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

Análise de benchmarks de CryptoAPI misuses para validação de análise dinâmica de código

Saulo M. Feitosa

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

Orientador
Prof. Dr. Rodrigo Bonifácio

Brasília
2022

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

Análise de benchmarks de CryptoAPI misuses para validação de análise dinâmica de código

Saulo M. Feitosa

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

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

Prof. Dr. Edna Canedo
Universidade de Brasília

Dr. Walter Mendonça
Universidade de Brasília

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

Brasília, 16 de maio de 2022

Dedicatória

Dedico o trabalho aqui apresentado aos meus pais Rosália e Feitosa que sempre me apoiaram e também aos meus amigos Adelson, Diego, José, Daniel, Matheus, Divino, Breno, Gabriel, João Pedro, João Victor, Lucas, Vinicius e Maria, que conheci durante esta jornada e me ajudaram a chegar até aqui.

Agradecimentos

Agradeço ao meu orientador Rodrigo Bonifácio e a seu mestrando Adriano Torres por me deixarem participar de sua pesquisa e me darem os caminhos para percorrer durante a escrita deste trabalho.

Resumo

O uso incorreto de APIs criptográficas pode levar a falhas de segurança no desenvolvimento de software, por isso, surge a necessidade de ferramentas que detectem o maior número possível de vulnerabilidades, seja utilizando análise dinâmica ou estática. Este documento visa descrever o processo de seleção e os desafios enfrentados na utilização de um *benchmark* para validar o desempenho de uma ferramenta capaz de detectar vulnerabilidades criptográficas utilizando análise dinâmica no uso da JCA(Java Cryptography Architecture) e também comparar os resultados obtidos por esta ferramenta ao executar o *benchmark* escolhido com os obtidos por ferramentas que realizam análise estática ao executar o mesmo *benchmark*.

Palavras-chave: análise dinâmica, análise estática, APIs criptográficas, benchmark

Abstract

The misuse of cryptographic APIs can lead to major security errors in software development, that is why we need tools able to detect as many vulnerabilities as possible, either in Runtime verification or in a static way. This document has the objective of describe the process of selection and usage of a benchmark to validate the performance of a tool that is able to detect cryptographic vulnerabilities in the use of the JCA (Java Cryptography Architecture) using *runtime verification* and also compare the results obtained by this new tool with those obtained by tools that perform static analysis.

Keywords: runtime verification, static verification, benchmark, vulnerabilidades criptográficas, Cryptographic APIs

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Vulnerabilidades criptográficas são causadas por usos incorretos de APIs de criptografia dentro do código fonte de algum software. Existem diversas ferramentas que realizam análise estática de código buscando por tais vulnerabilidades, esse tipo de análise não requer que um programa seja executado, ao invés disso a análise é feita em uma versão do código e muitas regras de segurança abstratas são redutíveis a propriedades concretas do programa, que são executáveis por meio de técnicas genéricas de análise estática [1]. Por tanto surge a questão, é possível identificar tais vulnerabilidades realizando uma análise em tempo de execução? Ferramentas desse tipo são baseadas em tempo de execução, onde a execução de um sistema de software é checada dinâmicamente em relação as especificações formais, o programa sendo monitorado é instruído a capturar eventos como chamadas de métodos e atualização do valor de atributos, assim, em tempo de execução o programa cria monitores que verificam se os eventos estão de acordo com as especificações e relatam violações quando alguma especificação é violada [2]. Este documento é parte de uma pesquisa mais ampla onde desenvolveu-se uma ferramenta visando responder a pergunta proposta neste parágrafo.

Tal ferramenta foi planejada para fazer análises dinâmicas, ou seja, em tempo de execução e desenvolvida utilizando JavaMOP, o único sistema de monitoramento paramétrico que permite vários formalismos lógicos diferentes. É também o mais eficiente em termos de sobrecarga de tempo de execução e muito competitivo em relação ao uso de memória [3]. Na primeira fase do desenvolvimento foram escritas as especificações dos usos incorretos da JCA que se desejava capturar. Na segunda fase, foi feito um estudo empírico de tais especificações, onde para fins de validação, selecionou-se um benchmark que pudesse ser executado sobre elas, afim de comparar em termos de *precision* e *recall* os erros capturados com os obtidos por analisadores estáticos. Em relação aos analisadores estáticos, foram selecionados o CogniCrypt [4] e o CryptoGuard [5]. O benchmarck selecionado foi o CryptoAPI-Bench [6], um abrangente benchmark sobre usos incorretos da JCA, ele

contém 16 tipos de vulnerabilidades criptográficas e casos de usos corretos e incorretos. Os trechos de código na Figura 1.1 e Figura 1.2 ilustram exemplos de programas presentes nesse benchmark.

```java
package org.cryptoapi.bench.brokencrypto;

import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;

public class BrokenCryptoABICase1 {
    public void go(String crypto, String keyAlgo) throws NoSuchPaddingException, NoSuchAlgorithmException, InvalidKeyException {
        KeyGenerator keyGen = KeyGenerator.getInstance(keyAlgo);
        SecretKey key = keyGen.generateKey();
        Cipher cipher = Cipher.getInstance(crypto);
        cipher.init(Cipher.ENCRYPT_MODE, key);
    }

    public static void main (String [] args) throws NoSuchPaddingException, NoSuchAlgorithmException, InvalidKeyException {
        BrokenCryptoABICase1 bc = new BrokenCryptoABICase1();
        String crypto = "DES/ECB/PKCS5Padding";
        String keyAlgo = "DES";
        bc.go(crypto,keyAlgo);
    }
}
```

Figura 1.1: Exemplo 1: BrokenCryptoABICase1.java

```java
package org.cryptoapi.bench.predictablekeystorepassword;

import java.io.IOException;
import java.net.URL;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.cert.CertificateException;

public class PredictableKeyStorePasswordABICase1 {
    URL cacerts;
    public static void main(String args[]) throws KeyStoreException, IOException, CertificateException, NoSuchAlgorithmException {
        PredictableKeyStorePasswordABICase1 pksp = new PredictableKeyStorePasswordABICase1();
        String key = "changeit";
        pksp.go(key);
    }

    public void go(String key) throws KeyStoreException, IOException, CertificateException, NoSuchAlgorithmException {
        String type = "JKS";
        KeyStore ks = KeyStore.getInstance(type);
        cacerts = new URL("https://www.google.com");
        ks.load(cacerts.openStream(), key.toCharArray());
    }
}
```

Figura 1.2: Exemplo 2: PredictableKeyStorePasswordABICase1.java

1.1 Desafios na Integração do CryptoAPI-Bench na Pesquisa

Esta seção apresenta os desafios encontrados na integração deste benchmark.

1.1.1 Execução automática do benchmark no RVSec

Durante a primeira fase de desenvolvimento do RVSec foram escritos arquivos de teste para cada uma das especificações CrySL traduzidas. Tais arquivos continham trechos de código com casos de uso correspondentes a determinada especificação, isso foi feito para testar se as traduções feitas estavam sendo efetivas na captura de usos incorretos da JCA. Porém, a mesma estratégia não pôde ser aplicada para as classses do benchmark escolhido, visto que este era proposto para análise estática de código, portanto foi necessário criar um novo módulo no RVSec que fosse capaz de executar tais classes. Isso foi feito com a criação da classe ReflectionBased, que a princípio carrega todas as classes presentes no benchmark em um array e então itera sobre cada uma utilizando o recurso Reflection da linguagem Java para executar seu método main e o módulo de erro do RVSec para capturar sempre que uma exceção é lançada durante a execução.

1.1.2 Erros de execução em classes do benchmark

Algumas mudanças foram necessárias para que a execução do benchmark fosse bem sucedida no RVSec. Foram encontrados erros de execução em alguns dos programas presentes no benchmark, tais erros precisaram ser corrigidos manualmente para que todo o benchmark pudesse ser executado pelo RVSec. A tabela 1.1 mostra um resumo dos erros encontrados, relacionando a quantidade encontrada com seu tipo de exceção.

	Exception	count(*)
1	java.io.IOException	9
2	java.security.InvalidAlgorithmParameterException	6
3	java.lang.NumberFormatException	3
4	java.net.UnknowHostException	1
5	java.net.MalformedURLException	1

Tabela 1.1: Resumo dos erros presentes no benchmark

1.1.3 Remoção de alguns dos pacotes

Visto que alguns dos pacotes presentes no benckmark não faziam parte da JCA ou não continham erros que seriam tratados por nossas especificações, decidimos remover tais pacotes e criamos assim uma versão alternativa do benchmark que chamamos MicroCryptoAPI-Bench. Os pacotes removidos foram:

1. Http
2. dummycertvalidation

3. dummyhostnameverifier
4. impropersslsocketfactory

1.2 Resultados

Após feitas as modificações no benchmark em decorrência dos desafios citados anteriormente, fomos capazes de executar sua nova versão no RVSec e fizemos o mesmo nas ferramentas de análise estática CogniCrypt e CryptoGuard. Para fins de comparação, calculamos a *precision* e o *recall* de cada uma das ferramentas e observamos que os resultados obtidos pelo RVSec foram competitivos nesses quesitos, o que prova que não só é possível detectar usos incorretos de APIs criptográficas utilizando análise dinâmica, como os resultados obtidos dessa forma podem ser superiores aos obtidos em análises estáticas.

Capítulo 2

Fundamentação Teórica

Falhas em software são um desvio entre o comportamento observado de um sistema e o comportamento esperado do mesmo. Quando relacionadas à segurança, são aquelas que podem ocasionar vazamento de dados do usuário e por isso são inaceitáveis. Parte destas falhas de segurança são causadas pelo uso incorreto de APIs criptográficas, que podem às vezes ser muito complexas ou conter uma documentação pobre com poucos exemplos de uso seguro. O estudo conduzido em [7] convidou 256 desenvolvedores python para resolver tarefas criptográficas comuns envolvendo criptografia simétrica e assimétrica, utilizando uma de cinco diferentes APIs para tal. Os resultados foram que para cerca de 20% das tarefas resolvidas, entre todas as APIs, os participantes acreditavam que seu código era seguro quando na verdade não era. Isso mostra como é difícil identificar um código escrito de forma insegura apenas lendo o mesmo, felizmente existem ferramentas que são capazes de realizar esta tarefa de forma automática, auxiliando assim o programador a identificar e corrigir potenciais falhas de segurança em seu código, listaremos algumas a seguir.

2.1 Ferramentas de análise estática

Ferramentas de análise estática são aquelas que examinam o código fonte de um programa estaticamente, sem tentar executá-lo. Teoricamente, elas podem examinar o código-fonte de um programa ou uma forma compilada do programa para igual benefício, embora o problema de decodificar o último possa ser difícil [8]. Este tipo de ferramenta não se importa se o programa em análise vai funcionar como deve ou não, apenas analisa seu código buscando por sequências incorretas de ações. Se tratando de criptografia, já existem diversas ferramentas de análise estática disponíveis, apresentaremos a seguir as duas que foram escolhidas para fins de comparação nesta pesquisa.

2.1.1 CrySL

É uma linguagem de definição que permite que especialistas em criptografia especifiquem o uso seguro de suas APIs Crypto que funciona em conjunto com CogniCrypt, um compilador que analisa e verifica as regras CrySL e as traduz em uma análise estática. A análise verifica automaticamente um determinado aplicativo Java ou Android quanto à conformidade com as regras CrySL codificadas. CrySL vai além dos métodos que são úteis para validação geral do uso da API permitindo a expressão de restrições específicas de domínio relacionadas a algoritmos de criptografia e seus parâmetros [9]. Esta ferramenta foi utilizada para analisar mais de 10000 aplicativos android com um conjunto de regras que compreendem as classes e interfaces da JCA e encontrou ao menos um uso incorreto da API em 96% dos aplicativos analisados.

2.1.2 CryptoGuard

CryptoGuard é um conjunto de algoritmos de detecção que refinam pedaços de programa identificando elementos irrelevantes específicos da linguagem nos mesmos. Os refinamentos reduzem os alertas falsos de vulnerabilidades criptográficas entre 76% a 80% [5]. Utilizado em 46 projetos Apache de grande escala e alto impacto e em 6.181 aplicativos Android, o CryptoGuard gerou muitos insights de segurança que ajudaram projetos Apache populares a melhorarem seus códigos, incluindo Spark, Ranger, and Ofbiz.

2.2 Análise dinâmica

Outro tipo de ferramenta capaz de analisar programas são as que funcionam em tempo de execução. Seguindo a definição feita em [10], análise dinâmica é a disciplina em ciência da computação que lida com o estudo, desenvolvimento e aplicação das técnicas de verificação que permitem checar se a execução de um sistema sobre minunciosa observação satisfaz ou viola as dadas propiedades de corretude. Esse tipo de verificação é feito durante a execução de um programa onde são monitoradas chamadas a funções e métodos ou alterações nos valores de variáveis e atributos. É uma verificação muito útil e evita que programas se comportem de forma inesperada quando executados pelo usuário final.

A qualidade de uma ferramenta que realiza análise dinâmica depende diretamente das especificações propostas para ela, pois é através destas que os vários casos de usos corretos e incorretos serão identificados. Tratando-se de segurança, esses casos de uso estarão relacionados a chamadas de funções de uma API criptográfica, a ordem em que estas funções são chamadas e a passagem de constantes como parâmetros para as mesmas. Alguns exemplos de ferramentas conhecidas são:

1. **MCC**: Uma ferramenta para análise de aplicações de usuário MCAPI (API de Comunicação Multicore) [11].
2. **Java PathExplorer**: Uma ferramenta para monitorar traços de execução de programas Java [12].
3. **HAWK**: Uma lógica com ferramentas de suporte para o monitoramento de programas Java [13].

Surge então a pergunta, é possível identificar vulnerabilidades criptográficas utilizando análise dinâmica? Para responder a isso optou-se por desenvolver uma ferramenta que realizasse análise dinâmica e que fosse capaz de identificar usos incorretos de uma API criptográfica. A seguir descrevemos os passos da implementação de tal ferramenta nomeada RVSec, criada com o objetivo de identificar usos incorretos da API criptográfica da linguagem Java em tempo de execução e explicamos também como fizemos a validação da mesma através de testes e comparações com outras ferramentas.

2.3 Trabalhos Relacionados

Esta seção enumera trabalhos relacionados à pesquisa reportada nessa monografia.

2.3.1 How Good Are the Specs? A Study of the Bug-Finding Effectiveness of Existing Java API Specifications

Os autores realizaram um estudo em profundidade sobre a efetividade em encontrar bugs com o uso de especificações propostas anteriormente. Eles utilizaram JavaMOP para monitorar 182 especificações feitas manualmente e 17 especificações mineiradas de forma automática, sobre mais de 18K testes escritos manualmente e mais de 2,1M de testes gerados automaticamente em 200 projetos de código aberto. O estudo mostra que ferramentas como Javamop possuem uma sobrecarga aceitável no monitoramento em tempo de execução e de fato as especificações existentes são capazes de encontrar bugs, porém a grande maioria dos bugs reportados são alarmes falsos [2].

2.3.2 Event-Based Runtime Verification of Java Programs

Neste artigo os autores propõe uma nova lógica e suas ferramentas, denominadas HAWKS em contraste a técnincas de análise estática, sendo elas “Model Checking” [14], “Theorem Proving” [15], e “Static Analysis” [16]. Com HAWKS, ao utilizar um subconjunto de expressões da linguagem Java como proposições, o usuário pode descrever propriedades

temporais relacionando diferentes pontos no programa e seus objetos acessíveis e verificar o programa em relação a essas propriedades durante o tempo de execução [13].

2.3.3 CrySL: Validating Correct Usage of Cryptographic APIs

Os autores apresentam CrySL, uma linguagem de definição que permite a especialistas em criptografia especificar o uso seguro das bibliotecas de criptografia que eles fornecem e CogniCrypt, um compilador que analisa e verifica as regras CrySL e as traduz em uma análise estática. Para validar CrySL, os autores codificaram um conjunto de regras para as classes e interfaces JCA (*Java Cryptography Architecture*). Utilizando a análise estática gerada ele avaliaram mais de 10K aplicações android e constataram que 96% delas apresentam pelo menos um uso incorreto da JCA [9].

2.3.4 Evaluation of Static Vulnerability Detection Tools with Java Cryptographic API Benchmarks

Neste artigo são propostos dois benchmarks(CryptoAPI-Bench e ApacheCryptoAPI-Bench) para a comparação de diversas ferramentas de detecção de vulnerabilidades criptográficas. Os autores caracterizam os tipos de erros existentes nos benchmarks e então comparam os resultados obtidos por cada uma das ferramentas ao executa-los, levando em conta a precisão, escalabilidade e garantias de segurança [1].

Capítulo 3

Integrando o CryptoAPI-Bench no RVSec

RVSec é um projeto criado como parte de uma pesquisa que busca responder se é possível detectar o uso incorreto de APIs criptográficas através de análise dinâmica. É um projeto Maven implementado utilizando a linguagem de programação Java com o objetivo de detectar usos incorretos da JCA (*Java Cryptography Architecture*), que é a principal API criptográfica da linguagem Java, em um determinado conjunto de programas enquanto os mesmos são executados.

3.1 Implementação do RVSec

O ponto de partida para a implementação do RVSec foi a escolha da framework JavaMOP, uma framework que utiliza Programação Orientada ao Monitoramento (MOP), ou seja, é uma estrutura de desenvolvimento e análise de software que visa reduzir a lacuna entre a especificação formal e a implementação, permitindo que estas formem um sistema [17]. Buscou-se então por especificações formais de uso seguro da JCA, para que as mesmas pudessem ser traduzidas utilizando o JavaMOP. Optou-se por traduzir as especificações CrySL presentes em [18], visto que as mesmas já são utilizadas no CogniCrypt, um plugin desenvolvido para auxiliar desenvolvedores Java no uso de APIs criptográficas.

A primeira fase de desenvolvimento consistiu então da tradução das especificações CrySL para JavaMOP, onde cada especificação traduzida corresponde a uma regra que quando infringida gera uma exceção. A cada especificação que era traduzida, eram também escritos testes de código em Java para saber se as mesmas estavam sendo capazes de detectar os específicos casos de uso incorreto da JCA. Foram feitas análises para identificar se todas as especificações de CrySL deveriam ser traduzidas para este novo projeto, identificou-se que mais de 90% dos usos incorretos da JCA eram de um grupo menor

de especificações e portanto apenas estas foram traduzidas pelos membros da pesquisa. Ainda assim, foram traduzidas 22 das 51 especificações CrySL pelos membros da pesquisa, algumas das principais foram: Cipher, KeyStore, MessageDigest, SecretKey.

Para cada uma das especificações foi criado um arquivo de teste com os vários casos de uso dos módulos da JCA e assim encerrou-se a primeira fase da implementação. A segunda fase consistiu em criar uma Classe Java capaz de armazenar os erros detectados durante a execução dos testes no RVSec, de forma a gerar uma saída com tais erros, afim de exibi-los ao fim da análise. A ferramenta RVSec recebe como entrada um ou mais arquivos no formato .java e executa cada um deles monitorando as chamadas a JCA, se uma exceção é lançada, a mesma será coletada pelo módulo de Erro da ferramenta. É importante salientar que independente do número de erros existentes em cada arquivo de entrada, o resultado é o mesmo, pois classificamos o arquivo inteiro como com erro ao invés de capturar seus erros separadamente.

Visto que as especificações se mostraram eficientes para os casos de teste escritos manualmente presentes no projeto, iniciou-se uma segunda fase da pesquisa que se tratou da escolha de um bechmark para validação da ferramenta RVSec. Foi selecionado o benchmark CryptoAPI-Bench [6], afim de se realizar um estudo empírico sobre como nossas especificações seriam efetivas para os casos presentes neste benchmark, assim poderíamos avaliar nossa ferramenta em termos de precision e recall com outras ferramentas já existentes na web.

3.2 CryptoAPI-Bench

Criado para a pesquisa [1], que buscava avaliar ferramentas de detecação de vulnerabilidades estáticas, este benchmark foi gerado manualmente e possui 181 casos de teste unitários, divididos em 18 categorias de usos incorretos da JCA que se enquadram em casos avançados e casos básicos. Dentre os casos de teste, embora em sua maioria sejam de usos incorretos, há também alguns casos de usos corretos da JCA, afim de testar se as ferramentas capturam casos falsos positivos. Entre as categorias estão: Cryptographic Key, Password in PBE, Password in KeyStore, Symmetric Ciphers, Cryptographic Hash, etc.

A definição dos tipos de vulnerabilidades presentes em cada uma destas categorias e também o número de casos de teste presentes em cada uma pode ser encontrado no artigo [1]. Também lá, este benchmark foi executado por 4 ferramentas de análise estática, onde se calculou a *precision* e o *recall* obtidos por cada uma, os resultados obtidos foram importantes para nossa pesquisa uma vez que decidimos avaliar nossa ferramenta

utilizando estas mesmas métricas. As ferramentas avaliadas por eles foram: SpotBugs, CryptoGuard, CrySL e Coverity.

3.3 Desafios

3.3.1 Execução automática do benchmark através da RVSec

Os testes escritos manualmente durante a primeira parte da pesquisa, eram testes unitários feitos utilizando a JUnit, um framework que facilita o desenvolvimento e execução de testes unitários em código Java [19]. O exemplo na figura 3.1 ilustra um destes testes, este foi feito para a especificação de uso da classe Cipher, presente na JCA, se tratando de um caso de uso correto desta. Há um arquivo de teste contendo diversos casos de teste unitário para cada uma das especificações traduzidas. Estes testes eram integrados no processo de build do RVSec via ferramenta Maven e isso não era possível de se fazer para os programas presentes no CryptoAPI-Bench a menos que se alterasse os códigos de cada programa.

```
@Test
public void cipherValidTest4()
        throws NoSuchPaddingException, IllegalBlockSizeException, NoSuchAlgorithmException, InvalidKeyException {

    KeyGenerator keyGenerator0 = KeyGenerator.getInstance("AES");
    SecretKey secretKey = keyGenerator0.generateKey();
    Assertions.hasEnsuredPredicate(secretKey);
    Assertions.mustBeInAcceptingState(keyGenerator0);

    Key wrappedKey = KeyGenerator.getInstance("AES").generateKey();

    Cipher cipher0 = Cipher.getInstance("AES/CBC/PKCS5Padding");
    cipher0.init(Cipher.WRAP_MODE, secretKey);
    byte[] wrappedKeyBytes = cipher0.wrap(wrappedKey);
    Assertions.hasEnsuredPredicate(wrappedKeyBytes);
    Assertions.mustBeInAcceptingState(cipher0);

}
```

Figura 3.1: Exemplo 3: Teste unitário feito durante a primeira parte da pesquisa

Portanto foi necessário criar um novo módulo para executar o CryptoAPI-Bench de forma automática no RVSec. Primeiro foi criado um protóripo independente do RVSec, para testar o recurso Reflection da linguagem Java, afim de descobrir se o utilizando, seria possível chamar o método main de cada um dos programas presentes no benchmark. Este protótipo abria o arquivo JAR gerado pelo benchmark que continha todas as classes deste, as armazenava em um array e então utilizava Reflection para executar cada uma,

iterando pelo array. Este protótipo serviu de base para a criação de uma nova classe no RVSec, que seria capaz de integrar o benchmark ao mesmo.

Foi então criada a classe, chamada ReflectionBased, que é capaz de executar todos os casos de testes do CryptoAPI-Bench, estes foram movidos para dentro do repositório do RVSec, em um diretório chamado de Bench02. ReflectionBased é uma classe Java que utiliza o recurso Reflection da linguagem para poder executar o método main de outras classes. Primeiro, a classe ReflectionBased carrega todas as classes do diretório Bench02 em um *array* de classes, depois itera por cada uma destas classes do benchmark e então utiliza o recurso Reflection do Java para invocar o método main de cada uma delas. O módulo de erro da ferramenta RVSec, que já era utilizado nos testes unitários desenvolvidos manualmente, captura as exceções lançadas durante essas execuções. Assim, tornou-se possível a automatização da execução do CryptoAPI-Bench pelo RVSec.

3.3.2 Erros de execução

Algumas mudanças foram necessárias para que a execução do benchmark fosse bem sucedida em nosso projeto. Foram encontrados erros de compilação e/ou execução em algumas das classes presentes no benchmark, tais erros precisaram ser corrigidos para que todo o benchmark pudesse ser executado, de forma automática, em nosso projeto. A Tabela 3.1 contém as informações detalhadas sobre cada um dos erros encontrados, suas colunas representam respectivamente o nome da classe, a linha do erro, a exceção lançada e a correção necessária.

3.3.3 Remoção de alguns dos pacotes

Visto que alguns dos pacotes presentes em CryptoAPI-Bench não estavam relacionados ao uso da JCA ou não continham erros que seriam tratados pelas especificações CrySL, optamos por remover tais pacotes e criamos assim uma versão alternativa do benchmark que chamamos MicroCryptoAPI-Bench. Os pacotes removidos e a quantidade de classes em cada um foram:

1. Http: 10 classes
2. dummycertvalidation: 3 classes
3. dummyhostnameverifier: 2 classes
4. impropersslsocketfactory: 1 classe

A seguir mostramos um exemplo de cada um destes pacotes sendo eles as figuras 3.2 classe do pacote Http, 3.3 classe do pacote DummyCertValidation, 3.4 classe do pacote

Classe	Linha	Exceção	Correção
PredictableKeyStorePasswordABICase1.java	21	IOException: Invalid keystore format null	String key = "password"; cacerts = new File("./target/test-classes/testInput-ks") .toURI().toURL();
PredictableKeyStorePasswordABICase2.java	32	IOException: Invalid keystore format null	public static final String DEFAULT_ENCRYPT_KEY = "password"; cacerts = new File("./target/test-classes/testInput-ks") .toURI().toURL();
PredictableKeyStorePasswordABICase3.java	26	IOException: nvalid keystore format	cacerts = new URL("./target/test-classes/testInput-ks");
PredictableKeyStorePasswordABMCCase1.java	11	IOException: Invalid keystore format null	String key = "password";
PredictableKeyStorePasswordABPSCase1.java	22	IOException: Invalid keystore format null	cacerts = new File("./target/test-classes/testInput-ks") .toURI().toURL(); String defaultKey = "password";
PredictableKeyStorePasswordBBCase1.java	21	IOException: Invalid keystore format null	cacerts = new File("./target/test-classes/testInput-ks") .toURI().toURL(); String defaultKey = "password";
PredictableKeyStorePasswordCorrected.java	21	IOException: Invalid keystore format null	cacerts = new File("./target/test-classes/testInput-ks") .toURI().toURL();
PredictableKeyStorePasswordBBCase1	21	IOException: Invalid keystore format	cacerts = new File("./target/test-classes/testInput-ks") .toURI().toURL(); String defaultKey = "password";
PredictableSeedsABICase4.java	24	NumberFormatException: For input string: "[C@dbe9223"null	String defaultKey = "password";
StaticInitializationVectorABHCase1.java	19	InvalidAlgorithmParameterException: Wrong IV length: must be 16 bytes long	byte [] bytes = "abcde————".getBytes("UTF-8");
StaticInitializationVectorABHCase2.java	22	InvalidAlgorithmParameterException: Wrong IV length: must be 16 bytes long	String name = "abcdef————"; hm.put("aaa", "abcde————");
StaticInitializationVectorABICase1.java	22	InvalidAlgorithmParameterException: Wrong IV length: must be 16 bytes long	byte [] bytes = "abcde————".getBytes();
StaticInitializationVectorABICase2.java	17	InvalidAlgorithmParameterException: Wrong IV length: must be 16 bytes long	public static final String DEFAULT_INITIALIZATION = "abcde————"; IvParameterSpec ivSpec = new IvParameterSpec(new byte[]{Byte .parseByte(new String(initialization))});
StaticInitializationVectorABICase3.java	25	InvalidAlgorithmParameterException: Wrong IV length: must be 16 bytes long	byte [] bytes = "abcde————".getBytes();
StaticInitializationVectorABMCCase1.java	11	InvalidAlgorithmParameterException: Wrong IV length: must be 16 bytes long	byte [] bytes = "abcde————".getBytes();
StaticInitializationVectorBBCase1.java	17	InvalidAlgorithmParameterException: Wrong IV length: must be 16 bytes long	byte [] bytes = "abcde————".getBytes();
StaticInitializationVectorABICase2	17	NumberFormatException: For input string: 'abcde'	IvParameterSpec ivSpec = new IvParameterSpec(new String(initialization).getBytes());
StaticSaltsABICase2.java	27	NumberFormatException: For input string: "[C@608b09d9"	pbeParamSpec = new PBEParameterSpec(new byte[]{Byte .parseByte(new String(salt))}, count);
ImproperSocketManualHostBBCase1	10	UnknownHostException: my.host.name	SSLSocket socket = (SSLSocket) ssf .createSocket("www.google.com", 443);

Tabela 3.1: Correções aplicadas a classes com erro em CryptoAPI-Bench

DummyHostNameVerifier e 3.5 classe do pacote ImproperSSLSocketFactory. Pode-se observar a ausência do uso de qualquer classe da JCA nos códigos destes exemplos, isso já as tornava desinteressantes para análise pelo RVSec visto que saía do escopo de suas especificações, além disso pode-se notar também a ausência de método *main* nas classes das figuras 3.3 e 3.4, o que impossibilitaria a execução destas e consequentemente a análise pelo RVSec, já que este captura erros apenas em tempo de execução e uma classe não pode ser executada sem um método *main*.

```
package org.cryptoapi.bench.http;

import java.net.MalformedURLException;
import java.net.URL;

public class HttpProtocolABICase1 {

    public static void main(String [] args) throws MalformedURLException {
        String url = "http://www.google.com";
        go(url);

    }

    private static void go(String url) throws MalformedURLException {
        System.out.println(new URL(url));
    }

}
```

Figura 3.2: Exemplo 4: HttpProtocolABICase1.java

3.4 Resultados

Após feitas as modificações no CryptoAPI-Bench para que o mesmo pudesse ser executado pela ferramenta RVSec, obtivemos um novo subconjunto de casos de testes que chamamos MicroCryptoAPI-Bench, este possui 152 casos de testes que são executados via Reflection por nossa ferramenta. Após a execução de todas as classes presentes no MicroCryptoAPI-Bench, as vulnerabilidades detectadas foram salvas em um arquivo .csv automaticamente pela ferramenta RVSec, para que pudéssemos observar os resultados obtidos. Utilizando o Ground truth original do CryptoAPI-Bench, constatamos que nossa ferramenta foi capaz de identificar 105 das 137 vulnerabilidades presentes no MicroCryptoAPI-Bench corretamente e apenas 4 de forma equivocada.

```java
package org.cryptoapi.bench.dummycertvalidation;

import javax.net.ssl.X509TrustManager;
import java.security.cert.CertificateException;
import java.security.cert.X509Certificate;

public class DummyCertValidationCase3 implements X509TrustManager {
    @Override
    public void checkClientTrusted(X509Certificate[] x509Certificates, String s) throws CertificateException {

    }

    @Override
    public void checkServerTrusted(X509Certificate[] x509Certificates, String s) throws CertificateException {

    }

    @Override
    public X509Certificate[] getAcceptedIssuers() {
        return null;
    }
}
```

Figura 3.3: Exemplo 5: DummyCertValidationCase3.java

```java
package org.cryptoapi.bench.dummyhostnameverifier;

import javax.net.ssl.HostnameVerifier;
import javax.net.ssl.SSLSession;

public class DummyHostNameVerifierCase1 implements HostnameVerifier {
    public boolean verify(String s, SSLSession sslSession) {
        return true;
    }
}
```

Figura 3.4: Exemplo 6: DummyHostNameVerifierCase1.java

Afim de avaliar o desempenho obtido pela ferramenta RVSec descrito acima, executamos o MicroCryptoAPI-Bench em outras duas ferramentas, estas de análise estática, sendo elas a CogniCrypt e a CryptoGuard. Observamos que a ferramenta CogniCrypt foi capaz de identificar 108 vulnerabilidades corretamente e 27 de forma equivocada enquanto que a CryptoGuard identificou 87 corretamente e 18 de forma equivocada. Comparamos então os resultados obtidos pelas três ferramentas em termos de precision, calculado em função dos verdadeiros positivos e falsos positivos e recall, calculado em função dos verdadeiros positivos e falsos negativos. Os valores obtidos por cada uma das ferramentas podem ser observados na Tabela 3.2.

Os resultados mostram que apesar de não realizar análise estática como as outras duas ferramentas, a RVSec obteve um melhor valor de precision e um valor de recall próximo

```java
package org.cryptoapi.bench.improperssslsocketfactory;

import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSocketFactory;
import java.io.IOException;

public class ImproperSocketManualHostBBCase1 {
    public static void main(String [] args) throws IOException {
        SSLSocketFactory ssf = (SSLSocketFactory) SSLSocketFactory.getDefault();
        SSLSocket socket = (SSLSocket) ssf.createSocket("my.host.name", 443);
    }
}
```

Figura 3.5: Exemplo 7: ImproperSocketManualHostBBCase1.java

Ferramenta	**Precision**	**Recall**
RVSec	0,96	0,76
CogniCrypt	0,8	0,78
CryptoGuard	0,82	0,63

Tabela 3.2: Comparação entre as ferramentas

ao das demais. É interessante observar que a ferramenta CogniCrypt utiliza o conjunto de regras CrySL, o mesmo que deu origem as nossas especificações, ainda assim o resultado obtido pela RVSec em relação a precision foi, com certa folga, superior enquanto que em relação ao recall os resultados foram muito próximos. Isso prova que mesmo utilizando análise dinâmica é possível identificar o uso incorreto de uma API criptográfica, de forma a obter resultados parecidos ou até melhores que os de ferramentas de análise estática.

Capítulo 4

Conclusão

A busca por vulnerabilidades criptográficas em programas é parte essencial do processo de desenvolvimento de software e está geralmente relacionada ao uso incorreto de APIs criptográficas. Embora não exista um método ideal capaz de sempre capturar todas as vulnerabilidades presentes em um programa, essa busca costuma ser feita por ferramentas de análise estática de código fonte, por isso, surgiu o questionamento, é possível buscar por vulnerabilidades criptográficas utilizando análise dinâmica?

Para responder esta pergunta, foi desenvolvida uma ferramenta que seguia especificações formais de usos incorretos da JCA e executava programas Java buscando detectar erros ao longo da execução baseados nas especificações. A ferramenta, nomeada RVSec, se mostrou capaz de identificar erros das 22 categorias de vulnerabilidades descritas em suas especificações. Para fins de validação do desempenho da ferramenta, selecionamos um benchmark no qual precisamos fazer algumas alterações para que o mesmo pudesse ser executado pela RVSec, umas vez que essas modificações foram feitas nós observamos os resultados obtidos por nossa ferramenta e também executamos essa nova versão do benchmark em ferramentas de análise estática, para poder comparar o desempenho de cada uma.

A comparação feita entre as ferramentas em termos de precision e recall mostrou que a RVSec obteve, em média, resultados próximos ou superiores ao das outras ferramentas. Isso nos permitiu concluir que não só é possível detectar vulnerabilidades criptográficas em tempo de execução com análise dinâmica, como também que essa verificação pode ser ainda mais eficaz que a análise estática. Observou-se porém que pode ser difícil encontrar benchmarks que estejam prontos para serem executados em análise dinâmica, afim de validar o desempenho de ferramentas deste tipo.

Em trabalhos futuros, deve-se buscar por mais benchmarks que possam ser executados pela ferramenta RVSec, o que pode ser um desafio, vistas as alterações que foram necessárias no CryptoAPI-Bench para que o mesmo pudesse ser executado pela RVSec,

porém com isso seria possível reavaliar o desempenho da ferramenta. Seria ideal também a criação de um benchmark voltado exclusivamente para análise dinâmica de código, visto que devem aparecer mais ferramentas deste tipo com o passar do tempo. Por fim, seria interessante realizar a tradução para MOP das especificações CrySL restantes e avaliar se estas teriam algum impacto no desempenho da RVSec sobre o benchmark selecionado neste documento e também sobre os futuros benchmarks que possam vir a ser escolhidos para avaliação da ferramenta.

Referências

[1] S. Afrose, Y. Xiao, S. Rahaman B. P. Miller D. Yao: *Evaluation of static vulnerability detection tools with java cryptographic api benchmarks.* Arxiv, 2021. 1, 8, 10

[2] O. Legunsen, W. U. Hassan, X. Xu G. Rosu e D. Marinov: *How good are the specs? a study of the bug-finding effectiveness of existing java api specifications.* Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, páginas 602—-613, 2016. 1, 7

[3] D. Jin, P. O. Meredith, C. Lee e G. Roşu: *Javamop: Efficient parametric runtime monitoring framework.* 34th International Conference on Software Engineering (ICSE), páginas 1427–1430, 2012. 1

[4] Foundation, Eclipse: *Cognicrypt.* `https://www.eclipse.org/cognicrypt/`, acesso em 2022-05-13. 1

[5] Sazzadur Rahaman, 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.* Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), 2019. 1, 6

[6] CryptoAPI-Bench: *Cryptoapi-bench.* `https://github.com/CryptoAPI-Bench/CryptoAPI-Bench`, acesso em 2022-04-26. 1, 10

[7] Acar, Yasemin, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L. Mazurek e Christian Stransky: *Comparing the usability of cryptographic apis.* Em *2017 IEEE Symposium on Security and Privacy (SP)*, páginas 154–171, 2017. 5

[8] Chess, B. e G. McGraw: *Static analysis for security.* IEEE Security Privacy, 2(06):76–79, 2004. 5

[9] S. Kruger, J. Spath, K. Ali E. Bodden M. Mezini: *Crysl: Validating correct usage of cryptographic apis.* Arxiv, 2017. 6, 8

[10] Leucker, Martin e Christian Schallhart: *A brief account of runtime verification.* The Journal of Logic and Algebraic Programming, 78(5):293–303, 2009, ISSN 1567-8326. `https://www.sciencedirect.com/science/article/pii/S1567832608000775`, The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07). 6

[11] Sharma, Subodh, Ganesh Gopalakrishnan, Eric Mercer e Jim Holt: *Mcc: A runtime verification tool for mcapi user applications*. Em *2009 Formal Methods in Computer-Aided Design*, páginas 41–44, 2009. 7

[12] Havelund, Klaus e Grigore Rosu: *Java pathexplorer - a runtime verification tool*. junho 2001. 7

[13] d'Amorim, Marcelo e Klaus Havelund: *Event-based runtime verification of java programs*. SIGSOFT Softw. Eng. Notes, 30(4):1–7, may 2005, ISSN 0163-5948. `https://doi.org/10.1145/1082983.1083249`. 7, 8

[14] E. M. Clarke, O. Grumberg e D. A. Peled: *Model Checking*, volume 1. The MIT Press, Cambridge, Massachusetts, 1999. 7

[15] Manna, Z. e A. Pnueli: *Temporal Verification of Reactive Systems: Safety*, volume 1. Springer, New York, 1995. 7

[16] F. Nielson, H. R. Nielson e C. Hankin: *Principles of Program Analysis*, volume 1. Springer-Verlag, 1999. 7

[17] Inc., Runtime Verification: *javamop*. `https://github.com/runtimeverification/javamop`, acesso em 2022-04-26. 9

[18] CROSSINGTUD: *Crypto-api-rules*. `https://github.com/CROSSINGTUD/Crypto-API-Rules`, acesso em 2022-03-03. 9

[19] Team, The JUnit: *Junit5*. `https://junit.org/junit5/`, acesso em 2022-05-13. 11