, Akihito Iwai e Grigore Rosu: *Rv-android: Efficient parametric android runtime verification, a brief tutorial.* Em *Runtime Verification*, páginas 342–357. Springer International Publishing, 2015. 4, 8, 9, 11

[6] Chen, Feng, Dongyun Jin, Patrick O'Neil Meredith e Grigore Rosu: *Monitoring oriented programming - a project overview.* Em , 2009. 5, 6

[7] Chen, Feng, Marcelo d'Amorim e Grigore Rosu: *A formal monitoring-based framework for software development and analysis.* Em *ICFEM*, volume 3308 de *Lecture Notes in Computer Science*, páginas 357–372. Springer, 2004. 5, 7

[8] Tarr, Peri L., Harold Ossher, William H. Harrison e Stanley M. Sutton Jr.: N *degrees of separation: Multi-dimensional separation of concerns.* Em *ICSE*, páginas 107–119. ACM, 1999. 7

[9] Falcone, Yliès e Sebastian Currea: *Weave droid: aspect-oriented programming on android devices: fully embedded or in the cloud.* Em *ASE*, páginas 350–353. ACM, 2012. 7