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

Instrumentação RVSec para Android
Um Relato de Experiência

Pedro Luis Chaves Rocha

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

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2023

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

Instrumentação RVSec para Android
Um Relato de Experiência

Pedro Luis Chaves Rocha

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

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

Jonata Teixeira Pastro
CIC/UnB

Pedro Costa
CIC/UnB

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

Brasília, 25 de junho de 2023

Dedicatória

Dedico esse trabalho a todos os meus amigos e familiares que me apoiaram durante todo o trabalho. Também dedico a todos os envolvidos no desenvolvimento do RVSec Android que poderão utilizar desse trabalho como auxílio no desenvolvimento da ferramenta.

Agradecimentos

Primeiramente, gostaria de expressar minha gratidão ao meu orientador, Professor Rodrigo Bonifácio, e ao colega Pedro Costa que me apoiaram e guiaram durante todo o processo de desenvolvimento desse trabalho.

Também gostaria de agradecer a Universidade de Brasília por todos os recursos disponibilizados. O acesso aos artigos foi muito importante para os estudos teóricos.

Por fim, também gostaria de agradecer meus familiares e a minha namorada pelo apoio e compreensão durante essa jornada.

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 uso de ferramentas de análise dinâmica se tornou comum para detectar determinados tipos de problemas em software em domínios que envolvem setores críticos da sociedade, como aplicações financeiras ou relacionados à saúde. Isso ocorre devido à dificuldade das ferramentas de análise estática em identificar certas classes de problemas, tendo um alto percentual de falsos positivos. Um exemplo significativo é a identificação de *Cripto API Misuses*, que ocorre quando os programadores utilizam de maneira incorreta bibliotecas de criptografia.

Para solucionar esse problema, diversos estudos estão sendo conduzidos para criar ferramentas capazes de identificar o uso incorreto de APIs de criptografia de forma automática. Um exemplo de estratégia que possui esse propósito é o RVSec [1] que utiliza uma técnica chamada de instrumentação via *Runtime Verification* para analisar dinamicamente programas em busca de falhas relacionadas à utilização incorreta de APIs de criptografia em aplicações Java. Nessa técnica, trechos de código responsáveis pela análise dos problemas desejado são inseridos no programa original. Assim, aproveitando desse estudo e da possibilidade de convertermos quaisquer aplicações Android em Java utilizando engenharia reversa, um grupo de pesquisadores da Universidade de Brasília desenvolveu uma nova ferramenta chamada de RVSec Android.

No decorrer do desenvolvimento dessa ferramenta, os pesquisadores se depararam com fato de que a solução conseguia instrumentar com sucesso um percentual reduzido de aplicativos. Desta forma, neste trabalho, conduzimos uma análise detalhada dos erros mais comuns na ferramenta, realizando várias pesquisas e experimentos para coletar mais informações e encontrar possíveis soluções para os problemas identificados. Dentre os resultados obtidos, confirmamos a existência de uma incompatibilidade da ferramenta com aplicativos implementados em Kotlin e descartamos a hipótese de que os erros eram resultado de possíveis incompatibilidades com a SDK do Android. Além disso, também identificamos que 3 dos 23 arquivos de especificações utilizados para analisar problemas de uso de bibliotecas de criptografia são responsáveis por cerca de 30% de todos os erros apresentados.

Também propomos ações que podem ser tomadas pelos desenvolvedores do RVSec

Android para continuar a busca pelos problemas. Dentre elas estão propostas de trabalhos futuros relacionados aos demais erros ao aprofundamento dos resultados obtidos nesse trabalho. Alternativamente, existe a possibilidade de abandonar o uso de ferramentas de engenharia reversa e direcionar os esforços para o desenvolvimento de novas ferramentas que tenham o mesmo objetivo, mas com um foco exclusivo em aplicativos Android. De maneira geral, os resultados desse trabalho podem ser analisados pelos responsáveis pela ferramenta que poderão tomar melhores decisões a respeito do desenvolvimento e dos próximos passos a serem tomados.

Palavras-chave: análise dinâmica, uso incorreto de APIs de criptografia, RVSec, Android

Abstract

The use of dynamic analysis tools has become common to detect certain types of problems that involve critical sectors of society, such as financial applications or those related to health. This occurs due to the difficulty of static analysis tools in identifying this problem, having a high percentage of false positives. A significant example is the *Crypto API Misuse*, which occurs when programmers improperly use cryptography libraries.

To solve this problem, various studies are being conducted to create tools capable of automatically identifying it. An example of software with this purpose is RVSec [1], which uses a technique called instrumentation to dynamically analyze programs in search of faults related to the incorrect use of cryptography APIs in JAVA applications. In this technique, snippets of code responsible for analyzing the desired problems are inserted into the original program. Thus, taking advantage of this study and the possibility of converting any Android applications into Java using reverse engineering, a group of researchers from the University of Brasília developed a new tool called RVSec Android.

During the development of this tool, the researchers found that the application could not successfully instrument even half of the tested applications on it. In this way, in this work, we conducted a detailed analysis of the most common errors in the tool, conducting some research and experiments to gather more information and find possible solutions to the identified problems. Among the results obtained, we confirmed the existence of a tool incompatibility with applications made in Kotlin and discarded the hypothesis that the errors were due to the use of incorrect versions of the Android SDK by the tool. In addition, we also identified that 3 out of the 23 specification files used to analyze problems of use of cryptography libraries are responsible for about 30% of all the errors presented.

We also propose actions that can be taken by those responsible for RVSec Android to continue searching for the problems. Among them are proposals for future work related to the remaining errors and a deepening of the results obtained in this work. Alternatively, there is the possibility of abandoning the use of reverse engineering tools and directing efforts to the development of new tools that have the same objective, but with an exclusive focus on Android applications. In general, the results of this work should also be analyzed by those responsible for the tool, who will be able to make better decisions regarding

development and the next steps to be taken.

Keywords: dynamic analisis, crypto API misuse, RVSec, Android

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ADB Android Debug Termianl.

AOP Aspect oriented Programming.

API Application Interface.

APK Android Application Package File.

APKs Android Application Package Files.

DEX Davilk Executable.

JAVA MOP Java Monitoring Oriented Programming.

JDK Java Development Kit.

RV Runtime Verification.

SDK Software Development Kit.

Capítulo 1

Introdução

Runtime Verification (RV) é uma técnica utilizada para testar o comportamento de um programa considerando o comportamento observado durante a execução dos programas. Estudos já demonstraram que existem situações em que esse tipo de abordagem é bem mais confiável do que análise estática [2] [3] [4] [1], que é conduzida apenas observando o código (fonte ou executável) dos programas, sem executar os programas em si. Assim, ferramentas que utilizam essa técnica estão se popularizando cada vez mais em sistemas tolerantes falhas como *blockchains* [5], aonde erros podem gerar problemas graves para os usuários e empresas.

Existem diversas formas de se implementar ferramentas de análise dinâmica. A mais comum é a utilizando o que chamamos de monitores, módulos ou componentes de código que observam a execução do programa. Esse tipo de programa pode ser criado utilizando uma técnica chamada de instrumentação [6], aonde os monitores são inseridos no código original, sem a necessidade de alterações do código original por parte dos programadores. Dessa forma, o código injetado também é executado durante o uso da aplicação para exibir, em formato de logs ou exceções, os problemas detectados [7].

Um dos propósitos de Runtime Verification (RV) é auxiliar programadores provendo informações sobre problemas que são difíceis de detectar apenas com uma análise estática. RV pode também complementar análise estática em alguns cenários, como, por exemplo, apoiar ferramentas de análise estática na identificação do uso incorreto de bibliotecas de criptografia [2] [3] [4] [1]. Esse é um exemplo bem claro, pois existem formas de se utilizar tais bibliotecas que estão sintaticamente e semanticamente corretas, mas que também são vulneráveis a falhas de segurança, como a utilização de uma chave criptográfica de maneira errada.

Para resolver este problema, alguns pesquisadores vêm explorando o aumento da confiabilidade dos sistemas com uma implementação de RV, chamada JAVA MOP, responsável por instrumentar um código Java com monitores escritos para detectar essas falhas. Para

definir as regras que serão testadas pelos monitores, são criadas especificações, com uma sintaxe descrita pelo próprio framework, aonde é possível definir de maneira formal, regras e comportamentos com as chamadas do programa original que serão observadas e validadas pelo framework. Assim, os eventos recebidos são interpretados conforme as especificações e processados em relatórios de uso gerados durante a execução do programa.

Hoje em dia, o framework JAVA MOP também é utilizado para identificar problemas de utilização de APIs de criptografia em aplicativos escritos em Java por meio da ferramenta RVSec [1]. Nesse projeto foram feitas traduções, para a sintaxe compreendida pelo JAVA MOP, de regras formais criadas para detectar problemas chamados de *Crypto API Misuse* [4].

Tendo o RVSec [1] como referência, pesquisadores da Universidade de Brasília trabalharam em um projeto chamado de RVSec Android, que possui como objetivo utilizar o RVSec [1], juntamente com diversas outras ferramentas para monitorar um aplicativo Android e identificar os problemas de uso de bibliotecas de criptografia presentes nos mesmos. No entanto, a solução proposta atualmente não funciona para maioria dos aplicativos nas quais é testado, tendo sua execução interrompida com diversos erros.

Dessa forma, como o percentual de aplicativos instrumentados é muito baixo para uma ferramenta criada para testar softwares com potenciais riscos de segurança, neste trabalho serão levantados e analisados os principais problemas apresentados pela execução do RVSec Android. Assim, os pesquisadores envolvidos no desenvolvimento dessa ferramenta terão maior clareza a respeito dos motivos por trás das falhas e poderão tomar melhores decisões relacionadas ao andamento do projeto. Além disso, ao decorrer dos experimentos também serão propostas sugestões de melhorias para a ferramenta visando aumentar o percentual de sucessos de utilização do RVSec Android.

No capítulo seguinte (2) trataremos sobre os trabalhos relacionados ao RVSec, trazendo uma análise sobre a vulnerabilidade conhecida como *Crypto API Misuse* e a utilização de *Runtime Verification* para o teste e monitoramento de aplicações Android com esse tipo de problema. Depois, essa monografia reservará o capítulo 3 para descrever todo o ambiente utilizados para testes e as configurações das ferramentas que compõem o RVSec Android, utilizadas para se obter os problemas encontrados atualmente na ferramenta. Já no capítulo 4 trataremos a respeito do contexto do RVSec Android e quais os problemas encontrados no estágio atual de desenvolvimento juntamente com os experimentos e resultados das análises descritas. Por fim, no capítulo final (5), teremos a conclusão do trabalho considerando os resultados obtidos juntamente com sugestões a respeito do andamento do projeto.

Capítulo 2

Trabalhos relacionados

Nesse capítulo traremos uma revisão da literatura dos principais referenciais teóricos utilizados para desenvolver o RVSec Android. Isso inclui uma breve análise de ferramentas de análise dinâmica e o estado da arte do problema conhecido como *Crypto API Misuse*. Além disso, também abordaremos algumas das ferramentas utilizadas pelo RVSec Android para obtermos um pouco de contexto a respeito do desenvolvimento e utilização desses programas.

2.1 Crypto API Misuse em aplicativos Android

Um dos problemas encontrados em aplicações que utilizam bibliotecas criptográficas está estritamente relacionado ao uso errado de suas interfaces e métodos. Isso pode acontecer quando o programador não possui o conhecimento necessário para utilizar a API disponibilizada, deixando o software vulnerável a diversas falhas de segurança [8]. Em seu estudo, Egele et al. [3] utilizou 11,748 aplicativos da Google Play para estudar o uso de bibliotecas de criptografia em aplicativos Android. Como resultado, ele demonstrou que cerca de 88% das aplicações testadas possuíam pelo menos um problema relacionado a má utilização de tais bibliotecas, ou seja, é um problema mais comum do que realmente aparenta ser.

Hoje em dia, a maioria dos estudos realizados sobre o problema de *Crypto APIs Misuse* [2] [3] [4], [9] [10] são feitos com base em uma análise quantitativa das aplicações, ou seja, estudando a quantidade e a frequência de ocorrência dos erros e vulnerabilidades. Nesses tipos de análise, são normalmente utilizadas métricas como percentual de aplicações vulneráveis ou quantidade de problemas encontrados por aplicação, mas não consideram o contexto e a gravidade das vulnerabilidades encontradas nos softwares testados.

Tendo isso em vista, Wickert et al. [11] em seu trabalho fizeram uma análise qualitativa de tais vulnerabilidades, ou seja, analisando a criticidade ao invés da quantidade de

ocorrências. Dessa forma, foi possível compreender de forma mais aprofundada os efeitos das vulnerabilidades causadas pelo mau uso das APIs de criptografia e as consequências para as empresas e clientes afetados. Nesse estudo, foi criado um modelo que utiliza como base os ataques possíveis a aplicações vulneráveis, e a seriedade de cada uma deles. Como resultado, o modelo utilizado classificou quase metade das vulnerabilidades testadas como críticas. Portanto, o *Crypto APIs Misuse*, além de ser extremamente comum, também expõe os aplicativos vulneráveis a falhas de segurança graves.

Ademais, a aplicação da tecnologia em áreas e processos com baixa tolerância a falhas se tornou bem comum atualmente. Os softwares desenvolvidos estão cada vez mais presentes em setores como saúde ou economia. Um dos principais motivos para isso é a criticidade dos dados e transações operadas por essas empresas, que podem impactar diretamente na vida e a saúde de milhares de pessoas. Tendo isso em vista, diversas ferramentas foram criadas para detectar esses problemas e avisar os desenvolvedores a respeito de possíveis vulnerabilidades criptográficas: *CrySL* [4], *CryptoLint* [3], *CryptoGuard* [9], *Crylogger* [10]. Assim, com o auxílio dessas ferramentas, essas empresas poderão se proteger de forma mais eficaz de hackers mal-intencionados.

Como uma solução para o problema apresentado, o *CrySl* [4] descreve um conjunto de especificações formais a respeito de como funções de criptografia devem ser utilizadas de maneira segura. Além disso, é proposto um framework chamado *CogniCryptSAST* que utiliza essas regras propostas no artigo em uma ferramenta de análise estática para aplicações Java e Android. Vale lembrar que, como as especificações são propostas de maneira formal e extensível, elas podem ser traduzidas e utilizadas por outras ferramentas que possuem o mesmo propósito, como foi feito no desenvolvimento do RVSec Android que será abordado neste trabalho.

2.2 Análise Estática e Dinâmica

Além das especificações formais a respeito do *Crypto APIs Misuse*, também é importante determinar como as ferramentas de análise vão utilizar essas regras. Tendo isso em vista, existem diversas técnicas hoje em dia para identificar problemas em um software, categorizadas em 2 grupos: análise estática e análise dinâmica.

A análise estática procura os problemas no programa por meio de técnicas de análise sintática e semântica diretamente no código dos programas—sem que os programas sejam executados [4] [3] [9], por considerarem apenas os arquivos escritos do programa. Um problema bem comum relacionado com essas ferramentas é a quantidade de falsos positivos [12], pois muitas vezes são criadas para buscar por erros genéricos e não levam e conta

todo o contexto de execução do programa. Mesmo assim, são muito utilizadas hoje em dias pela facilidade de uso e agilidade dos feedbacks a respeito dos problemas.

Por outro lado, as ferramentas de análise dinâmica, precisam ser utilizadas durante a execução do programa e procuram verificar a interação entre as chamadas internas do programa. Assim, as informações utilizadas para os testes são mais próximas da realidade, ao considerar todo o contexto de execução como: dados, variáveis, chamadas a funções e até mesmo o sistema operacional e a máquina utilizada. Essa diferença faz com que esse tipo de ferramenta seja mais eficiente e precisa para detectar erros mais sensíveis ao contexto do programa, como o é o caso do *Crypto APIs Misuse* [2] [3] [4] [1].

2.3 Runtime Verification

Runtime Verification [13] [14] é uma estratégia de análise dinâmica que consiste em observar a execução do programa em busca de eventos que correspondam às especificações definidas pelo programador. Essa análise é feita ao adicionar, ao software original, trechos de código chamados de monitores que monitoram chamadas específicas contendo as informações desejadas. Assim, essas informações são interpretadas e processadas por um conjunto de regras formais criadas pelo programador em busca de problemas no fluxo de execução do software.

Além disso, os monitores são normalmente criados com o propósito de serem totalmente modulares e desacoplados, pois não podem interferir no funcionamento original da aplicação. Um exemplo de ferramenta que faz parte do grupo de Runtime Verification é o framework Java Monitoring Oriented Programming (JAVA MOP) [15], o qual Meredith et al. descreveram como essa ferramenta implementa sua própria linguagem de especificações para monitoramento de código Java. Esse framework usa o AspectJ e todo o contexto de Aspect oriented Programming (AOP) para a inserção de trechos de código responsáveis por monitorar o programa [16].

Assim, é possível utilizar quaisquer conjuntos de regras formais para detecção de *Crypto APIs Misuse* [2] [3] [4] para monitorar o uso de APIs de criptografia em aplicações em Java. Além disso, como as regras estão descritas formalmente, podemos utilizar outras linguagens de programação ou especificação para implementar o monitoramento das regras. Um exemplo disso é o próprio RVSec [1], que utilizava o JAVA MOP e as regras formais descritas no CrySl [4] para detectar problemas com o uso de bibliotecas de criptografia em programas Java.

2.4 AspectJ

AspectJ é uma extensão da linguagem de programação Java que implementa formas de se programar utilizando um paradigma chamado Aspect oriented Programming (AOP). Esse paradigma permite aumentar a modularidade do código por meio da separação de responsabilidades em pequenos módulos chamados de aspectos. Esses aspectos, mesmo que aumentem a modularidade do código, são normalmente utilizados para aplicações bem específicas como registro de logs, tratamento de exceções e validações, por dificultar a compreensão do código me projetos maiores. Isso acontece por conta da modularização, que torna o fluxo de execução do programa mais imprevisível e não linear.

Tendo isso em vista, os aspectos são responsáveis por encapsular comportamentos específicos que podem ser injetados em classes ou métodos de maneira independente e sem alterar o arquivo original. Essa injeção de código é feita a partir do que chamamos de *pointcuts*, expressões que visam identificar pontos de código a partir de suas definições, também conhecidas como *joinpoints*. Assim, quando um trecho de código correspondente a uma *pointcut* é encontrado pelo AspectJ, o código associado a ela, chamado de *advice* é inserido no original para adicionar o comportamento desejado.

Portanto, como essas ferramentas e estudos são utilizados no RVSec Android para identificar problemas de *Crypto APIs Misuse*, elas também serão alvo de investigação durante este trabalho em busca de possíveis pontos de falha na utilização com aplicativos Android. Portanto, no próximo capítulo entraremos mais a fundo na estrutura do RVSec Android e em como ele usa as ferramentas discutidas nessa revisão de literatura para identificar possíveis problemas de segurança.

Capítulo 3

RVSec Android - Arquitetura de Instrumentação

Neste capítulo, será discutido um pouco a respeito do progresso atual do desenvolvimento do RVSec Android. Além disso, será feito um resumo de todas as ferramentas utilizadas e suas configurações de modo que seja possível replicar os cenários descritos ao longo deste trabalho. Por fim, teremos uma seção sobre o conjunto de aplicativos utilizados nos testes que trará características de cada um que irá compor nosso dataset para os experimentos.

3.1 RVSec Android

O RVSec Android surgiu como uma forma de adaptar, para o Android, uma ferramenta já existente chamada de RVSec [1]. Em suma, o RVSec tem como propósito instrumentar aplicações escritas em Java com um código responsável por verificar se as bibliotecas de criptografia estão sendo utilizadas da maneira correta. Assim, o RVSec Android se aproveita do fato que é possível transformar bytecode Android em bytecode Java, em ambos os sentidos. Isso permite reusar a mesma estratégia de instrumentação do RVSec tradicional em aplicativos Android.

Para entender o RVSec Android, primeiro, temos que entender o que são e como funcionam os arquivos DEX. Atualmente, a grande maioria dos sistemas operacionais para dispositivos mobile são baseados em Android. Tais sistemas foram criados para executar um formato de arquivo específico, chamado de Davilk Executable (DEX). Esses arquivos são gerados a partir de processos de *build* presentes nas ferramentas de montagem de aplicativos, como o D8, que convertem bytecode Java para esse formato compatível com sistema operacional do Android. Além disso, estes arquivos também são empacotados

juntamente com arquivos de recursos (imagens, sons e metadados) no que chamamos de Android Application Package Files (APKs)[1].

A primeira etapa do RVSec Android consiste em descompactar um arquivo APK expondo seu código executável em formato DEX juntamente com todos os recursos e metadados. Tendo esses arquivos separados em uma pasta, utilizamos uma ferramenta de engenharia reversa (*dex2jar*) que consegue transformar arquivos DEX em arquivos que empacotam bytecode Java (arquivos .jar). Tais arquivos são então descompactados, expondo os bytecodes, também conhecidos *class files* por conta de sua extensão (.class).

Tendo o código da aplicação extraído e convertido em classes, o RVSec Android utiliza o framework Java Monitoring Oriented Programming (JAVA MOP) [15] para o processo de instrumentação. Nessa fase o código responsável por monitorar a ocorrência de problemas relacionados a má utilização das APIs de criptografia [4], [1] é injetado no programa original. De forma resumida, esse processo acontece da seguinte maneira. Primeiro, as especificações no formato (.mop) são processadas pelo JAVA MOP, gerando os aspectos (.aj) e os monitores(.rvm). Após esse processamento, os monitores, gerados anteriormente, são convertidos para o formato Java (.java) por uma ferramenta chamada de RV-Monitor. Por fim, com todos esses arquivos, o AspectJ mescla os aspectos e os monitores no código original, como mostra a Figura 3.1.

Por fim, a última etapa da execução no RVSec Android consiste em pegar todo o código JAVA instrumentado pelo JAVA MOP, convertê-lo de volta no formato DEX e remontar o arquivo do aplicativo (APK). Atualmente esse processo de remontagem do aplicativo a partir do código em Java é feito por uma ferramenta chamada *D8*, presente no kit de desenvolvimento (SDK) do Android. Esse programa recebe as classes Java como parâmetro e gera o arquivo DEX correspondente em seu retorno.

Após todos esses processos, é possível instalar e utilizar o aplicativo em qualquer dispositivo Android compatível. Assim, enquanto a aplicação é utilizada, o código adicionado faz as verificações de como as chamadas às bibliotecas de criptografia estão se comportando. No caso do RVSec Android, o monitoramento é feito por meio de logs do Android Debug Termianl (ADB). Esses logs são capturados e escritos em arquivos de textos por meio de um script Python criado pelos desenvolvedores do RVSec Android para facilitar o processo de execução e testes dos aplicativos utilizando emuladores.

[1]Note que esses nomes também estão relacionados com as extensões dos arquivos (.dex e .apk).

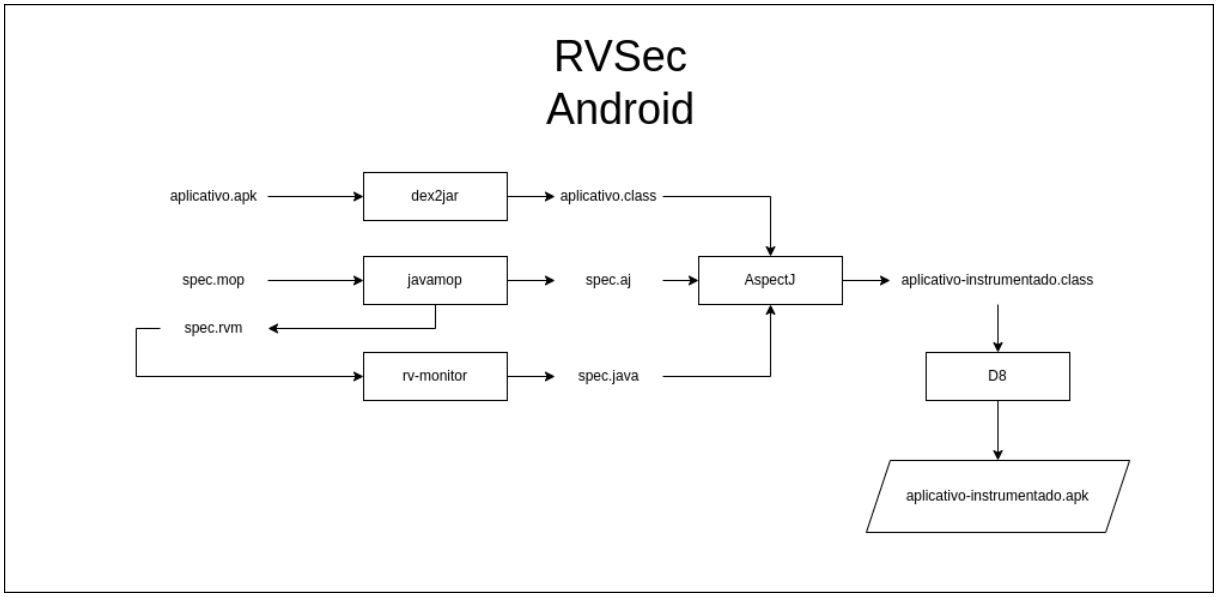

Figura 3.1: Estrutura do RVSec Android.

3.2 Especificações MOP

A especificação MOP é onde são escritas as regras e comportamentos que serão monitodos durante a execução do programa. Para descrever tais comportamentos, as especificações possuem 3 partes principais:

Events

Expressões que definem quais elementos do código serão observados durante a execução. Definidas nos arquivos *.mop* por meio do prefixo *event*, podem descrever chamadas a métodos, acessos a variáveis, tratamentos de erros, entre outros. No exemplo abaixo definimos 2 eventos que serão utilizados pela especificação. O primeiro representa todas vezes que um método com nome A é chamado e o segundo representa todas vezes que um método com nome B é chamado.

```
1    event methodACalled: call(* *.A(..));
2    event methodBCalled: call(* *.B(..));
```

Properties

Expressões que definem sequências ou grupos de *events*. Essas expressões são escritas por meio de algum tipo de formalismo como: expressões regulares, gramáticas livres de contexto ou lógicas temporais. São elas que irão definir quais comportamentos são

desejados ou não durante a execução do programa por meio dessas regras formais. No exemplo abaixo, a propriedade determina que todas as chamadas do método B devem ocorrer após o método A, já definidos como *events*.

```
1    property: methodACalled methodBCalled;
```

Handlers

Trechos de código Java executado quando as *properties* avaliam se as sequências de eventos descrita é válida ou não. Sua função principal é permitir que alguma ação seja feita a partir das verificações feitas, como, por exemplo, registrar as falhas em um arquivo de log ou imprimir no terminal. No exemplo abaixo, podemos observar um tipo de *handler* que é executado toda vez que propriedade, definida anteriormente, é violada. Em outras palavras, caso em algum momento o método A seja executado após o B, uma mensagem no console de execução será mostrada avisando do ocorrido.

```
1    violation {
2        System.out.println("Metodo B chamado apos o A");
3    }
```

Neste trabalho, não entraremos em detalhes a respeito de como cada uma das especificações criadas para o RVSec funcionam [1]. No entanto, existe um ponto relevante para o decorrer do trabalho. Cada uma das 23 especificações utilizadas atualmente no RVSec Android buscam monitorar um problema descrito no *CrySl* [4]. Porém, algumas delas necessitam de dados gerados por outras especificações para funcionarem, ou seja, elas possuem dependências entre sí. Assim, como não existe uma forma de representar dependências na sintaxe do JAVA MOP, foi criado um contexto de execução para que essas especificações possam armazenar e utilizar dados obtidos por outras durante a execução do programa. Esse contexto é acessado por meio da seguinte importação:

```
1    ...
2    import br.unb.cic.mop.ExecutionContext;
3    ...
```

3.3 Ambiente Experimental

Para padronizarmos os experimentos, será utilizado um ambiente de testes com as ferramentas e versões descritas ao longo dessa seção para evitar variabilidade nos resultados por conta de diferenças no sistema ou nas versões. Além disso, algumas dessas ferramentas já serão atualizadas para suas versões mais novas.

Para o sistema operacional será utilizado um Ubuntu 22.04 com os seguintes softwares instalados:

- Java OpenJDK (1.8.x)
- Android SDK Platform (33)
- Android Build Tools (33.0.1)
- AspectJ (1.9.2)
- Apache Maven (3.6.3)

Para a instalação da SDK do Java, foi utilizado uma ferramenta chamada *sdkman*. Essa ferramente é um gerenciador de versões para diversos kits de desenvolvimento criados para sistemas baseados em *linux*. Como os testes foram executados no Ubuntu, após a instar o *sdkman*, basta instalar o Java 8 com o comando:

```
1    sudo sdk install java 1.8.0-tem
```

Esse comando instala uma versão de código aberto do Java Development Kit (JDK) chamado *Temurin* mantido pela *Eclipse Foundation*, muito conhecida pela sua estabilidade e compatibilidade. Atualmente, essa não é a versão mais nova do Java presente no mercado, mas é a versão necessária para que algumas ferramentas utilizadas pelo RVSec Android funcionem, como no caso do JAVA MOP. Além disso, também é necessário instalar o *Maven*, um gerenciador de pacotes para projetos escritos em Java utilizado para montar as dependências do RVSec Android. Esse software pode ser instalado com o seguinte comando:

```
1    sudo apt install maven
2    mvn clean install -DskipMopAgent -DskipTests
```

Já no caso das ferramentas da SDK do Android, podemos instalá-las diretamente pela interface do *Android Studio* ou manualmente a partir do *sdkmanager*. Esse programa está presente no conjunto de ferramentas para linha de comando para desenvolvimento Android (*Android Command Line Tools*) que também pode ser baixada no site do Android Studio[2].

Tendo a SDK do Android instalada, basta obter o *Android SDK Platform* e o *Android Build tools* com os seguintes comandos[3]:

[2]Url: `https://developer.android.com/studio#command-tools`, acessado em 21 de junho de 2023
[3]Note que o local da instalação pode variar dependendo do método utilizado

```
export ANDROID_HOME="/usr/lib/android-sdk"
export PATH="$ANDROID_HOME/cmdline-tools/latest/bin:$PATH"
export PATH="$ANDROID_HOME/emulator:$PATH"

sdkmanager --install "build-tools;33.0.0"
sdkmanager --install "platforms;android-33"
```

O último programa necessário para executar o RVSec Android é o AspectJ que pode ser instalado com o gerenciador de pacotes do Ubuntu (APT). Nesse trabalho utilizamos a versão 1.9.2 que pode ser instalada com o seguinte comando:

```
sudo apt-get update
sudo apt-get install aspectj=1.9.2
```

Além disso, é necessário aumentar o limite de memória utilizado pelo AspectJ, pois o valor que vem como padrão é menor do que o necessário para instrumentar os aplicativos com as especificações utilizando o JAVA MOP. Nesse caso utilizaremos um valor de 10240M ou 10G de memória para a máquina virtual java no lugar dos 64M que vinham por padrão. Esse aumento pode ser feito ao abrir o arquivo presente no diretório *$ASPECTJ_HOME/bin/ajc* e alterar o parâmetro *Xmx64m* para *Xmx10240m*:

```
...
exec "${JAVACMD:=java}" -classpath "$AJPATH${CLASSPATH:+:$CLASSPATH}" \
  "${JAVA_OPTS:=-Xmx10240m}" \
  org.aspectj.tools.ajc.Main "$@"
```

Uma vez que temos essas ferramentas instaladas, conseguimos instrumentar os aplicativos combinando as ferramentas: *javamop* e *rv-monitor*, presentes no RVSec Android. Com os comandos abaixo, conseguimos processar as especificações MOP. O primeiro é responsável por gerar os aspectos e os monitores (.rvm) e o segundo utiliza esses monitores e os converte em código (.java).

```
...
javamop -d $MOP_OUT_DIR -merge $MOP_DIR/*.mop
rv-monitor -merge -d $MOP_OUT_DIR $MOP_DIR/*.rvm
...
```

Por fim, para a análise dos resultados utilizaremos scripts escritos em Python para processar os dados de cada execução. Também usaremos a biblioteca *plotly* para gerar os gráficos necessários. Tais gráficos de grande importância tanto para comparar os resultados dos experimentos quanto para gerar as imagens que serão inseridas neste trabalho.

Capítulo 4

Posposta e Metodologia

Nesse capítulo abordaremos a respeito do que é o RVSec Android, suas principais características e ferramentas. Também, será discutido a respeito dos problemas que fazem a solução proposta atualmente não funcionar para alguns aplicativos. Por fim, definiremos a metodologia que será a utilizada para analisar os percentuais de erro e sucesso da ferramenta.

4.1 Objetivo do Estudo Empírico

O objetivo deste estudo empírico é categorizar os erros encontrados ao instrumentar aplicativos Android via o RVSec Android. Tendo o ambiente de testes descrito e padronizado, os problemas serão agrupados e analisados pela sua quantidade de ocorrências dentro do nosso dataset. Também olharemos para quantidade total de aplicativos instrumentados com sucesso para acompanhar a taxa de sucesso e erro da execução da ferramenta.

4.2 Aplicativos Usados nos Experimentos

Tendo o ambiente configurado e executando RVSec Android da forma correta, baixamos diversos aplicativos disponibilizados gratuitamente pela plataforma *F-Droid*. Essa plataforma possui um catálogo com diversas aplicações Android de código aberto agrupados por domínios, ou seja, por setores de atuação. Assim, para este trabalho, utilizamos uma seleção já feita pelos responsáveis do RVSec Android que consiste em:

- Desconsiderar aplicativos que não possuem atualizações a partir do ano de 2019.
- Selecionar aplicativos dos seguintes domínios: Financeiro, Segurança, Conectividade, Telefonia, Sistemas, presentes no *F-Droid*.

A ideia inicial de remover do dataset a aplicações atualizadas pela última vez antes de 2019 era facilitar o contato com os desenvolvedores responsáveis pelos aplicativos testados para estudar melhor os impactos do RVSec Android. No entanto, para o escopo deste projeto, essa filtragem terá como objetivo apenas de selecionar aplicativos com tecnologias mais recentes.

Já a seleção pelo domínio da aplicação tem como propósito apenas selecionar aplicativos impactados por soluções de segurança. Dessa forma, conseguimos aplicativos que utilizam APIs de criptografia de alguma forma para garantir a segurança dos dados utilizados.

Considerando os filtros citados acima, para realizar os experimentos deste trabalho, obtivemos um dataset com cerca de 365 aplicações. No entanto, dentro dessa lista existem aplicativos que utilizam versões muito antigas do Android que podem ser alvo de problemas e bugs que já foram corrigidos nas mais recentes.

Além disso, o RVSec Android é um programa com um tempo de execução que depende do tamanho do aplicativo e pode demorar aproximadamente 30 segundos na máquina utilizada para os testes. Assim, para reduzir o tempo gasto com os experimentos deste trabalho, reduziremos o tamanho deste dataset. Para isso, os aplicativos que utilizam versões da SDK do Android muito antigas ou muito discrepantes dos demais serão todos descartados. Assim, além de economizar tempo com os experimentos, também conseguimos filtrar problemas muito específicos ou incomuns dentro do dataset.

Para identificar as versões do SDK com que cada aplicativo foi criado, usamos uma ferramenta chamada *aapt* ou *Android Asset Packaging Tool*. Essa ferramenta, presente no *Android Build Tools*, consegue extrair diversas informações a partir do arquivo de instalação de um aplicativo Android (APK). No entanto, como essas informações são retiradas de um arquivo chamado de *AndroidManifest.xml*, criado durante a montagem do aplicativo, esses valores podem vir em formatos e nomes diferentes, pois dependem da ferramenta utilizada montar o APK. Porém, mesmo com essas dificuldades, ainda podemos obter o valor da versão utilizada procurando pelos seguintes campos no retorno do *aapt*[1] [2]:

- sdkVersion: Versão mínima do SDK necessária para rodar o aplicativo.
- targetSdkVersion: versão máxima do SDK para qual o aplicativo foi projetado e testado.
- maxSdkVersion: versão máxima do SDK compatível para rodar o aplicativo.

[1]Note que a versão utilizada no teste pode ser qualquer uma das 3 listadas, pois cada um desses valores podem estar presentes em alguns aplicativos e em outros não

[2]Também é importante lembrar que essa busca foi feita por meio de uma expressão regular, então caso mais de um valor esteja presente, será utilizada a ordem na qual foram retornados pelo aapt

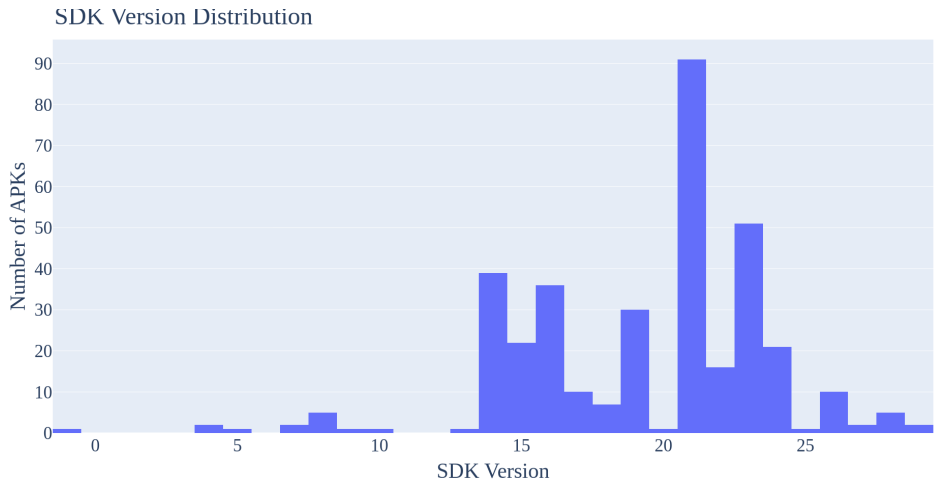

Figura 4.1: Versão mínima da SDK Android (sdkVersion) dos aplicativos presentes no dataset.

Portanto, tendo em vista a Figura 4.1 e os filtros descritos nessa seção, para termos maior confiabilidade nos resultados e maior velocidade na hora de realizar os experimentos, filtraremos os aplicativos que possuem versão de SDK maior que 16 e SDK menor que 23. Dessa forma, reduzimos o escopo e a variabilidade de busca dos problemas para apenas as versões mais utilizadas dentro no nosso dataset. Também excluímos todos os aplicativos nos quais o *aapt* não conseguiu obter os dados necessários por algum erro ou falta dos dados necessários. Assim, com essa filtragem, chegamos a um dataset com 240 aplicativos que serão instrumentados pelo RVSec Android nos experimentos documentados nos próximos capítulos.

4.3 Procedimentos de análise

Para a análise dos resultados, utilizaremos gráficos de setores gerados a partir dos logs obtidos da instrumentação do RVSec Android para cada um dos 240 aplicativos selecionados para o experimento. Para isso, os erros retornados devem ser categorizados e agrupados de forma que seja possível analisar a quantidade de ocorrências de cada um deles. Assim, foram escolhidas de forma empírica frases específicas que representem e agrupem cada um dos erros presentes nos logs obtidos após a execução, ou seja, cada uma dessas frases deve indicar um resultado possível da execução do RVSec Android dentro do dataset analisado. Essa abordagem irá facilitar a análise e estudo dos problemas, agrupando os aplicativos em categorias que podem ser comparadas entre diferentes execuções do experimento. Também é importante ressaltar que essas categorias foram obtidas como forma

de agrupar os problemas durante a análise, ou seja, os nomes não estão relacionados com os estudos abordados neste trabalho. Segue abaixo a lista dos grupos obtidos:

Grupo	Etapa do RVSec Android
Done! Final apk generated	Final
CIRCULAR REFERENCE:	D8
cannot cast the outer type to a reference type	Instrumentação
java.lang.StackOverflowError	Instrumentação
Mismatch when building parameterization map	Instrumentação
java.lang.IllegalStateException:	Instrumentação

Tabela 4.1: Grupos de resultados possíveis da execução do RVSec Android

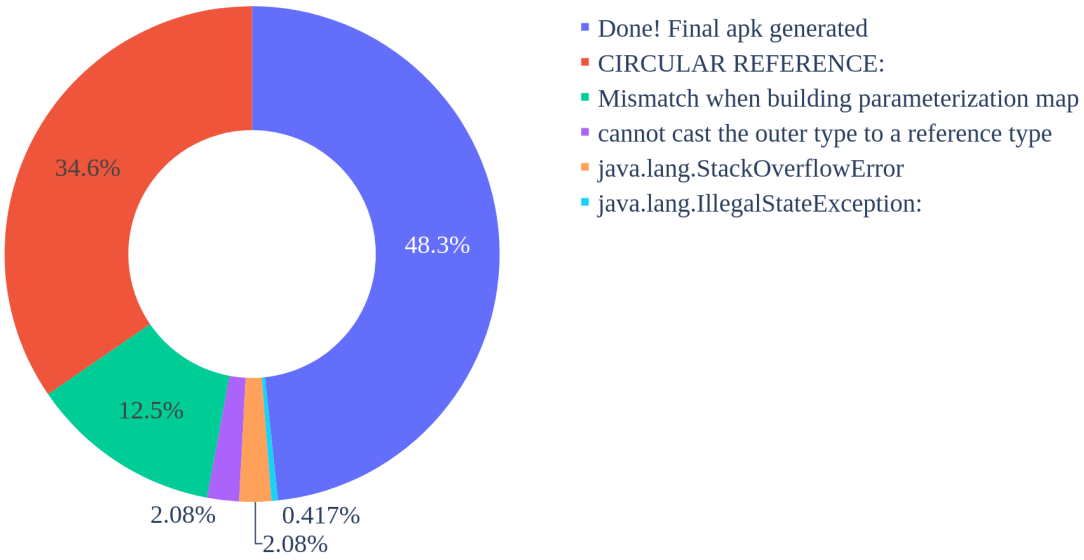

Figura 4.2: Percentual de Resultados Atuais do RVSec Android.

4.4 Resultados

4.4.1 Análise Preliminar

Agora que temos os principais erros que ocorrem ao executar o RVSec Android classificados, podemos olhar para os mais recorrentes e analisar as possíveis causas em busca de informações suficientes para formularmos as nossas hipóteses. Assim, como podemos observar na Figura 4.2, temos 2 erros principais que juntos somam 47,1% de todos os resultados obtidos durante a nossa primeira execução. A partir de tal consideração, para este trabalho analisaremos a fundo apenas esses 2 principais erros, por serem os que nos trarão maior retorno sobre o tempo investido.

- *CIRCULAR REFERENCE*: Erro que acontece após instrumentar o aplicativo durante fase de remontagem do DEX utilizando o D8. Ele é um erro do tipo *ArrayIndexOutOfBoundsException* que indica uma falha inesperada ao chamar o D8 para remontar aplicativo instrumentado. Após algumas pesquisas e testes, chegamos a conclusão que esse erro pode se um indício de algum bug no D8 ou no código injetado pelo AspectJ. Além disso, também encontramos a seguinte mensagem de erro: *Merging dex file containing classes with prefix j$* que, segundo as pesquisas realizadas, podem indicar problemas relacionados a um processo interno do D8 chamado de *desugaring*.
- *Mismatch when building parameterization map*: Erro que ocorre no *dex2jar*, durante a fase de engenharia reversa dos arquivos DEX. Um ponto importante a se notar é que a grande maioria desses erros estão relacionados a classes que possuem Kotlin no nome. Kotlin é uma linguagem de programação criada para ser executada na máquina virtual Java, mas, com foco em ambientes mobile. Assim, podemos assumir que esse erro está relacionado com aplicações desenvolvidas nessa linguagem, pois é pouco provável de encontramos classes com esses nomes em projetos criados puramente em Java.

4.4.2 Formulação de Hipóteses

Tendo em vista os erros de execução do RVSec Android, podemos propor algumas hipóteses a respeito de quais podem ser as possíveis causas desses problemas. Essas hipóteses serão utilizadas como guia para podermos analisar melhor as taxas de sucesso e erro da ferramenta a partir de modificações criadas para testar essas hipóteses. Assim, com base na análise preliminar dos estudos a respeito da plataforma do Android e do RVSec, chegamos as seguintes hipóteses:

- Hipótese 1: Sabemos que existem diversas versões do Android disponíveis e que nossos aplicativos foram criados em diversas delas, porém, na versão atual do RVSec Android, a versão da plataforma utilizada é apenas a 33. Isso pode ser um problema, pois o arquivo *android.jar* é utilizado pelo AspectJ para realizar a instrumentação. Assim, caso alguma dessas aplicações utilizem classes, novos recursos ou recursos depreciados do Android, problemas podem aparecer durante diversos pontos da execução do RVSec Android.
- Hipótese 2: Além da versão da plataforma do Android, também temos a versão utilizada para remontar o aplicativo com o D8. Essa ferramenta possui um parâmetro chamado *–min-api*, responsável por definir para qual versão do Android aquele

aplicativo está sendo montado. Atualmente o RVSec Android está utilizando o valor fixo de 21 para essa propriedade e pode estar relacionada com os erros na etapa de remontagem do aplicativo após a instrumentação.

- Hipótese 3: Como a maioria dos problemas ocorre durante a fase de instrumentação do código das especificações MOP, também não podemos descartar a possibilidade de ser alguma incompatibilidade entre o Android e as especificações MOP criadas para detectar problemas de *Crypto API Missuse*. Provavelmente, existe alguma funcionalidade utilizada pelo JAVA MOP ou pelas especificações que não está sendo reconhecida pelo D8 ou pelo AspectJ.

Conduzimos um conjunto de experimentos com o intuito de comprovar ou rejeitar as hipóteses descritas acima. Esses experimentos foram compostos em uma sequência de execuções do RVSec Android, com modificações pensadas para testar cada uma dessas hipóteses, para todos os 240 aplicativos presentes no dataset. Assim, foi possível utilizar os logs de cada execução, juntamente com os percentuais de sucesso e erro categorizado no início deste capítulo, para registrar e comparar os valores com os experimentos anteriores.

Além dos gráficos de setores com os percentuais de cada resultado da execução, como duas de nossas hipóteses são relativas à versão da SDK, também utilizaremos um gráfico do tipo *heatmap* para melhor visualizar a distribuição das versões Android para cada resultado obtido. Nesse gráfico, o eixo X representa cada resultado possível, descrito no início da seção de Metodologia, o eixo Y representa a versão da SDK do Android e o eixo Z a quantidade de aplicativos que estejam na interseção de cada uma das propriedades presentes nos eixos X e Y. A distribuição atual da ferramenta sem nenhuma modificação está representada na Figura 4.4.

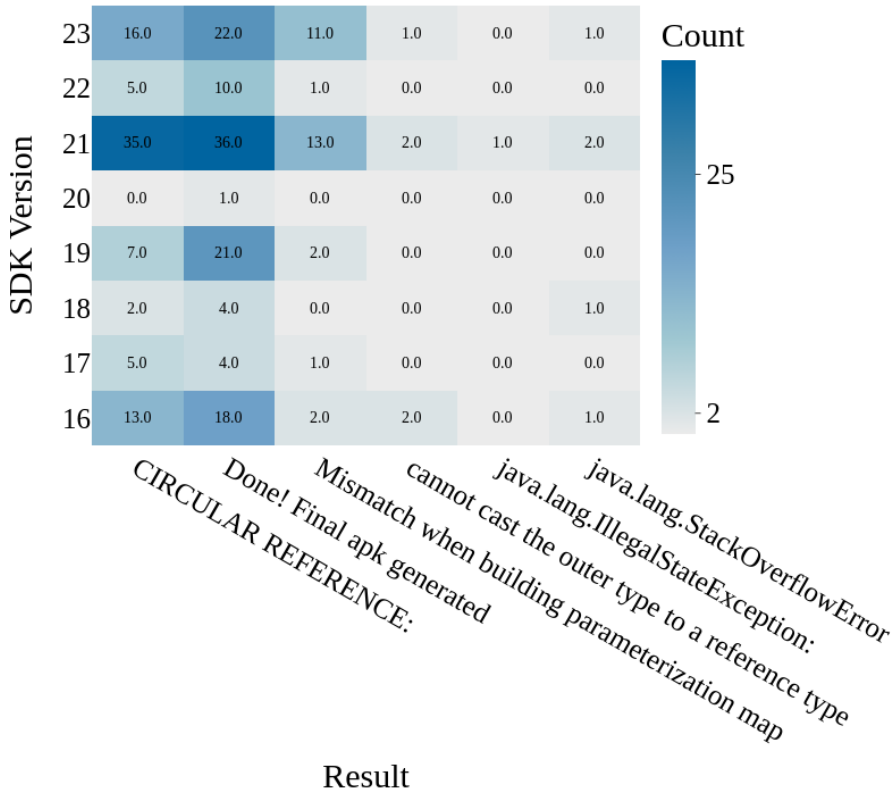

Figura 4.3: Quantidade de Aplicativos por Resultados Possíveis e Versões de SDK atuais do RVSec Android.

4.4.3 Experimentos Realizados

Tendo em vista as hipóteses formuladas anteriormente, podemos descrever quais experimentos foram conduzidos para confirmar ou refutá-las. Como este é um trabalho de exploração e análise dos problemas atuais do RVSec Android, inicialmente realizamos três experimentos. Esses experimentos possuem como objetivo obter um ponto de partida para análises mais profundas que são descritas mais a frente neste trabalho.

Experimento 1

Para testar a primeira hipótese, realizaremos a execução do RVSec Android passando uma versão dinâmica da SDK a partir da utilizada originalmente para a criação do aplicativo. Essa versão pode ser obtida a partir da ferramenta *aapt* como mostrado no Capítulo 3. Além disso, é importante que tenhamos todas as versões do Android possíveis dentro do nosso dataset, da 16 à 23, instaladas:

```
for i in {16..23}
do
    sdkmanager --install "platforms;android-$i"
done
```

Dessa forma, podemos selecionar exatamente qual a versão da SDK desejamos o RVSec utilize para cada um dos aplicativos instrumentados. Com o retorno dessas execuções salvos em arquivos de logs, podemos classificar novamente os erros para comparar os gráficos obtidos no experimento com os presentes na análise preliminar.

Experimento 2

Já para a segunda hipótese, podemos reutilizar o mesmo código para obter a versão da SDK da mesma forma descrita no primeiro experimento. Porém, esse valor será passado como argumento para o D8 na propriedade *–min-api* para que ele possa remontar o aplicativo com versão do Android adequada após os processos de engenharia reversa e instrumentação. Também é importante lembrar que será utilizada a versão 33.0.0 do D8 presente no pacote de ferramentas *Android Build Tools*, pois diversos bugs são corrigidos durantes as atualizações.

Com o ambiente de experimentos preparado com as novas modificações, repetimos os processos de obtenção e processamento de dados já explicados nesse capítulo para comparar novamente com os dados obtidos na análise preliminar. Além disso, dependendo dos resultados obtidos no experimento 1, as mudanças propostas nele deverão ser utilizadas também, pois, ambas tratam de possíveis problemas relacionados com a versão do Android. Em outras palavras, as mudanças propostas nesse experimento podem necessitar das mudanças conduzidas no experimento 1 para eliminar os erros que ocorrem durante a fase de montagem do aplicativo com o D8.

Experimento 3

Por fim, para a terceira hipótese, utilizamos subconjuntos diferentes de especificações MOP visando identificar possíveis problemas de incompatibilidade entre o Android e as especificações MOP criadas para detectar problemas de *Crypto API Misuse*. O principal ponto que motivou a criação dessa hipótese é que o framework JAVA MOP não foi criado com o propósito de instrumentar código Java gerado a partir da engenharia reversa de aplicativos Android. Assim, podem existir features utilizadas por alguma especificação que não são suportadas pelo ambiente de desenvolvimento do Android ou por alguma ferramenta utilizada no processo de desenvolvimento dos aplicativos.

O primeiro subconjunto é a utilização apenas da especificação chamada *MessageDigestSpec.mop*. Essa especificação foi escolhida por não depender de nenhuma outra que está sendo utilizada nos testes. Assim, a ideia é utilizar a menor unidade de especificações possíveis para identificar diferenças nos erros apresentados após a execução do RVSec Android.

O segundo subconjunto é composto apenas pelas especificações mais importantes: *MessageDigestSpec.mop*, *CipherSpec.mop* e a *MacSpec.mop*. No entanto, como elas dependem de outras especificações, essas dependências também devem ser incluídas no subconjunto. Com essas especificações, poderemos incrementar um pouco a quantidade de verificações já testadas no subconjunto descrito anteriormente buscando alguma alteração nos percentuais de aplicativos instrumentados com sucesso pela ferramenta.

4.5 Resultados quantitativos para os testes de hipótese

Como não sabemos como serão os resultados dos experimentos, existe a possibilidade de que todas as hipóteses sejam refutadas. Isso ainda seria um resultado positivo para este trabalho, pois abriria espaço para uma discussão a respeito do projeto e quais os próximos passos a serem seguidos. Também abriria a possibilidade para outros trabalhos testarem os demais erros ou formulem novas hipóteses.

Já no caso de obtermos pelo menos uma hipótese confirmada, realizaremos outros experimentos de caráter exploratório buscando obter mais informações a respeito dos problemas identificados. O objetivo dessa abordagem é aproveitar os cenários analisados pelos primeiros experimentos para buscar as razões por trás de cada uma das hipóteses confirmadas. Além disso, também descartaremos os cenários referentes às hipóteses refutadas, por indicarem que as mudanças conduzidas não são responsáveis pelos erros apresentados.

Após a execução dos experimentos que tinham como objetivo validar as hipóteses relacionadas a versão da SDK do Android, observamos que não houve mudanças nas taxas de sucesso e nem nos erros obtidos após a execução do RVSec Android, como mostra a Figura 4.4. É importante ressaltar que, como filtramos algumas versões do Android durante a amostragem dos aplicativos, esses resultados são válidos somente no intervalo descrito no Capítulo 3.

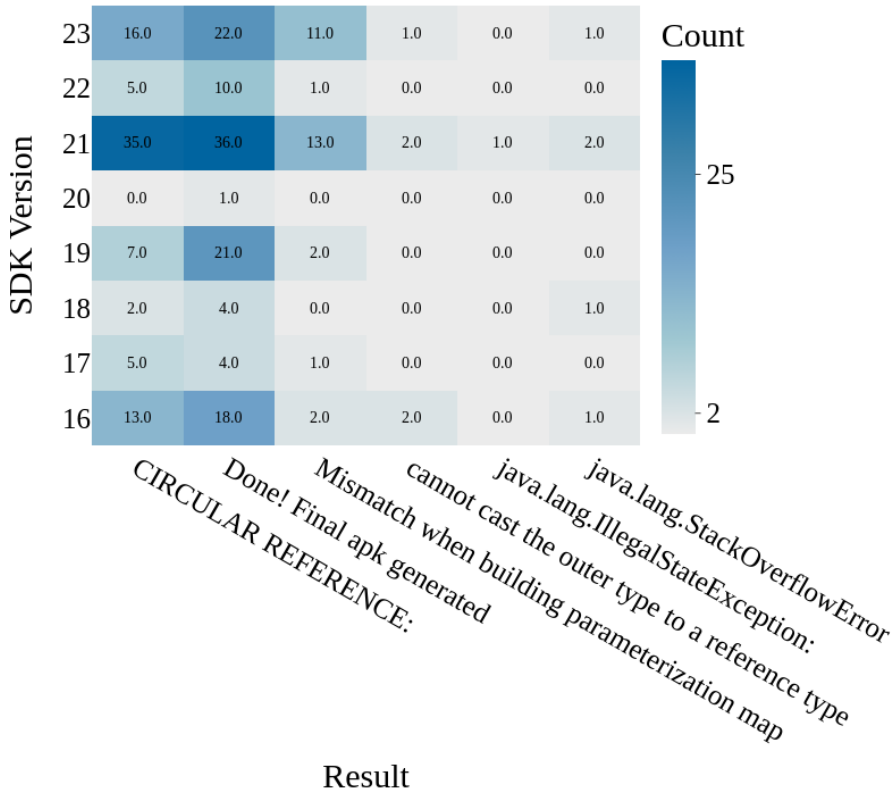

Figura 4.4: Distribuição dos erros presentes em cada versão do Android com as melhorias descritas nos experimentos .

Para os testes que buscavam alterações percentuais nas taxas de sucesso e erro da ferramenta ao utilizar subconjuntos das especificações MOP, também obtivemos resultados bem promissores. Ao utilizar somente a especificação *MessageDigestSpec.mop* notamos uma mudança significativa nos percentuais de sucesso obtidos como resultado. Como observado na Figura 4.5, os erros do tipo *CIRCULAR REFERENCE* diminuíram de 34.6% para 5% e o percentual de sucesso subiu de 48.3% para 75%. Também podemos observar na Figura 4.5 que apareceu um novo tipo de erro que possui como mensagem principal a seguinte frase: *Cannot fit request classes in a single dex file.* No entanto, ele será ignorado no restante deste trabalho, pois a quantidade de ocorrências dessa mensagem de erro é muito baixa comparada com as demais.

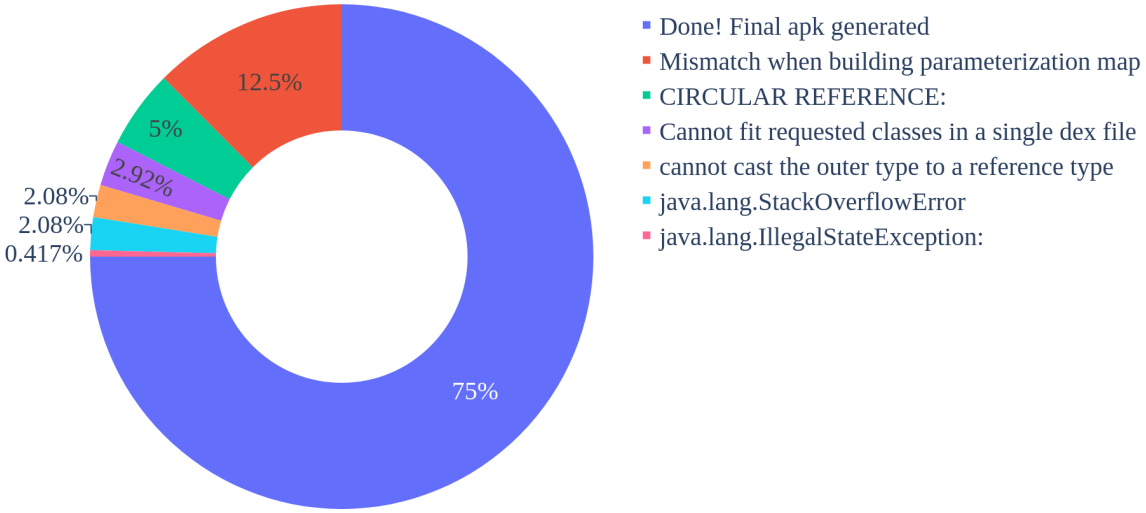

Figura 4.5: Percentual de resultados do RVSec Android apenas com a especificação *MessageDigestSpec*.

No segundo subgrupo de especificações também obtivemos resultados semelhantes. Primeiramente encontrado um pequeno erro com a especificação *CipheSpec.mop* na qual não possuía a importação Java de uma classe que era utilizada. Como essa dependência era importada por alguma outra especificação, esse problema não era visível anteriormente, pois o JAVA MOP junta todas as especificações em apenas um arquivo. Assim, tendo ajustado esse pequeno problema de execução, conseguimos executar o RVSec Android utilizando o subconjunto das especificações descritas junto com suas dependências. De maneria geral os resultados foram quase os mesmos obtidos apenas com a especificação *MessageDigestSpec.mop*, com exceção de 2 aplicativos que voltaram a apresentar o erro *CIRCULAR REFERENCE*, resultando em uma taxa de sucesso de 74.2%. Os aplicativos que tiveram os resultados alterados foram os seguintes:

- org.freenetproject.mobile_306.apk
- com.gaurav.avnc_7.apk

4.6 Discussão

Tendo em vista os resultados apresentados na seção anterior, conseguimos dados bastante relevantes para o desenvolvimento do RVSec Android e para a continuação deste trabalho. Assim, nessa seção, iremos analisá-los brevemente para chegar a conclusões mais

assertivas e que possam ser utilizadas pelos desenvolvedores do RVSec Android em seus trabalhos futuros. Durante essa análise conduziremos novos experimentos para detalhar os resultados obtidos anteriormente e aprofundar nos pontos de destaques de cada um deles. Além disso, esses experimentos serão descritos durante essa seção e seguirão uma ordem cronológica, pois dependem dos resultados uns dos outros.

O primeiro ponto de destaque é a ausência de alterações nos resultados obtidos pelos experimentos voltados a alteração das versões da SDK do Android. Tal resultado refuta as hipóteses 1 e 2 propostas no Capítulo 4 e indica que os problemas não estão relacionados com a versão do Android em si. Esse resultado pode não parecer tão relevante em um primeiro momento, pois refuta nossas duas primeiras hipóteses. No entanto, é uma informação muito valiosa para os responsáveis pelo RVSec Android, pois evitará que sejam colocados esforços desnecessários nessa direção.

Como segundo destaque dos resultados, temos o aumento do percentual de sucesso do RVSec Android ao reduzir o conjunto de especificações MOP instrumentadas. Esse resultado é muito relevante, pois indica que uma parte dos erros estão relacionados diretamente com os monitores criados pelo JAVA MOP. Assim, é possível que exista alguma funcionalidade utilizada pelo JAVA MOP ou pelas próprias especificações que seja incompatível com o D8 ou alguma outra ferramenta utilizada pelo RVSec Android.

Para obter mais informações sobre esse aumento no percentual de sucesso, foram realizados alguns experimentos de caráter exploratório para compreender melhor a situação. Esses experimentos consistem ir adicionando conjuntos de especificações gradativamente para observar as mudanças nas taxas de sucesso e erro dos experimentos. Dessa forma, analisaremos cada um dos subgrupos utilizados e, caso existam, quais são os testes responsáveis pelas diferenças nos resultados.

Na primeira análise utilizamos todas as especificações que se relacionam diretamente com a *CipheSpec.mop* e a *MessageDigestSpec.mop*. Dessa forma, criamos um grupo de testes semelhante ao que já foi documentado com algumas adições. A lista dos primeiro subgrupo a ser analisado, com 11 das 23 especificações, pode ser observada abaixo.

- CipherInputStreamSpec.mop
- CipherOutputStreamSpec.mop
- CipherOutputStreamSpec.mop
- CipherSpec.mop
- KeyGeneratorSpec.mop
- KeyPairGeneratorSpec.mop
- KeyPairSpec.mop

- KeyStoreSpec.mop
- MacSpec.mop
- MessageDigestSpec.mop
- HMACParameterSpecSpec.mop

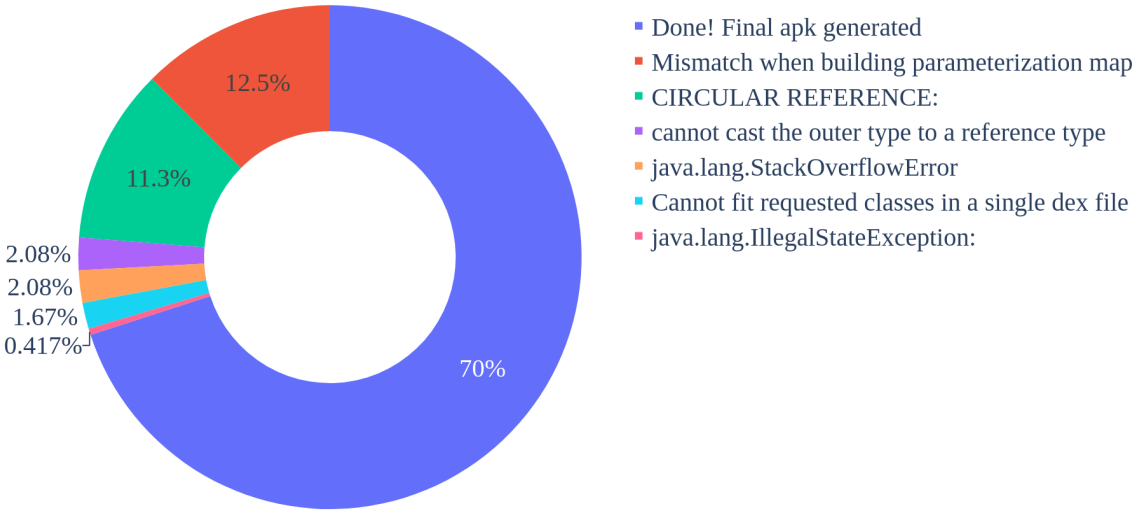

Figura 4.6: Percentual de Resultados do RvSec Android Para o Primeiro Subgrupo da Análise .

Após a execução da instrumentação do RVSec Android com as especificações listadas acima, houve uma pequena redução na taxa de sucesso da ferramenta, caindo para 70%, como mostra a Figura 4.6. Além disso, os erros que voltaram a aparecer foram de grupos variados e não representam mudanças que justifiquem considerar alguma das especificações utilizadas como defeituosa. Assim, para o segundo experimento da análise, utilizaremos os mesmos arquivos MOP da primeira parte acrescidos dos arquivos listados abaixo, resultando em um conjunto com 17 das 23 especificações.

- DHGenParameterSpecSpec.mop
- GCMParameterSpecSpec.mop
- RandomStringPassword.mop
- SecretKeySpec.mop
- SecretKeySpecSpec.mop

- SecureRandomSpec.mop

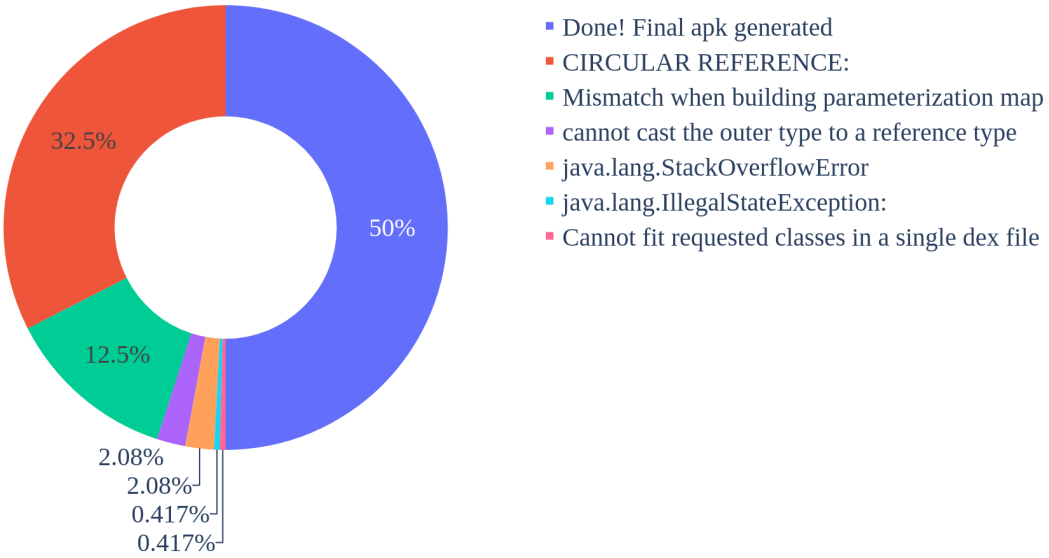

Figura 4.7: Percentual de Resultados do RvSec Android Para o Primeiro Subgrupo da Análise .

Ao contrário do resultado da primeira análise, com esse novo grupo de especificações, o percentual de instrumentação caiu de 70% para 50%, como mostra a Figura 4.7. Esse resultado indica um possível problema com alguma das especificações adicionadas. Ao testá-las individualmente, com um dos aplicativos que apresentaram essa mudança, observamos que a especificação *SecretKeySpecSpec.mop* é a principal responsável pela queda no percentual de aplicativos instrumentados. Para finalizar, essa especificação foi testada apenas com suas dependências em um experimento que resultou em uma taxa de sucesso de apenas 53%, como mostra a Figura 4.8. A análise do porquê dessa especificação não funcionar corretamente não será feita neste trabalho e pode ser uma ideia utilizada em trabalhos futuros.

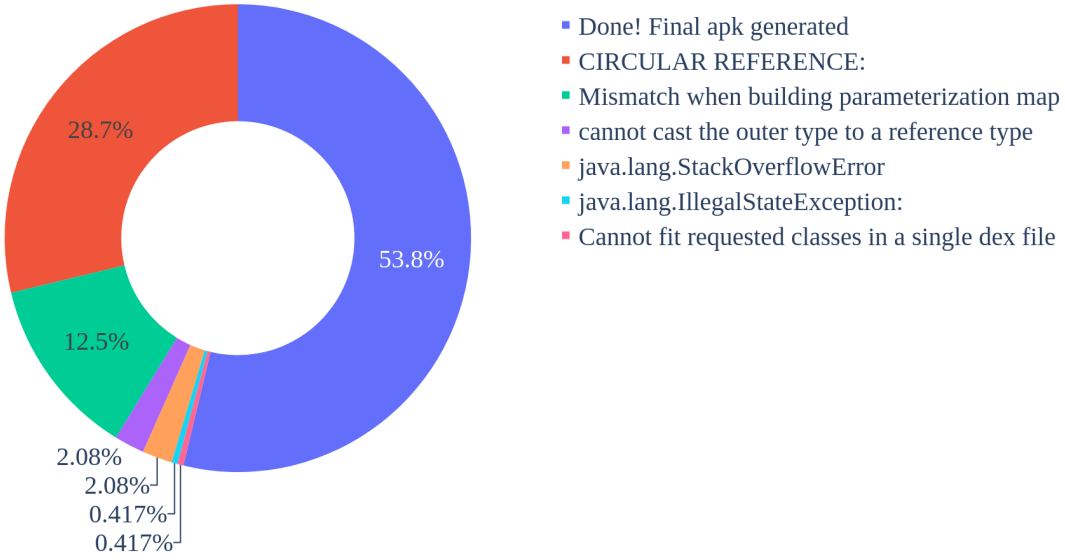

Figura 4.8: Percentual de resultados do RvSec Android para a espesificação *SecretKeySpecSpec.mop* e suas dependências.

Após alguns testes e verificações, notamos que a especificação *SecretKeySpecSpec.mop* não é a única responsável pela diferença observada anteriormente. Isso foi notado ao gerar um gráfico para calcular o ganho percentual real relativo ao estado inicial do RVSec Android. No experimento realizado para gerar esse gráfico, foram testadas todas as especificações com exceção apenas da defeituosa. Nos resultados desse experimento notamos que a ferramenta conseguiu instrumentar com sucesso cerca de 58.3% do dataset, 10% a mais do o obtido inicialmente com a ferramenta, mas ainda longe dos 70% obtidos com o subgrupo anterior e justifica a continuidade dessa análise[3].

Analisando o restante das especificações, identificamos que os arquivo *IvParameterSpec.mop* e *SecureRandomSpec.mop* também são responsáveis por um percentual elevado dos erros obtidos. Para este caso, mesmo que existam especificações que dependem delas, executaremos alguns experimentos removendo cada uma delas, apenas para observar o impacto nos resultados[4]. Primeiramente, foram removidos, os arquivos *SecretKeySpecSpec.mop* e *IvParameterSpec.mop*, pois já possuímos os resultados removendo apenas a primeira. Após a execução, obtivemos um percentual de sucesso de 61.7%, 13.4% a mais do que os 48.3% obtidos com todos os monitores. Depois, também removemos o

[3]Vale lembrar que, mesmo com essas diferenças, esse resultado ainda é bem relevante, pois a remoção dessa especificação sozinha foi capaz reduzir o total de erros em aproximadamente 19%.

[4]Note que as especificações que dependem delas provavelmente não irão funcionar corretamente, mas os resultados de montagem do aplicativo continuam válidos.

SecureRandomSpec.mop e conseguimos uma taxa de sucesso de 66.7%, como mostra a Figura 4.9.

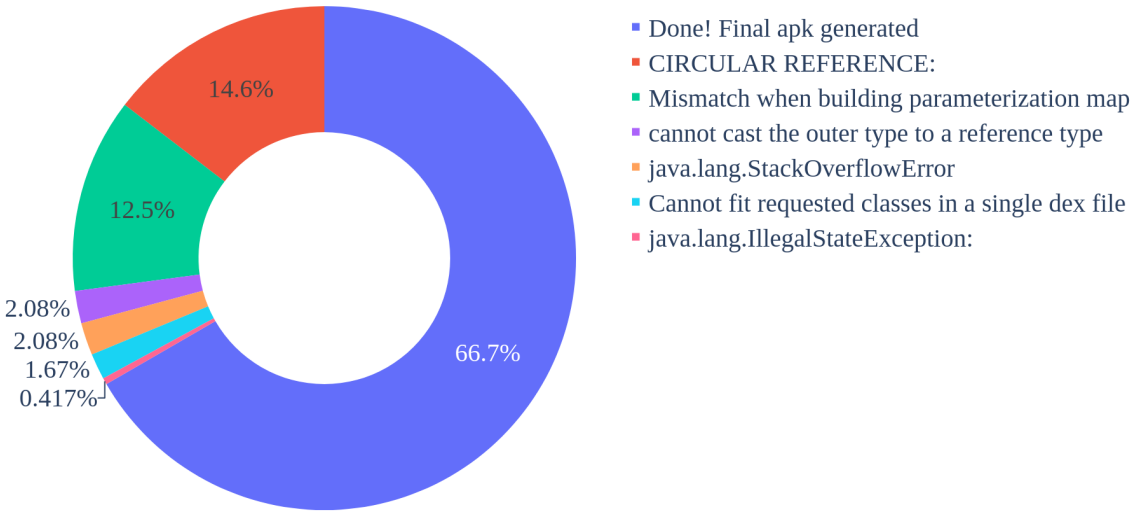

Figura 4.9: Resultados do RVSec Android sem as espesificações defeituosas.

Diante do exposto, obtivemos resultados bem satisfatórios com relação aos problemas apresentados atualmente pelo RVSec Android. Mesmo com pequenas variações não explicadas nos percentuais, conseguimos identificar que existem 3 especificações MOP responsáveis por um percentual notável dos erros apresentados. Também confirmamos que ao remover essas 3 especificações, mesmo que não seja a solução ideal para o contexto da ferramenta, conseguimos elevar a taxa de sucesso de 48.3% para 66.7%, uma melhora de aproximadamente 30%.

Capítulo 5

Conclusões

Como discutidos nos trabalhos relacionados, a utilização de análise dinâmica é mais eficiente que a estática para identificar problemas criados pelo mal uso de bibliotecas criptográficas. Torres et al. [1] utilizaram especificações formais criadas para identificar esses problemas [4] e converteram-nas para o padrão utilizado pelo framework JAVA MOP. No entanto, a primeira versão da ferramenta adaptada para aplicativos Android possuía uma taxa de sucesso menor que 50%, muito baixa para uma ferramenta cujo propósito é identificar problemas de segurança.

Ademais, o objetivo desse trabalho é analisar e identificar possíveis causas para os problemas apresentados atualmente pelo RVSec Android. Com esses dados, os responsáveis pelo projeto terão informação o suficiente para trabalhar em possíveis soluções. Além disso, as decisões envolvendo o projeto serão tomadas com maior assertividade e objetividade considerando os resultados obtidos.

5.1 Resultados

Durante o decorrer desse trabalho foram realizadas diversas análises e experimentos em busca de respostas para os problemas apresentados pelo RVSec Android. Alguns deles foram resultados de pesquisas na internet e outros por meio de experimentação e análise dos resultados. Dessa forma, chegamos nos seguintes resultados como contribuição principal desse trabalho:

- Classificamos e categorizamos os erros apresentados atualmente na execução do RVSec Android.
- Detectamos que uma parcela dos erros ocorrem devido à incompatibilidade da ferramenta com aplicativos feitos em Kotlin.

- Descartamos a hipótese de que a versão da SDK do Android era a responsável pela maioria erros.
- Identificamos que as especificações *SecretKeySpecSpec.mop*, *SecureRandomSpec.mop* e *IvParameterSpec.mop* são responsáveis por aproximadamente 30% dos erros obtidos durante os experimentos.

5.2 Sugestões

Assim, concluímos que o uso de uma ferramenta de análise dinâmica pode ser utilizada para detectar problemas gerados pelo mau uso de APIs de criptografia em aplicativos Android. Porém, a abordagem atual possui diversos desafios como a falta de incompatibilidade que possuímos hoje em dia com ferramentas de engenharia reversa utilizadas e o JAVA MOP.

Com os resultados desse trabalho, identificamos cenários que diminuem o impacto da incompatibilidade entre as ferramentas utilizadas e aumentam a taxa de sucesso do RVSec Android. Também identificamos possíveis restrições que podem ser aplicadas à ferramenta para evitar esses problemas como: não utilizar com aplicativos escritos em Kotlin ou para versões muito antigas do Android. Além disso, tendo os resultados obtidos durante os experimentos e análises, conseguimos propor algumas sugestões:

- Remover as especificações problemáticas. No entanto, isso vai contra o propósito da ferramenta que é detectar problemas de segurança e ainda seria necessário continuar explorando os motivos dos demais erros.
- Compreender o motivo pela qual as especificações problemáticas não funcionam para alguns dos aplicativos. Esse tipo de análise, além de corrigir os problemas atuais, também ajudaria a compreender o restante deles. Porém, essa é uma abordagem tanto quanto arriscada, pois custará muita energia e pode não dar o retorno desejado.
- Continuar a análise em busca de outros motivos para os erros além dos estudados neste trabalho. Sabemos que o Ambiente Android é cheio de restrições e peculiaridades, por executar em sistemas e arquiteturas diferentes do Java convencional. Ou seja, voltar os estudos para uma compreensão maior de como o Android em si e seus aplicativos funcionam.

Por fim, outra alternativa é buscar ou desenvolver outros softwares capazes de instrumentar os aplicativos com a mesmas regras, mas diretamente nos arquivos DEX, ou seja, sem utilizar engenharia reversa. Esse tipo de abordagem foi pouco explorada atualmente e pode ser um tema bem interessante para trabalhos futuros.

Referências

[1] Torres, Adriano: *Rvsec: Runtime verification methods for high precision detection of cryptography api misuse*, 2022. v, vii, 1, 2, 5, 7, 8, 10, 29

[2] Gao, Jun, Pingfan Kong, Li Li, Tegawende F. Bissyande e Jacques Klein: *Negative results on mining crypto-api usage rules in android apps.* IEEE International Working Conference on Mining Software Repositories, 2019-May:388–398, maio 2019, ISSN 21601860. 1, 3, 5

[3] Egele, Manuel, David Brumley, Yanick Fratantonio e Christopher Kruegel: *An empirical study of cryptographic misuse in android applications.* Proceedings of the ACM Conference on Computer and Communications Security, páginas 73–83, 2013, ISSN 15437221. `https://dl.acm.org/doi/10.1145/2508859.2516693`. 1, 3, 4, 5

[4] Kruger, Stefan, Johannes Spath, Karim Ali, Eric Bodden e Mira Mezini: *Crysl: An extensible approach to validating the correct usage of cryptographic apis.* IEEE transactions on software engineering, 47(11):2382–2400, 2021, ISSN 0098-5589. 1, 2, 3, 4, 5, 8, 10, 29

[5] Ganguly, Ritam: *Runtime verification for blockchains.* Proceedings of the IEEE Symposium on Reliable Distributed Systems, 2021-September:347–348, 2021, ISSN 10609857. 1

[6] Kempf, Torsten, Kingshuk Karuri e Lei Gao: *Software Instrumentation*, páginas 1–11. John Wiley Sons, Ltd, 2008, ISBN 9780470050118. `https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470050118.ecse386`. 1

[7] Chen, Boyuan e Zhen Ming Jack Jiang: *A survey of software log instrumentation.* ACM Computing Surveys (CSUR), 54, maio 2021, ISSN 15577341. `https://dl-acm-org.ez54.periodicos.capes.gov.br/doi/10.1145/3448976`. 1

[8] Nadi, Sarah, Stefan Kruger, Mira Mezini e Eric Bodden: *Jumping through hoops: Why do java developers struggle with cryptography apis?* Proceedings - International Conference on Software Engineering, 14-22-May-2016:935–946, maio 2016, ISSN 02705257. 3

[9] Rahaman, Sazzadur, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu e Danfeng (Daphne) Yao: *Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects.* Em *Proceedings of the 2019 ACM SIGSAC Conference on computer and communications security*, CCS '19, páginas 2455–2472. ACM, 2019, ISBN 9781450367479. 3, 4

[10] Piccolboni, Luca, Giuseppe Di Guglielmo, Luca P Carloni e Simha Sethumadhavan: *Crylogger: Detecting crypto misuses dynamically*. Em *arXiv.org*, Ithaca, 2020. Cornell University Library, arXiv.org. 3, 4

[11] Wickert, Anna Katharina, Lars Baumgärtner, Michael Schlichtig, Krishna Narasimhan e Mira Mezini: *To fix or not to fix: A critical study of crypto-misuses in the wild*. setembro 2022. `https://arxiv.org/abs/2209.11103v2`. 3

[12] Johnson, Brittany, Yoonki Song, Emerson Murphy-Hill e Robert Bowdidge: *Why don't software developers use static analysis tools to find bugs?* Em *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, página 672–681. IEEE Press, 2013, ISBN 9781467330763. 4

[13] Bartocci, Ezio, Yliès Falcone, Adrian Francalanza e Giles Reger: *Introduction to runtime verification*. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 10457 LNCS:1–33, 2018, ISSN 16113349. `https://link.springer.com/chapter/10.1007/978-3-319-75632-5_1`. 5

[14] Leucker, Martin e Christian Schallhart: *A brief account of runtime verification*. The Journal of Logic and Algebraic Programming, 78:293–303, maio 2009, ISSN 1567-8326. 5

[15] Meredith, Patrick O'Neil, Dongyun Jin, Dennis Griffith, Feng Chen e Grigore Roşu: *An overview of the mop runtime verification framework*. International journal on software tools for technology transfer, 14(3):249–289, 2012, ISSN 1433-2779. 5, 8

[16] Kiczales, Gregor, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean Marc Loingtier e John Irwin: *Aspect-oriented programming*. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 1241:220–242, 1997, ISSN 16113349. `https://link.springer.com/chapter/10.1007/BFb0053381`. 5