

**UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROJETO DE GRADUAÇÃO**

KELVIN FERREIRA MILACH

**DETECÇÃO DE ATAQUES DE NEGAÇÃO DE SERVIÇO
POR INJEÇÃO DE SQL UTILIZANDO A TELEMETRIA DA
COMPUTAÇÃO EM NUVEM**

VITÓRIA
2021

KELVIN FERREIRA MILACH

**DETECÇÃO DE ATAQUES DE NEGAÇÃO DE SERVIÇO POR
INJEÇÃO DE SQL UTILIZANDO A TELEMETRIA DA
COMPUTAÇÃO EM NUVEM**

Parte manuscrita do Projeto de Graduação do aluno **Kelvin Ferreira Milach**, apresentado ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheiro Eletricista.

Orientador: Prof. Dr. Moisés R. Nunes Ribeiro

Coorientador: MSc. João Henrique G. M. Corrêa

VITÓRIA
2021

KELVIN FERREIRA MILACH

**DETECÇÃO DE ATAQUES DE NEGAÇÃO DE SERVIÇO POR
INJEÇÃO DE SQL UTILIZANDO A TELEMETRIA DA
COMPUTAÇÃO EM NUVEM**

Parte manuscrita do Projeto de Graduação do aluno **Kelvin Ferreira Milach**, apresentado ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheiro Eletricista.

Aprovada em 20 de abril de 2021.

COMISSÃO EXAMINADORA:

Prof. Dr. Moisés R. Nunes Ribeiro
Universidade Federal do Espírito Santo
Orientador

MSc. João Henrique G. M. Corrêa
Universidade Federal do Espírito Santo
Coorientador

Prof. Dr. Magnos Martinello
Universidade Federal do Espírito Santo
Examinador

MSc. Victor M. G. Martínez
Universidade Federal do Espírito Santo
Examinador

Aos meus amigos e família, por todo o amor e suporte dado durante essa árdua jornada,
sempre me dando coragem durante os momentos mais difíceis desta graduação.

RESUMO

A computação em nuvem tem se tornando cada vez mais parte da realidade de organizações mundo afora, sejam estas grandes empresas ou instituições públicas. Observando-se o crescente número de casos de serviços tendo sua disponibilidade comprometida, há de se pensar em novos meios para se detectar prematuramente estes ataques. A adoção dessas tecnologias pode trazer novos métodos para lidar com ataques de negação de serviço (DoS, do inglês *denial-of-service*) ou distribuídos (DDoS, do inglês *distributed denial-of-service*), visto que estes são as maiores ameaça às organizações. Este projeto busca testar a eficácia da utilização da telemetria de ambientes em nuvem, com o projeto *OpenStack*, para a detecção de ataques DoS na camada de aplicação, visto que a distinção entre requisições válidas e maliciosas é uma tarefa complexa. Dessa forma, foram utilizados os seguintes algoritmos de aprendizagem de máquina supervisionados: árvore de decisão, floresta aleatória, *k-nearest neighbors*, Naïve Bayes gaussiano e perceptron multicamadas, obtendo-se uma detecção eficaz de comportamentos anômalos na aplicação afetada, com resultados comparáveis a outros estudos com fontes mais tradicionais de dados.

Palavras-chave: Aprendizagem de máquina. Computação em nuvem. Negação de serviço. Plataforma *OpenStack*. Telemetria.

ABSTRACT

Cloud computing has become increasingly part of the reality of organizations around the world, whether these are large companies or public institutions. In view of the growing number of service cases with their availability compromised, new means must be considered to detect these attacks prematurely. The adoption of these technologies may bring new methods to deal with denial-of-service (DoS) or distributed (DDoS, distributed denial-of-service) attacks, since these are the biggest threat to organizations. This project seeks to test the effectiveness of using telemetry in cloud environments, with the OpenStack project, for the detection of DoS attacks at the application layer, since the distinction between valid and malicious requests is a complex task. Thus, the following supervised machine learning algorithms were used: Decision Tree, Random forest, K-Nearest Neighbors, gaussian Naïve Bayes and Multilayer Perceptron, obtaining an effective detection of anomalous behaviors in the affected application, with results comparable to other studies with more traditional data sources.

Keywords: Machine learning. Cloud computing. Denial-of-service. OpenStack platform. Telemetry.

LISTA DE FIGURAS

Figura 1 – Ataques DoS tradicionais (a) e na camada de aplicação (b)	12
Figura 2 – Modalidades de serviço em nuvem	17
Figura 3 – Arquitetura do serviço Nova	19
Figura 4 – Ataques (a) DoS e (b) DDoS.....	21
Figura 5 – Injeção de SQL para listar todos os usuários da aplicação	25
Figura 6 – Topologia lógica dentro da plataforma <i>OpenStack</i>	28
Figura 7 – Processo de validação cruzada (com $K = 3$).....	31
Figura 8 – Matriz confusão do modelo Árvore de Decisão.....	36
Figura 9 – Matriz confusão do modelo Naïve Bayes gaussiano.....	37
Figura 10 – Matriz confusão do modelo <i>K-Nearest Neighbors</i>	37
Figura 11 – Matriz confusão do modelo <i>Multi Layer Perceptron</i>	38
Figura 12 – Matriz confusão do modelo <i>Random Forest</i>	38
Figura 13 – Curvas ROC por estimador	39

LISTA DE TABELAS

Tabela 1 – Métricas para avaliação dos modelos treinados.....	35
---	----

LISTA DE QUADROS

Quadro 1 – Serviços <i>OpenStack</i>	18
Quadro 2 – Tráfegos ao servidor Web.....	27
Quadro 3 – Métricas coletadas	29
Quadro 4 – Hiperparâmetros selecionados	32

LISTA DE ABREVIATURAS E SIGLAS

API	<i>Application Programming Interface</i>
AUC	<i>Area Under Curve</i>
AWS	<i>Amazon Web Services</i>
CIA	<i>Confidentiality, Integrity and Availability</i>
DoS	<i>Denial-of-Service</i>
DDoS	<i>Distributed Denial-of-Service</i>
DVWA	<i>Damn Vulnerable Web Application</i>
FN	Falso Negativo
FP	Falso Positivo
HTTP	<i>Hypertext Transfer Protocol</i>
IaaS	<i>Infrastructure as a Service</i>
ISO	<i>International Organization for Standardization</i>
NaaS	<i>Network as a Service</i>
NERDS	Núcleo de Estudos em Redes Definidas por Software
NIST	<i>National Institute of Standards and Technology</i>
OSI	<i>Open System Intercommunication</i>
PaaS	<i>Platform as a Service</i>
ROC	<i>Receiver Operating Characteristic</i>
SaaS	<i>Software as a Service</i>
SQL	<i>Structured Query Language</i>
SQLi	<i>SQL injection</i>
SSH	<i>Secure Shell</i>
VM	<i>Virtual Machine</i>
VN	Verdadeiro Negativo
VP	Verdadeiro Positivo

SUMÁRIO

1	INTRODUÇÃO	11
1.1	Justificativas	13
1.2	Objetivos	14
1.2.1	Objetivo Geral.....	14
1.2.2	Objetivos Específicos.....	14
2	EMBASAMENTO TEÓRICO	15
2.1	Computação em Nuvem	15
2.2	<i>OpenStack</i>	17
2.3	Segurança da Informação	19
2.4	Ataques de Negação de Serviço	20
2.5	Injeção de SQL	23
2.6	Aprendizagem de Máquina	25
3	METODOLOGIA	27
3.1	Introdução	27
3.2	Geração do <i>Dataset</i>	27
3.3	Pré-Processamento	29
3.4	Treinamento de Modelos e Classificação dos Dados	30
3.5	Recursos Computacionais	32
4	RESULTADOS	34
4.1	Métricas de Avaliação	34
4.2	Experimentos	34
5	CONCLUSÕES	40
	REFERÊNCIAS BIBLIOGRÁFICAS	42
	APÊNDICE A – CÓDIGO PARA TREINAMENTO E MÉTRICAS	45
	APÊNDICE B – CÓDIGO PARA CONEXÃO COM O GNOCCHI	47
	APÊNDICE C – CÓDIGO PARA GERAÇÃO DE DADOS	56

1 INTRODUÇÃO

Em um mundo cada vez mais conectado, a evolução e a adoção crescente da computação em nuvem têm afetado significativamente como a indústria e outras instituições se organizam, tendo em mente as vantagens oferecidas, como provisionamento automático de recursos, alta elasticidade e disponibilidade. Todas são características desejáveis que levam a esta rápida expansão de serviços e aplicações em nuvem (VERAS, 2015). Entretanto, este paradigma de computação em nuvem ainda apresenta velhos problemas, sendo requisitos de segurança uma das principais e constantes preocupações para esta tecnologia.

Entre as possíveis ameaças que existem para serviços hospedados, os ataques de negação de serviço (DoS, do inglês *denial-of-service*) estão se tornando cada vez mais comuns, com um crescente aumento no volume de ataques nos últimos anos, sendo previsto que o número desses subirá para 15,4 milhões até 2023 em uma escala global (ARBOR NETWORKS, 2018). Em 2016, websites brasileiros ligados a organizações das Olimpíadas do Rio de Janeiro sofreram ataques de DDoS (do inglês *distributed denial-of-service*) com um volume de 500 Gbps (ROHR, 2016), um dos maiores registrados até então. Em 2018, o ataque contra o Github bateu recorde, com um volume de pico de 1,3 Tbps (NEWMAN, 2018).

Um ataque de negação de serviço é aquele que busca atacar a disponibilidade do sistema, um dos componentes da tríade da segurança da informação, dessa forma degradando ou até mesmo negando o acesso aos recursos do sistema por seus usuários (FENG et al. 2020). Um exemplo seria causar que um servidor web não seja capaz de responder clientes que mandem solicitações ao mesmo.

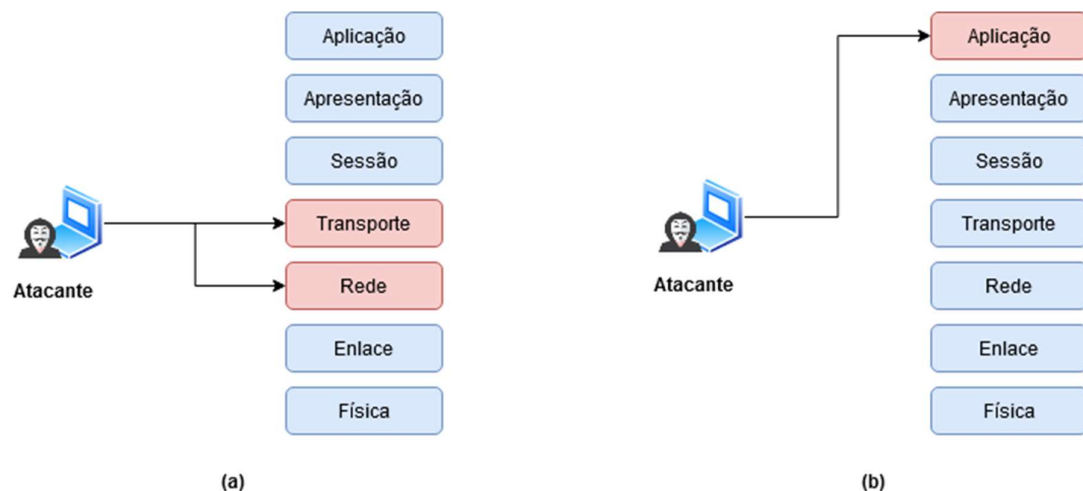
Os ataques de negação de serviço podem ser divididos em duas formas: se este é classificado como distribuído ou não. Ataques de DoS são provenientes de uma única máquina atacante, sendo esses mais fáceis de se detectar e mitigar devido a esta limitação. Já ataques que se originam de múltiplas máquinas ou endereços IP diferentes são ditos distribuídos, sendo esta a forma mais comum de ataque atualmente, visto que a capacidade de gerar tráfego é muito maior, causando uma indisponibilidade mais rápida no alvo.

Ataques DDoS também podem ser separados em outros três tipos (BHOSALE et al., 2017):

- Ataques de inundação: são aqueles que enviam um grande número de pacotes ao alvo, incapacitando-o caso o tráfego do ataque seja superior à capacidade do alvo de processar o mesmo. Um exemplo deste tipo de ataque seria o *UDP flood*;
- Ataques de amplificação: ataques que se utilizam de *IP spoofing*, enviando pequenas requisições a uma série de refletores. Estes, por sua vez, enviarão suas respostas que, possuindo uma grande quantidade de bytes, são enviadas à vítima, gerando um enorme tráfego e incapacitando a mesma. Uma forma conhecida de gerar este tráfego é a amplificação DNS;
- Ataques na camada de aplicação: estes ataques buscam atingir a camada de aplicação do modelo OSI, se utilizando de poucos recursos para causar a negação de serviço. Os métodos HTTP GET e HTTP POST são muito empregados nesta categoria de ataques.

Ataques DoS ou DDoS, como mostrado na Figura 1(a), tradicionalmente utilizam de protocolos nas camadas de rede e/ou transporte do modelo OSI (do inglês *Open System Interconnection*) (TANENBAUM; WETHERAL, 2011) para causar negação de serviço em suas vítimas, através de uma grande quantidade de pacotes gerados. Com isso, uma organização teria todos os serviços de seu servidor atacado indisponíveis para responder requisições de usuários legítimos. Um exemplo de ataque desse tipo seria o *SYN Flood* (HUSSAIN et al., 2019), que faz a utilização intencionalmente incompleta do *three-way handshake* do protocolo TCP, abrindo novas conexões com o alvo via pacotes SYN. Após a resposta SYN+ACK do alvo, o atacante não responde com pacotes ACK, ocupando recursos da vítima e causando a negação de serviço.

Figura 1 – Ataques DoS tradicionais (a) e na camada de aplicação (b)



Fonte: Produção própria do autor.

Entretanto, sabe-se também da existência de ataques de negação de serviço que atuam diretamente em aplicações, como bancos de dados ou servidores web, como mostrado na Figura 1(b). Estes ataques, como o GET *flood*, não são facilmente detectáveis (ABURADA; ARIKAWA; USUZAKI, 2019), devido à sua grande semelhança a requisições feitas por usuários legítimos, o que acarretou seu crescente uso em ataques DDoS recentes (PRASEED; THILAGAM, 2019).

Com isto, tem-se a necessidade de buscar outras técnicas que possam indicar quando um ataque DDoS de aplicação possa estar ocorrendo. Uma possível resposta a este problema seria a utilização de métricas em tempo real advindas de telemetria de soluções da computação em nuvem, buscando adquirir informações que métodos tradicionais não obtêm. Dessa forma, um operador, utilizando dados do uso de recursos, como processamento de CPU, escrita e leitura de disco e número de pacotes recebidos e enviados, entre outros, poderia aplicar técnicas de detecção de anomalias, reconhecendo a presença de ataques na camada de aplicação.

Algoritmos de aprendizagem de máquina (do inglês, *machine learning*) são largamente utilizados quando uma grande quantidade de dados é disponibilizada, assim como múltiplas características que podem ser aproveitadas, sendo possível usar estas técnicas para a detecção de ataques de DoS e DDoS (ROEMPLUK; SURINTA, 2019). Neste caso, os dados de telemetria mencionados anteriormente (processamento de CPU, número de pacotes etc.) são candidatos perfeitos para tentar correlacionar com a presença de ataques na nuvem.

Pelo exposto, neste trabalho foi desenvolvido um modelo de detecção de ataques DoS, utilizando-se da telemetria nativa de um ambiente de computação em nuvem, recorrendo-se ao um modelo de aprendizagem de máquina para este fim.

1.1 Justificativas

A crescente incidência de ataques de negação de serviço e grande dificuldade na identificação daqueles que atacam a camada de aplicação tornam necessário a busca por técnicas mais avançadas para a resolução, voltando-se especialmente para aquelas que possam envolver tecnologias de computação em nuvem.

Em Yan e outros (2016), são mencionadas nuvens que providenciam redes como serviço (NaaS, do inglês *Networking-as-a-Service*), utilizando um controle total de sua infraestrutura de redes, esse estabelecido através de redes definidas em *software* (SDN, do inglês *software defined network*). Com isso, tem-se algumas vantagens na defesa contra ataques DDoS nestes ambientes em nuvem, nos quais é possível disponibilizar controladores centrais que possuem visão completa da rede, monitorando o tráfego desta em busca de ameaças e com resposta imediata às mesmas.

Muitos trabalhos propõem utilizar o protocolo de comunicações OpenFlow para se obter dados para análise e subsequente detecção e mitigação de ataques DDoS, como FortNOX e AVANTGUARD (YAN; YU, 2015). Outros, como Dayal e Srivastava (2018), buscam empregar métodos de aprendizado e mineração de dados para otimizar técnicas tradicionais de detecção de ataques DDoS.

1.2 Objetivos

1.2.1 Objetivo Geral

O objetivo geral deste trabalho é definir a viabilidade do uso de dados provenientes de telemetria de nuvem, especificamente da plataforma *OpenStack*, na detecção de ataques de DoS causados por injeção de SQL.

1.2.2 Objetivos Específicos

Para a execução deste projeto, tem-se os objetivos específicos a seguir:

- a) Construir uma topologia lógica para geração de dados de tráfego de rede, simulando um tráfego considerado normal, composto de requisições legítimas à aplicação Web, e outro que objetiva replicar um ataque DoS por injeção de SQL;
- b) Construir o mecanismo para a coleta dos dados de telemetria presentes no banco de dados Gnocchi da plataforma *OpenStack* utilizada nos experimentos;
- c) Criação de modelos ML treinados com os dados coletados, que possam fazer uma detecção efetiva de ataques DoS realizados através da injeção de SQL.

2 EMBASAMENTO TEÓRICO

Este trabalho utilizará como plataforma de testes e arrecadação de dados um ambiente de nuvem privado, simulando um ataque DoS. Tendo isto em mente, faz-se necessário o entendimento do conceito de computação em nuvem e a plataforma *OpenStack*, utilizada neste projeto para obter a telemetria necessária.

O trabalho desenvolvido busca utilizar de algoritmos de *machine learning* para detectar um tipo de ataque específico em bancos de dados SQL, evitando a indisponibilidade deste para acesso de usuários legítimos. Com isso, é necessário definir conceitos como ataques DoS, injeção de SQL, segurança da informação e aprendizagem de máquina.

2.1 Computação em Nuvem

De acordo com o Instituto Nacional de Padrões e Tecnologia americano (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, 2011), a computação em nuvem é definida como:

“[...] um modelo para permitir acesso onipresente, conveniente e sob demanda a um conjunto compartilhado de recursos de computação configuráveis (por exemplo, redes, servidores, armazenamento, aplicativos e serviços) que podem ser provisionados e liberados rapidamente com o mínimo de esforço de gerenciamento ou interação com o provedor de serviços.” (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, 2011, p. 6, tradução nossa).

Esses recursos gerenciáveis são acessados através de um portal *web*, alocados sobre uma série de conjuntos de *hardware*, *software*, plataformas de desenvolvimento e serviços, estes compondo grandes *data centers*. Esta conexão, entre os serviços disponíveis ao operador, servidores e unidades de processamento a serem utilizadas, formam a estrutura da nuvem.

Dessa forma, um cliente que deseja utilizar dos serviços disponíveis não requer mais um alto poder de processamento e armazenamento local, podendo complementar ou substituir sua infraestrutura, garantindo uma maior personalização do seu ambiente. Para os provedores de serviço na nuvem e seus clientes, existem algumas modalidades de serviços que são detalhadas a seguir (SOFTLINE GROUP, 2017):

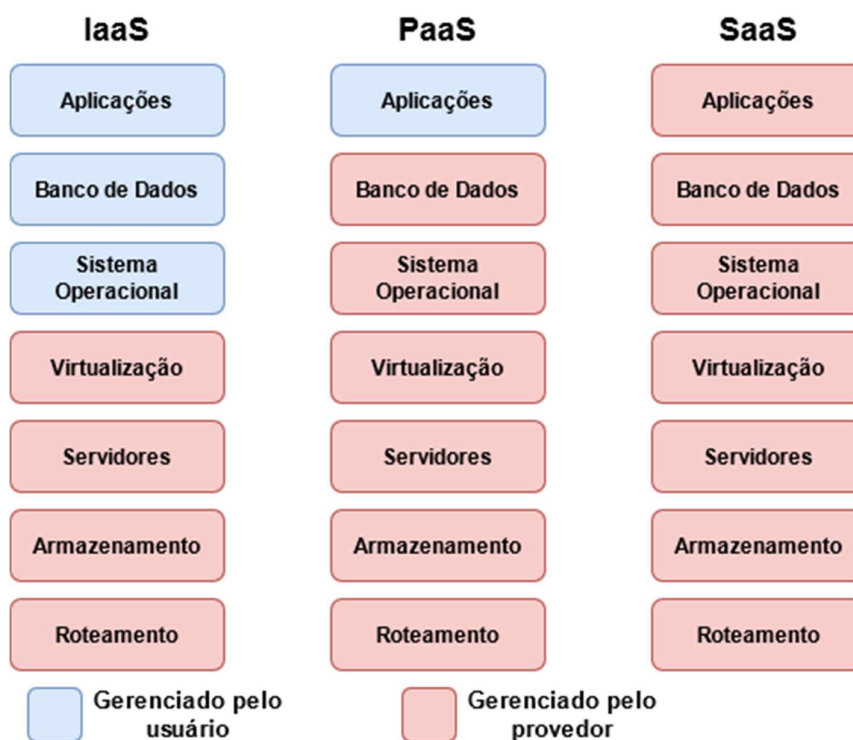
- Infraestrutura como serviço: IaaS (do inglês, *Infrastructure as a Service*) é o modelo em que o provedor oferta recursos computacionais, como infraestrutura de rede, servidores ou

armazenamento, como serviços ao cliente. Estes são tipicamente acessados através de um portal *web* ou API, dando controle total dos recursos ao cliente, obtendo as mesmas capacidades que um *data center* tradicional sem ter que gerenciá-lo fisicamente;

- Plataforma como serviço: PaaS (do inglês, *Platform as a Service*) permite ao provedor fornecer ambientes de pré-configurados a seus clientes, que não necessitam de fazer a manutenção e gerenciamento da infraestrutura. Dessa forma, este modelo é mais voltado a desenvolvedores de aplicações e *software*, possibilitando uma entrega mais rápida de seus produtos sem as preocupações e tarefas de manter um ambiente atualizado e seguro;
- *Software* como serviço: SaaS (do inglês, *Software as a Service*) permite a utilização de *softwares* por meio da Internet sem a necessidade de instalação destes localmente. De um modo geral, é utilizado um modelo de subscrição para o acesso a essas aplicações, cujos recursos para sua operação são gerenciados pelo fornecedor do serviço. Normalmente, o usuário final não possui conhecimento sobre a infraestrutura operante sob a aplicação.

Estas três modalidades de serviços são extremamente utilizadas nos dias de hoje, seja por grandes, médias ou pequenas empresas e instituições, objetivando a otimização de seus processos e serviços. A Figura 2 ilustra as diferenças gerais entre os 3 modelos mencionados anteriormente, demonstrando quais partes da infraestrutura em nuvem o usuário necessita de gerenciar a depender de sua modalidade de serviço.

Figura 2 – Modalidades de serviço em nuvem



Fonte: Produção própria do autor.

2.2 OpenStack

O alto custo de manter uma infraestrutura local para fornecimento de armazenamento, processamento e outras necessidades de organizações leva estas a buscarem recursos através da computação em nuvem. Dentre as várias plataformas que são possíveis de criar este tipo de serviço, o *OpenStack* (OPENSTACK, 2010) se destaca como uma opção de código aberto para nuvens públicas e privadas, dando ao usuário todos os serviços disponíveis em um serviço modelo IaaS. Com APIs compatíveis com outros provedores públicos, como a AWS (do inglês *Amazon Web Services*), fornecendo uma maior flexibilidade e usabilidade em um ambiente híbrido.

A plataforma tem seu código-fonte primariamente em Python (PYTHON SOFTWARE FOUNDATION, 2020), sendo composta de múltiplos componentes, ou serviços, que foram sendo adicionados a cada nova iteração da mesma. O Quadro 1, apresentado a seguir, demonstra alguns dos serviços mais comuns presentes na arquitetura do *OpenStack*, dentre as dezenas que o projeto atualmente possui.

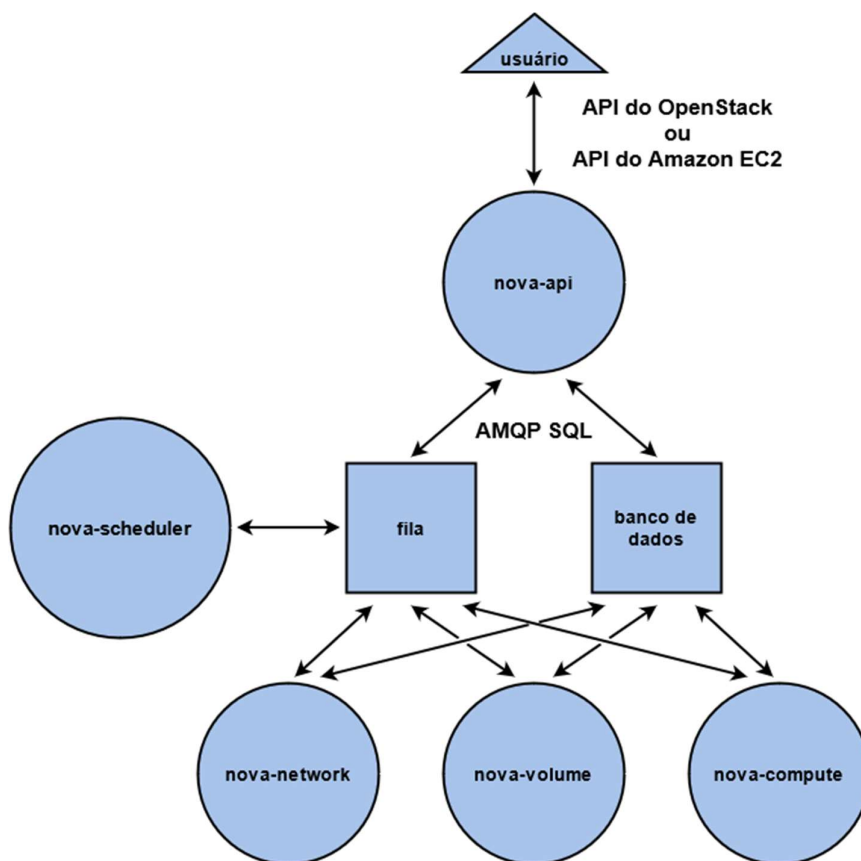
Quadro 1 – Serviços *OpenStack*

Serviço	Função
Nova	Implementa serviços e módulos necessários para providenciar recursos computacionais, VMs e contêineres.
Swift	Utilizado para armazenar grandes volumes de dados e objetos, de forma eficiente e segura.
Cinder	Virtualiza o gerenciamento de armazenamento em blocos.
Neutron	Fornecer serviços de redes definidas em <i>software</i> para os ambientes virtuais.
Keystone	Providencia uma API para autenticação dos clientes.
Glance	Inclui a criação e registro de imagens de VMs, através de uma API RESTful.
Heat	Orquestração de recursos através de APIs.
Horizon	<i>Dashboard</i> web para utilização da plataforma.

Fonte: Produção própria do autor.

Cada um destes serviços *OpenStack* também podem ser subdivididos em outros subcomponentes, cada um responsável por determinada ação do serviço. Por exemplo, o Nova é composto por 7 componentes principais, incluindo Nova-API, Nova-Compute, Nova-Network, Nova-Scheduler, Nova-Volume, Message Queue e um banco de dados SQL, como pode ser observado na Figura 3.

Figura 3 – Arquitetura do serviço Nova



Fonte: Pepple (2011).

Nota: Adaptado pelo autor.

Para serviços de telemetria, o *OpenStack* possui o projeto *Ceilometer*, cuja função é coletar dados produzidos de outros serviços da plataforma, desde informações do uso de cache L3 da CPU, até quantidade de operações de escrita/leitura no disco e conexões ativas na rede. Tais métricas podem ser então serem recolhidas e utilizadas para novas visualizações sobre a nuvem, construindo casos de uso apropriados ao usuário.

2.3 Segurança da Informação

Atualmente, tem-se um mundo cada vez mais conectado por meio da Internet ou outros serviços de rede, cuja moeda de troca é a informação, seja esta os dados de clientes ou conhecimentos de cunho sigiloso, incentivando empresas e indivíduos a protegerem essas mesmas informações. Torna-se, então, cada vez mais necessário manter essas e outros serviços seguros mediante investimentos e aplicação da segurança da informação.

Entende-se por segurança da informação como “um conjunto de medidas que possuem o objetivo de tornar as informações mais seguras” (VECCHIA, 2020), protegendo estas durante seu uso, armazenamento ou transmissão. Essa proteção pode ser alcançada com a aplicação e desenvolvimento de políticas, tecnologias e mecanismos para manter essas informações livres de ameaças. Estas ameaças podem ser classificadas de acordo com a forma que elas buscam afetar a informação impactada, o que inclui a destruição, modificação, acesso não-autorizado ou indisponibilidade da informação.

Assim, faz-se necessário a utilização da tríade CIA (do inglês *Confidentiality, Integrity and Availability*), os 3 pilares que são conceitos fundamentais da segurança da informação (VECCHIA, 2020):

- Confidencialidade (do inglês *confidentiality*): refere-se ao conceito da proteção dos dados contra aqueles não-autorizados a acessá-los, sendo necessário a definição e aplicação de controles de acesso à informação;
- Integridade (do inglês *integrity*): entende-se como o conceito da integridade dos dados, no que tange a sua modificação ou destruição por terceiros não-autorizados, além da possibilidade de reversão quando modificações danosas são realizadas. Dentro de integridade também está incluso o não-repúdio, que confirma a origem de uma informação;
- Disponibilidade (do inglês *availability*): neste pilar, é dado que a informação requerida pode ser sempre acessada quando necessário, incluindo canais para uso da mesma e mecanismos de autenticação. A perda de disponibilidade pode ser acarretada por variados motivos, como perda de energia, problemas de aplicação ou ataques DDoS por agentes maliciosos.

2.4 Ataques de Negação de Serviço

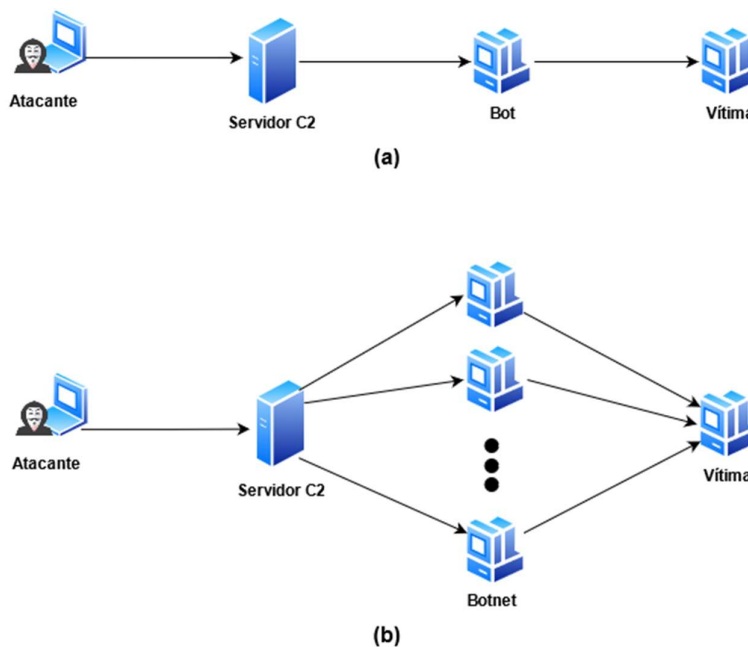
Ataques de negação de serviço são um problema cada vez mais frequente para empresas e outras organizações nos tempos atuais, impactando seus negócios e serviços. Estes ataques ameaçam um dos pilares da tríade CIA, mais especificamente, a Disponibilidade. De acordo com a Agência de Cibersegurança e Segurança de Infraestrutura (CISA, do inglês *Cybersecurity and Infrastructure Security Agency*, 2009), este tipo de ataque ocorre quando o acesso por usuários legítimos a algum tipo de recurso, seja este um serviço de e-mail, uma aplicação *web* ou até servidores inteiros, torna-se indisponível devido às ações de agentes maliciosos.

Este tipo de ameaça pode ser concretizado de duas formas: seja apenas um atacante causando a indisponibilidade do alvo ou com múltiplas máquinas realizando um ataque de forma distribuída (ataques DDoS). Atualmente, este último é o método o mais utilizado nos ataques, devido à sua capacidade de gerar um grande volume de tráfego, causando a indisponibilidade do alvo mais rapidamente.

Dessa forma, entende-se que o atacante busca esgotar recursos do alvo para que o mesmo não possa responder a quaisquer requisições de usuário legítimos. Estes recursos podem ser variados, como a capacidade de processamento e memória do alvo, a habilidade da aplicação de atender solicitações ou a largura de banda do alvo.

A organização mais comum para esse tipo de ataque é a utilização de um servidor de comando e controle (C2, do inglês *Command and Control*) sob domínio do atacante, sendo o “cérebro” do ataque. O atacante gera comandos ao servidor C2 que então envia instruções a um ou mais máquinas infectadas, cunhados de *bots*, que obedecem às instruções dadas. Estes *bots*, por sua vez, são as máquinas que efetivamente geram a indisponibilidade na vítima. Na Figura 4, pode-se observar como funciona um ataque DoS (Figura 4(a)) e um ataque DDoS (Figura 4(b)).

Figura 4 – Ataques (a) DoS e (b) DDoS



Fonte: Produção própria do autor.

Em geral, nos ataques DDoS, tem-se múltiplos *bots* formando uma *botnet*, um conjunto de máquinas “zumbis” que, em geral, sofreram algum tipo de ataque para serem infectadas de tal forma, seja através de vírus Trojan, *worms*, ataques utilizando *fileless malware* e outros. Após o estabelecimento da comunicação entre o servidor C2 e a *botnet*, as máquinas infectadas enviam requisições coordenadas à vítima, sobrecarregando-a e a impedindo de responder usuários legítimos.

Além da maior capacidade de gerar um grande volume de tráfego, é notoriamente difícil de se detectar a origem de ataques DDoS, tanto devido a técnicas de ocultação, como o *spoofing* de endereços IPs, quanto a dificuldade de se distinguir entre um verdadeiro ataque ou um grande tráfego repentino de clientes legítimos, este chamado de *Flash Events* (DHINGRA; SACHDEVA, 2018). Estes são eventos nos quais existe um grande número de requisições feitas por usuários legítimos a alguma aplicação, serviço ou servidor. Isto gera uma grande carga sobre os recursos computacionais do ativo afetado, causando efeitos similares a um ataque de negação de serviço.

Sabe-se que ataques DoS podem ser realizados de diferentes formas, afetando recursos ou utilizando vetores de ataque diferentes sendo, então, necessária uma classificação apropriada destes ataques. Em geral, existem duas formas de causar indisponibilidade em um alvo: o atacante pode esgotar recursos como memória RAM e CPU, desabilitando todos os serviços que estão hospedados na vítima ou tornando uma aplicação em específico indisponível. Este segundo são ataques DDoS que visam atingir a camada de aplicação na vítima, se utilizando de vulnerabilidades no serviço para alcançar seu objetivo (PRASEED; THILAGAM, 2019). Ao contrário de ataques mais tradicionais, estes não necessitam de uma grande quantidade de tráfego para causar a indisponibilidade, assemelhando-se a um cliente legítimo. Dessa forma, métodos usuais de detecção não são eficientes para este tipo de ataque.

Um dos tipos de ataque de negação de serviço na camada de aplicação são os chamados ataques *low-rate*, que serão explicados a seguir. Estes têm como sua maior característica o baixo tráfego de pacotes, semelhante a um usuário legítimo, um fator que dificulta em muito a sua detecção (ZHIJUN et al., 2020). Este tipo de ataque explora vulnerabilidades nos protocolos utilizados pelas aplicações, removendo o espaço para requisições de usuários legítimos.

Um exemplo de ataque *low-rate* seria o *Slowloris* (SHOREY et al., 2018), que busca manter múltiplas conexões com o servidor *web* alvo. Isso é feito através do envio de múltiplas requisições HTTP incompletas, sendo as conexões periodicamente renovadas por outras requisições à vítima. Dessa forma, o limite de conexões simultâneas da aplicação *web* é atingido e não é possível de outros clientes se conectarem ao serviço.

2.5 Injeção de SQL

SQL é uma linguagem utilizada para gerenciamento de dados dentro de bancos de dados relacionais, codificada em 1987 pela Organização Internacional pela Padronização (ISO, do inglês *International Organization for Standardization*) como o padrão ISO 9075:1987, tendo este sido revisado pela última vez em 2016. Sendo utilizada em diferentes bancos de dados, como *MySQL*, *Oracle*, *PostgresSQL* e outros, não é surpreendente que a linguagem tenha presença em inúmeras aplicações interativas, desde sistemas financeiros até lojas virtuais *online*.

Ataques de injeção de SQL (SQLi, do inglês *SQL Injection*) constituem uma das maiores ameaças para estas aplicações (OPEN WEB APPLICATION SECURITY PROJECT, 2017), permitindo que agentes maliciosos obtenham acesso direto aos dados ali contidos. Qualquer aplicação *web* sem validação suficiente de entradas pode estar vulnerável a este ataque, o que permite que o atacante possa dar comandos arbitrários ao banco de dados, podendo destruir o mesmo, modificá-lo ou simplesmente obter seu conteúdo.

De acordo com Ma e outros (2019), existem dois tipos de ataques SQLi:

- O atacante injeta código malicioso diretamente nas variáveis de entrada que, por sua vez, estão concatenadas com o comando SQL que é então executado junto do código do atacante.
- O código malicioso é utilizado através de um ataque indireto, sendo armazenado em tabelas ou documentos dentro do banco de dados. Esta *string* armazenada é inserida em algum código SQL dinâmico, sendo o código então executado.

Estes ataques podem ocorrer por inúmeras razões, sendo a seguir mostrado o ataque *SQL Injection* por meio da utilização de caracteres de escape. Este ocorre quando a entrada não filtra caracteres de escape e um atacante pode utilizá-los para injetar código SQL diretamente na

aplicação. Por exemplo, uma linha de código que receba uma variável *searchName* para procurar numa lista de usuários:

```
$USER = "SELECT * FROM users WHERE name = '$searchName'";
```

Verificando este código, é possível perceber que foi feito para procurar um usuário com certo nome em uma tabela, mas um atacante pode escapar utilizando uma entrada de aspas simples, permitindo a injeção de código posteriormente. Por exemplo, fazendo com que a variável *searchName* seja:

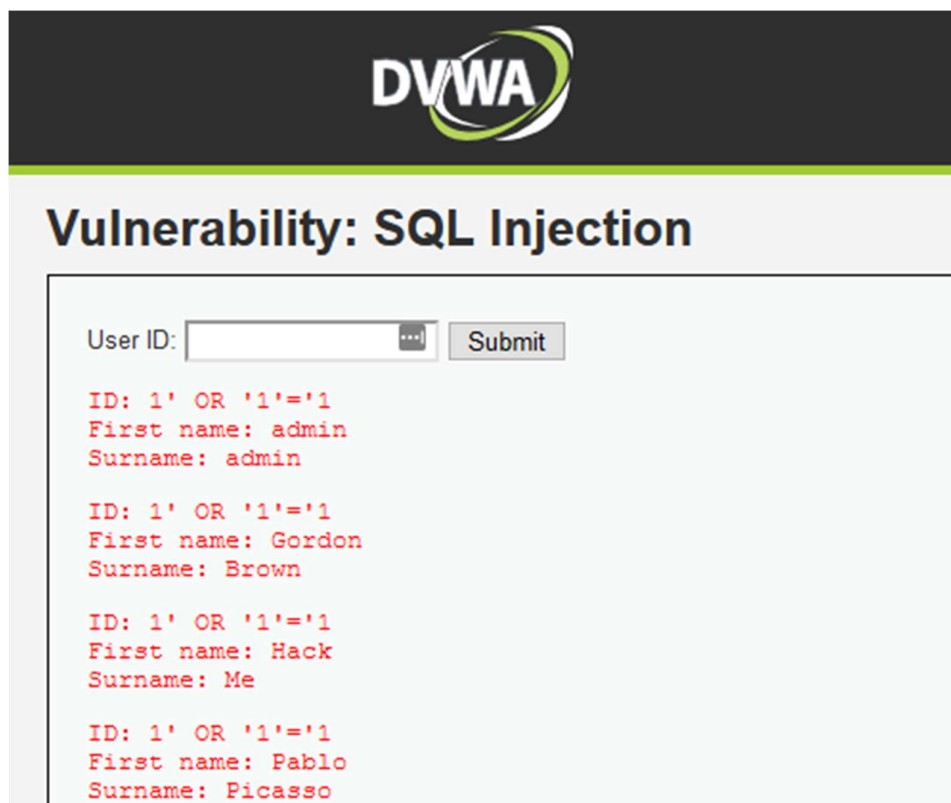
```
1' OR '1'='1'
```

O código ficará como

```
$USER = "SELECT * FROM users WHERE name = '1' OR '1'='1'";
```

Como o atacante manipulou a entrada para que tenha a expressão `'1'='1'`, sendo esta considerada sempre uma expressão verdadeira, ele valida a busca, podendo listar todos os usuários do banco de dados. A execução deste ataque pode ser verificada na Figura 5, utilizando-se de uma aplicação *web* intencionalmente vulnerável.

Figura 5 – Injeção de SQL para listar todos os usuários da aplicação



Fonte: Produção própria do autor.

Além disso, algumas aplicações permitem a execução de múltiplos comandos em uma mesma *query*. Com isso, apenas a imaginação e o conhecimento do atacante são limites para o que pode ser feito com a aplicação vulnerável.

2.6 Aprendizagem de Máquina

Aprendizagem de máquina (ML, do inglês *Machine Learning*) é o estudo da aplicação de algoritmos e modelos estatísticos para que um sistema possa automaticamente aprender e otimizar uma dada tarefa (GOODFELLOW et al., 2016). Deve-se ter em mente que essa otimização pode ser realizada de duas formas, apresentadas a seguir.

- **Aprendizagem supervisionada:** aqui o algoritmo utiliza de dados rotulados, isto é, cada entrada a ser manuseada é previamente mapeada a uma classe, categoria ou valor. Quando otimizado, o modelo treinado pode então determinar a classe de novos dados não-rotulados. Um exemplo deste tipo de algoritmo são máquinas de vetores de suporte.

- **Aprendizagem não-supervisionada:** esta forma emprega algoritmos que não operam com dados previamente rotulados, estes vindo de forma “bruta”. Dessa forma, o modelo trabalha para descobrir padrões e outras informações de forma desassistida. Redes neurais são exemplos deste tipo de aprendizagem.

Existem vários tipos de tarefas que podem ser otimizadas, sendo alguns exemplos destas dadas a seguir:

- **Classificação:** nesta tarefa, o sistema precisa classificar a qual categoria ou categorias uma determinada entrada pertence. Cada algoritmo pode fazer essa classificação de formas diferentes, por exemplo, o *K-Nearest Neighbors* atribui uma distribuição de probabilidades para as categorias definidas. Um exemplo de problema de classificação seria o reconhecimento automático de um conjunto de objetos, classificando-os entre cubos, esferas e outros;
- **Regressão:** é pedido ao sistema prever um valor numérico de saída dada alguma entrada, tendo o algoritmo fazendo o mapeamento de uma função $f: R^n \rightarrow R$. Emprega-se a regressão para um problema de predição de um valor de seguro, dadas certas circunstâncias do comprador;

Em geral, os algoritmos de *machine learning* se utilizam de conjuntos de dados para fazer sua otimização, sendo classificados como algoritmos supervisionados ou não-supervisionados. Esta segunda contém algoritmos que usam o banco de dados e “aprendem” características sobre os mesmos. Os algoritmos supervisionados manipulam dados que já são classificados, efetivamente guiando o modelo a produzir uma função que, dados as entradas do conjunto, produza-se os mesmos resultados da classificação dada.

Deve-se ter em mente que o algoritmo precisa ser capaz de generalização, a habilidade de se obter um bom desempenho com dados não usados previamente. O desafio, então, é providenciar que o algoritmo seja adequadamente treinado com um conjunto de dados variados para obter essa característica de generalização. Para que isso seja testado, utilizam-se dados específicos para tal objetivo, chamado de conjunto de testes, no qual o modelo será aplicado e poderá ser verificado quão próximo da classificação correta o modelo chegou.

3 METODOLOGIA

3.1 Introdução

O trabalho em questão objetiva verificar a viabilidade de utilizar dados de telemetria do *OpenStack* para a detecção de ataques DoS por *SQL Injection*. Para o desenvolvimento da solução, faz-se necessário o treinamento de modelos *machine learning* para classificação dos dados obtidos da plataforma, avaliando-se os resultados obtidos.

Dessa forma, este projeto é classificado como uma pesquisa exploratória quanto aos seus objetivos, sendo também definido como uma pesquisa experimental no que diz respeito a procedimentos técnicos. Finalmente, a metodologia aplicada tem uma abordagem quantitativa e de natureza aplicada, visto que tem a finalidade de produzir conhecimento acerca de problema específico.

3.2 Geração do *Dataset*

Nesta etapa são feitas as coletas de dados necessárias para o treinamento do modelo ML e para a classificação binária entre tráfego normal e malicioso. Para este fim, foram desenvolvidos *scripts* Python que estabeleceriam uma conexão com as máquinas virtuais hospedadas na plataforma *OpenStack*, criando uma rotina onde seriam gerados ataques SQLi, além de requisições legítimas à página da aplicação web. Ambos os tipos de tráfego são produzidos com variadas intensidades, classificadas entre “Ataque forte”, “Ataque fraco”, “Ataque médio”, “Cliente forte”, “Cliente fraco” e “Cliente médio”, visto que o objetivo era simular um cenário realístico no experimento. As definições para estes diferentes tipos de tráfego são dadas no Quadro 2 a seguir.

Quadro 2 – Tráfegos ao servidor Web

(continua)

Tipo de tráfego	Descrição
Ataque forte	Tráfego com quantidade de ataques SQLi para causar total indisponibilidade na aplicação Web
Ataque médio	50% do total de requisições feitas pelo “Ataque forte”
Ataque fraco	25% do total de requisições feitas pelo “Ataque forte”

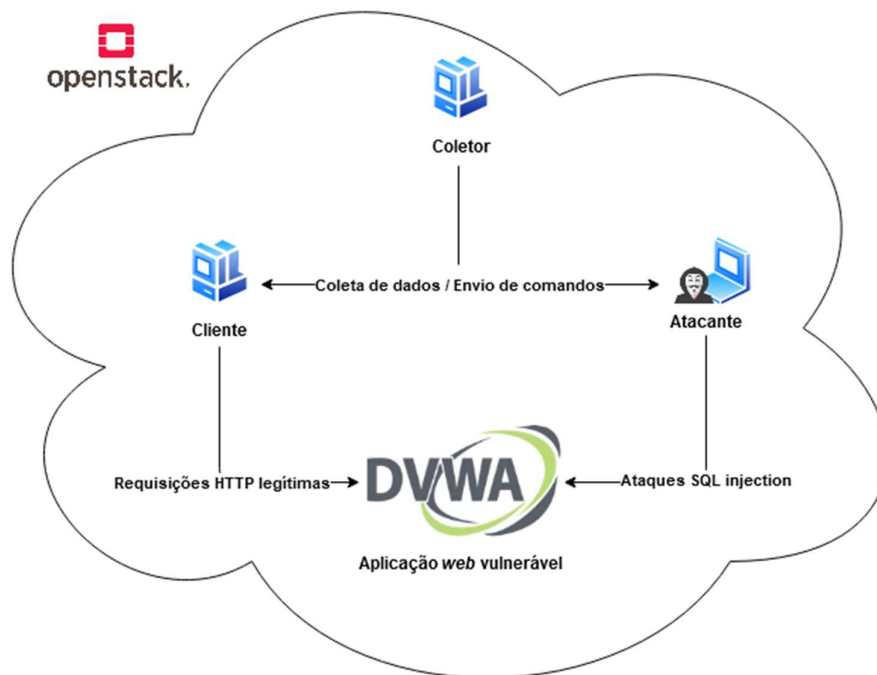
Quadro 2 – Tráfegos ao servidor Web

(conclusão)

Tipo de tráfego	Descrição
Cliente forte	Tráfego com quantidade de requisições à aplicação, tal que inicia a perda de requisições devido ao volume das mesmas
Cliente médio	50% do total de tráfego gerado pelo “Cliente forte”
Cliente fraco	25% do total de tráfego gerado pelo “Cliente forte”

Fonte: Produção própria do autor.

No experimento, foram criadas 4 máquinas virtuais, simulando o seguinte cenário: 1 atacante que produziria o tráfego DoS, 1 cliente responsável por fazer chamadas consideradas inofensivas e o servidor que hospedaria a aplicação vulnerável. Além destes, há uma máquina responsável por enviar comandos ao atacante e cliente, enquanto recolhe os dados de telemetria da nuvem, chama de “Coletor”. A Figura 6 ilustra a topologia utilizada nos experimentos.

Figura 6 – Topologia lógica dentro da plataforma *OpenStack*

Fonte: Produção própria do autor.

Para gerar os dados, a rotina de criação de dados buscava fazer combinações entre tráfego de cliente e aquele causado por ataques, e.g “Cliente fraco” + “Ataque forte”, junto de tráfego composto somente de requisições legítimas ou somente SQLi. Com isso, tem-se um total de 24 combinações possíveis de tráfego, com o script rotulando o tráfego gerado como 1 quando este

era somente ataque ou alguma combinação cliente e ataque, e 0 para quando o tráfego constituído apenas de requisições do cliente.

Para a geração da base de dados, cada uma das combinações descritas anteriormente era monitorada por 1 hora, com a máquina coletora se conectando ao banco de dados Gnocchi e recolhendo as métricas de telemetria. Com 24 combinações, cada chamada ao script durava 24h, com este processo sendo repetido por 5 dias. Neste tempo, foram coletadas certas métricas do *Ceilometer*, especificamente as apresentadas no Quadro 3.

Quadro 3 – Métricas coletadas

Métrica	Descrição
cpu	Tempo de CPU utilizado em nanosegundos
memory	Volume de memória RAM alocada, em megabytes
memory_swap_in	Memória <i>swap</i> (entrada), em megabytes
memory_swap_out	Memória <i>swap</i> (saída), em megabytes
disk_request_read	Nº de requisições de leitura de disco, em bytes
disk_request_write	Nº de requisições de escrita de disco, em bytes
interface_bytes_in	Nº de <i>bytes</i> de rede (entrada)
interface_bytes_out	Nº de <i>bytes</i> de rede (saída)
interface_packets_in	Nº de pacotes de rede (entrada)
interface_packets_out	Nº de pacotes de rede (saída)

Fonte: Produção própria do autor.

3.3 Pré-Processamento

As métricas recolhidas foram tratadas de duas formas como parte de seu pré-processamento. Primeiro, algumas instâncias dos dados retornaram com valor “-1”, que indica não ter sido possível obter o real valor, potencialmente devido à um *buffer* interno de leitura na plataforma. Estes valores foram corrigidos fazendo uma média aritmética entre os dois valores anteriores quando possível, ou igualando ao valor anterior caso contrário.

Segundo, foi observado que, devido à natureza de cada métrica, estas possuíam escalas variadas de valores, o que poderia resultar em certas variáveis tendo uma maior influência durante o treinamento. Com isso, foi feita uma normalização dos dados coletados, utilizando a seguinte equação Min-Max (1).

$$valor_{normalizado} = \frac{valor_{original} - valor_{mínimo}}{valor_{máximo} - valor_{mínimo}} \quad (1)$$

Dessa forma, todos os dados ficam numa faixa de valores entre 0 e 1, igualando o “peso” que cada variável terá no treinamento do modelo. Por fim, foi observado que as métricas “memory_swap_in” e “memory_swap_out” retornaram apenas valores nulos, sendo conseqüentemente removidas.

Após o tratamento dos dados, pôde-se verificar que havia um total de 250.042 entradas para o treinamento dos modelos. Desse total, 124.343 (49,73%) foram rotuladas como tráfego apenas de cliente (“0”) e o restante, 125.699 entradas (50,27%), foram classificadas como tráfego contendo ataque (“1”).

3.4 Treinamento de Modelos e Classificação dos Dados

Aqui, escolhem-se alguns algoritmos supervisionados de *machine learning* para identificação da presença ou não de ataques SQL *Injection*, explicitados a seguir:

- a) Árvores de decisão;
- b) *K-Nearest Neighbors*;
- c) *Multi Layer Perceptron*;
- d) Naïve Bayes gaussiano;
- e) *Random Forest*.

Os dados coletados são divididos em duas partes, em uma proporção 70 : 30, onde 70% do *dataset* forma um conjunto de treinamento e o restante é utilizado como um conjunto de testes. Dessa forma, o primeiro é inicialmente utilizado para preparar os parâmetros em relação às métricas recolhidas, com o segundo sendo usado para performar uma avaliação do modelo treinado.

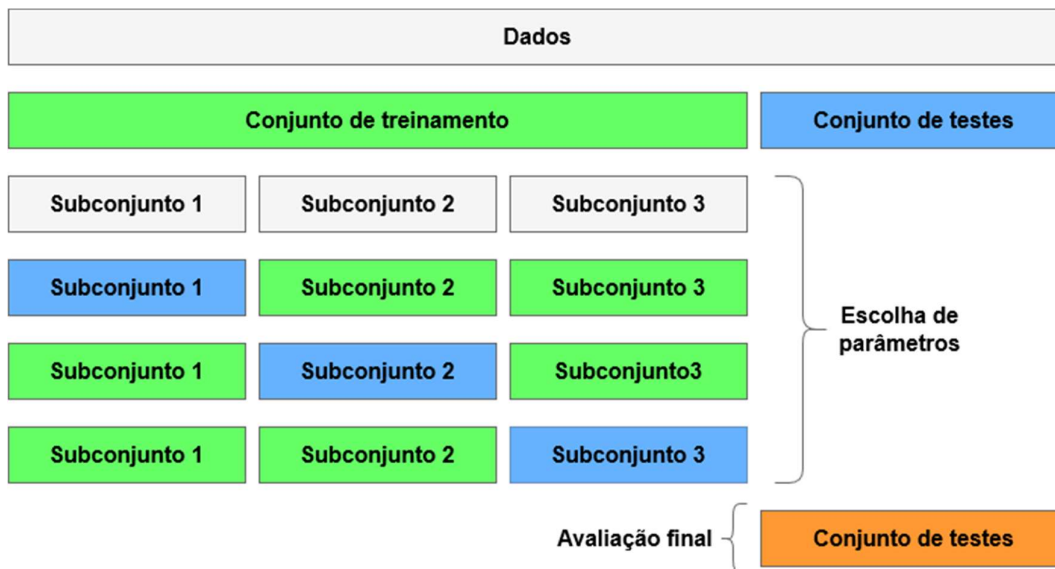
Para a escolha dos hiperparâmetros, estes definindo as condições iniciais de cada modelo, foram definidos múltiplos intervalos. Estes intervalos possuíam valores diferenciados para cada hiperparâmetro: por exemplo, no treinamento de um MLP, é necessário definir a função de

ativação para as camadas. Dessa forma, é definida uma lista com todas as possíveis escolhas de função, repetindo o mesmo processo para outras características.

Com os intervalos estipulados, pode-se então efetuar uma busca pela escolha de valores de cada modelo que obtém a melhor performance. Para este fim, é realizado múltiplas iterações de treinamento, nas quais os hiperparâmetros iniciais são selecionados aleatoriamente a partir dos intervalos previamente definidos, treinando-se o modelo com estas características escolhidas.

Entretanto, para se evitar problemas de *overfitting* no conjunto de testes e que o modelo ainda possua uma desejada capacidade de generalização, cada treinamento também passará por um processo de validação cruzada, em que o *dataset* de treinamento seria dividido em 3 partes iguais. Destas, 2 serão utilizadas para o treinamento apropriado e a última sendo empregada para validar o modelo treinado. Este processo é então repetido mais duas vezes, com os outros dois subconjuntos de dados sendo utilizado para a etapa de validação. Este processo é ilustrado na Figura 7.

Figura 7 – Processo de validação cruzada (com $K = 3$)



Fonte: Produção própria do autor.

Finalmente, cada um dos modelos terá um total de 200 iterações com escolhas aleatórias de hiperparâmetros e, ao final dessas, são escolhidos aqueles cujo estimador possui, em média, a maior acurácia encontrada ao fim de cada treinamento. O Quadro 4 a seguir nos retorna quais

parâmetros foram selecionados para cada algoritmo (com nomenclatura das funções do Scikit-learn).

Quadro 4 – Hiperparâmetros selecionados

Modelo	Hiperparâmetros
Árvore de Decisão	splitter: best, min_samples_split: 2, min_samples_leaf: 1, max_depth: 10, criterion: gini
Naïve-Bayes gaussiano	var_smoothing: 1e-20
<i>K_Nearest Neighbors</i>	weights: distance, n_neighbors: 20, leaf_size: 35, algorithm: ball tree
<i>Multi Layer Perceptron</i>	max_iter: 600, learning_rate_init: 0.002, learning_rate: invscaling, hidden_layer_sizes: (30,), activation: relu
<i>Random Forest</i>	n_estimators: 140, min_samples_split: 2, min_samples_leaf: 1, max_depth: 100, criterion: entropy

Fonte: Produção própria do autor.

3.5 Recursos Computacionais

Recursos de hardware: Uma máquina pessoal com sistema operacional Windows 10; processador Intel Core i5-7200U, 2.5GHz e 2 núcleos físicos; 12 GB de memória RAM e armazenamento de 1TB de disco rígido. O segundo seria a plataforma de computação em nuvem *OpenStack*, hospedada nos servidores utilizados pelo laboratório NERDS, localizado no CT VI da Universidade Federal do Espírito Santo. Esta foi utilizada para hospedar 3 máquinas virtuais, operando com sistema operacional Ubuntu Desktop 18.04 para a máquina do cliente, Ubuntu Server 18.04 para o servidor Web, Kali Linux versão 2020.1 para a máquina atacante, com todas as máquinas possuindo 4 GB de memória RAM e 2 vCPUS cada. Já o treinamento dos modelos foi efetuado em uma máquina virtual, com sistema operacional Ubuntu Server 18.04, 16 GB de memória RAM e processador AMD Ryzen 7 2700, utilizando 4 núcleos virtuais.

Recursos de software: As etapas de geração do *dataset* foi feita através da utilização do Apache JMeter, um *software* de código aberto escrito em Java para realizar testes de estresse em aplicações, assim como o *Damn Vulnerable Web Application* (DVWA), aplicação Web escrita em PHP/MySQL para testes de segurança. A solução proposta baseia-se na linguagem Python,

utilizando-se métodos de classificação da biblioteca Scikit-Learn (PEDREGOSA, 2011). As bibliotecas Pandas (MCKINNEY, 2010) e Matplotlib (HUNTER, 2007) foram amplamente utilizadas nas etapas de pré-processamento e geração dos gráficos, respectivamente.

Os dados utilizados foram obtidos através do *OpenStack*, utilizando-se uma biblioteca Python que permite a comunicação com a API do Gnocchi, o banco de dados utilizado pelo *Ceilometer*. Foi utilizado outro *script* que capturaria, a cada segundo, algumas métricas da telemetria presente dentro da nuvem: (i) uso de CPU; (ii) uso de memória RAM; (iii) requisições de leitura de disco; (iv) requisições de escrito de disco; (v) quantidade de *bytes inbound*; (vi) quantidade de *bytes outbound*; (vii) quantidade de pacotes *inbound*; (viii) quantidade de pacotes *outbound*; (ix) *timestamp*. Também é adicionada um rótulo para cada leitura para indicar a presença ou não de ataque, sendo aquelas feitas através de uma máquina virtual, denominada “Coletor”, enviando comandos para fazer requisições legítimas e ataques *SQL Injection*, para as máquinas “Cliente” e “Atacante” respectivamente.

4 RESULTADOS

Nesta seção são apresentados, comparados e discutidos os resultados obtidos a partir dos experimentos realizados.

4.1 Métricas de Avaliação

Para comparação entre os modelos criados, foram utilizados quatro métricas: acurácia, precisão, *recall* e valor F1. As equações (2), (3), (4) e (5) determinam, respectivamente, como são calculados tais valores, levando-se em conta o total de entradas classificadas corretamente, verdadeiros positivos (VP) e negativos (VN), e as classificadas incorretamente, falsos positivos (FP) e negativos (FN).

$$Acurácia = \frac{número_{VP} + número_{VN}}{número_{VP} + número_{VN} + número_{FP} + número_{FN}} \quad (2)$$

$$Precisão = \frac{número_{VP}}{número_{VP} + número_{FP}} \quad (3)$$

$$Recall = \frac{número_{VP}}{número_{VP} + número_{FN}} \quad (4)$$

$$Valor F1 = 2 \times \frac{Precisão * Recall}{Precisão + Recall} \quad (5)$$

Também serão desenvolvidas as matrizes confusão e as curvas ROC, o primeiro indicando os erros e acertos do modelo treinado sendo comparados com os resultados esperados, e o segundo mede o grau de separabilidade do modelo entre as classes (0 e 1).

4.2 Experimentos

Inicialmente, os modelos de *machine learning* são iniciados sem quaisquer parâmetros iniciais selecionados, visto que estes serão apenas escolhidos após uma busca randômica, feita pela função `RandomizedSearchCV`, da biblioteca Python `Scikit-Learn`. Tal escolha é feita após o

treinamento de cada estimador utilizando o *dataset* de treinamento, como definido dentro do *script* Python.

Após isso, foi utilizado o conjunto de dados de testes para gerar as métricas de avaliação, utilizando o modelo treinado para classificar os estimadores gerados obtendo, de acordo com as equações (2) a (5). A Tabela 1 informa sobre os resultados desta avaliação:

Tabela 1 – Métricas para avaliação dos modelos treinados

Modelo	Acurácia	Precisão	Recall	Valor F1
Árvore de Decisão	92,54%	93,07%	91,98%	92,52%
Naïve-Bayes gaussiano	79,32%	93,41%	63,24%	75,42%
<i>K_Nearest Neighbors</i>	94,22%	94,53%	93,92%	94,22%
<i>Multi Layer Perceptron</i>	92,42%	92,86%	91,96%	92,41%
<i>Random Forest</i>	95,17%	96,20%	94,09%	95,14%

Fonte: Produção própria do autor.

Como pode ser observado, excetuando-se o algoritmo de Naïve Bayes gaussiano, os modelos treinados alcançaram valores significativos, com todas as métricas de avaliação apresentando valores acima de 91%. Entretanto, verifica-se que o algoritmo de *Random Forest* claramente foi o melhor entre os 5 testados, alcançando os maiores resultados nos 4 campos de avaliação.

Esses valores podem ser comparados com alguns resultados de outros trabalhos da literatura, como pode ser visto a seguir:

- a) *Application-Layer DDoS Defense with Reinforcement Learning* (FENG et al. 2020). Obtiveram uma acurácia de cerca de 95,53%, utilizando métodos de aprendizagem por reforço, especificamente processos de decisão de Markov.
- b) *Distributed Denial of Service Attack Detection using Naïve Bayes and K-Nearest Neighbors for Network Forensics* (KACHAVIMATH et al., 2020). Utilizaram-se de

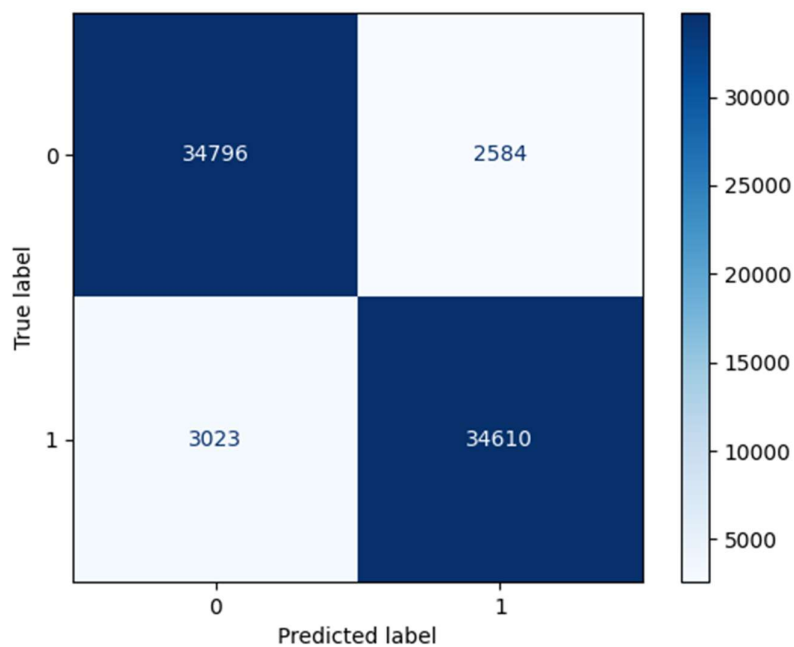
dois modelos: KNN e Naïve Bayes, obtendo uma acurácia de 98,51% e 93,95%, respectivamente.

É importante observar que as literaturas listadas não buscam especificamente a detecção de um ataque *SQL Injection* e fazem utilização de diferentes tipos de dados, com Feng e Nguyen recolhendo *strings* das requisições feitas e o segundo fazendo uso de *datasets* públicos, contendo informações de pacotes de rede.

Além das métricas descritas, também são criadas matrizes confusão para cada modelo, com a quantidade de verdadeiros e falsos positivos quanto aos dados rotulados (0 para tráfego apenas do cliente e 1 para a presença de um ataque DoS).

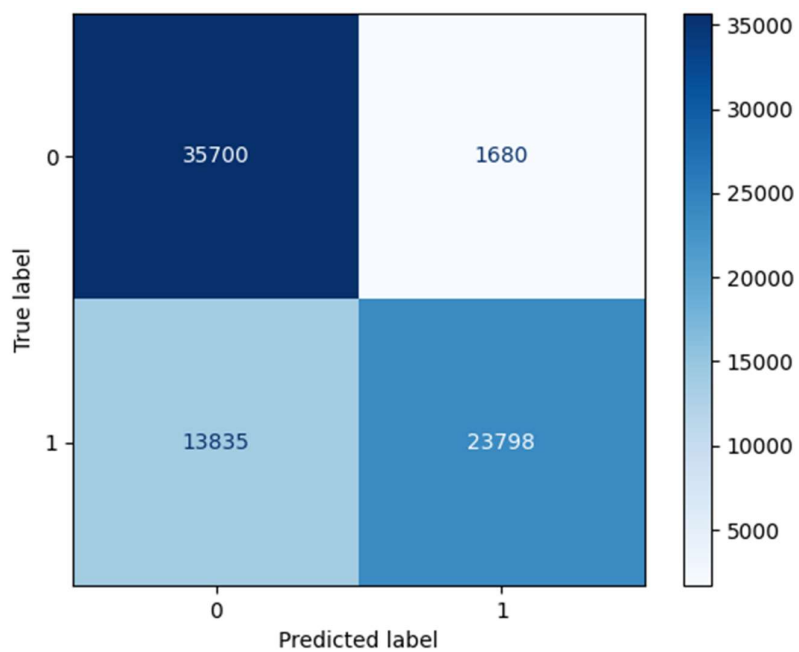
Nas Figuras 8 a 12 a seguir, pode-se observar estas matrizes para cada classificador.

Figura 8 – Matriz confusão do modelo Árvore de Decisão

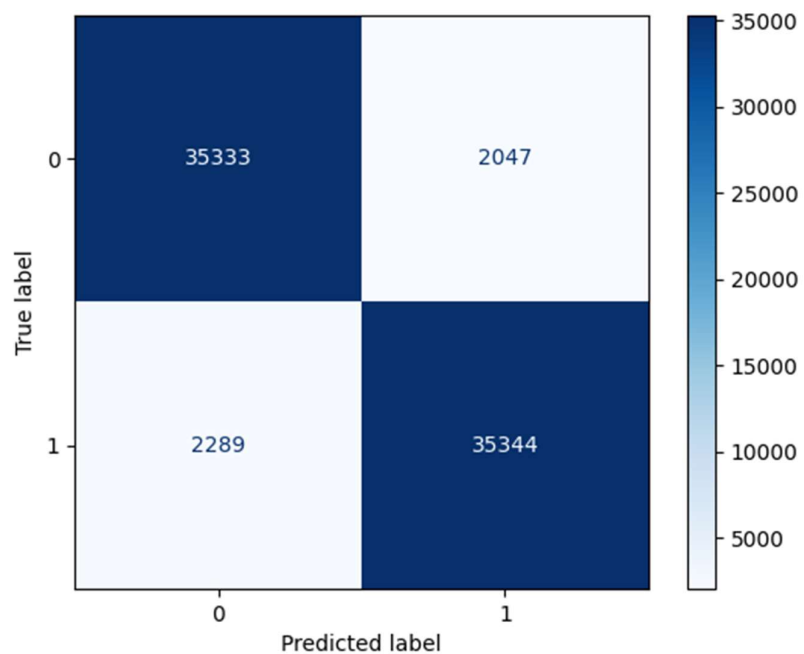


Fonte: Produção própria do autor.

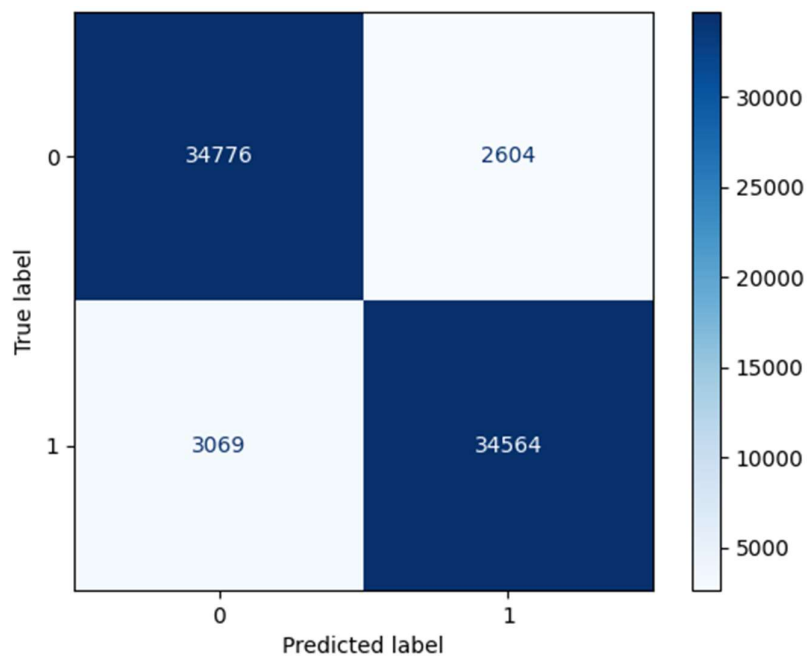
Figura 9 – Matriz confusão do modelo Naïve Bayes gaussiano



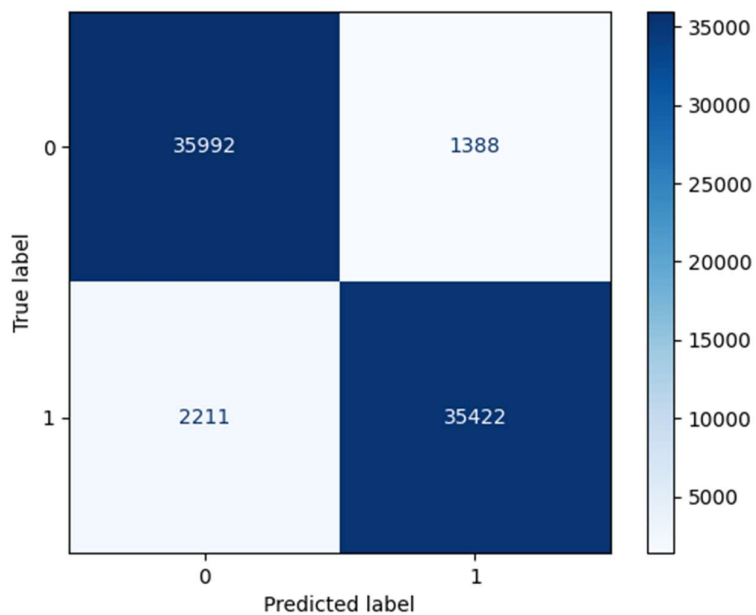
Fonte: Produção própria do autor.

Figura 10 – Matriz confusão do modelo *K-Nearest Neighbors*

Fonte: Produção própria do autor.

Figura 11 – Matriz confusão do modelo *Multi Layer Perceptron*

Fonte: Produção própria do autor.

Figura 12 – Matriz confusão do modelo *Random Forest*

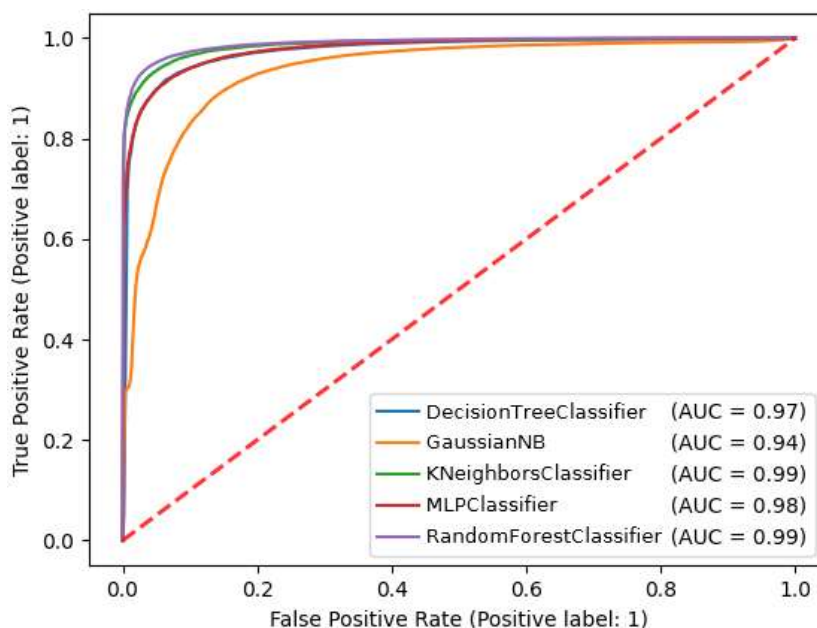
Fonte: Produção própria do autor.

Como pode ser observado, grande parte dos modelos treinados obtiveram bons resultados, com a única exceção sendo o Naïve Bayes gaussiano, que atingiu muitos falsos positivos quanto ao

tráfego com presença de ataques, com um total aproximado de 36.76% das amostras sendo classificadas erroneamente.

Além das matrizes confusão, são geradas as curvas ROC para cada modelo treinado com os melhores hiperparâmetros selecionados a partir das buscas randômicas. A Figura 13 ilustra uma média das curvas geradas nas 3 etapas de validação cruzada de cada classificador, além de uma linha vermelha pontilhada indicando um modelo aleatório, com 50% de chance de catalogar corretamente. Pode-se notar que, dentre todos os modelos, especialmente o *Random Forest* e *K-Nearest Neighbors* atingem um alto valor AUC.

Figura 13 – Curvas ROC por estimador



Fonte: Produção própria do autor.

Inicialmente, tais gráficos podem dar uma ideia de que os dados recolhidos podem ser de alto valor para uma detecção de ataques DoS por SQLi, dependendo de qual algoritmo ML é utilizado e treinado para este fim.

5 CONCLUSÕES

Este trabalho teve como objetivo principal estudar a viabilidade do uso de dados de telemetria de computação para a detecção de ataques DoS de aplicação, estes causados por SQL *injection*. Para isso, foram implementadas e comparadas algumas técnicas de *machine learning* para a detecção e identificação destes ataques, optando-se pela utilização de 5 algoritmos: árvores de decisão, *K-Nearest Neighbors*, *Multi Layer Perceptron*, Naïve Bayes gaussiano e *Random Forest*. Este último o maior sucesso na taxa de detecção dos ataques, atingindo uma acurácia de 95,17% e precisão de 96,0%.

Ao longo das análises feitas, pôde-se observar que há potencial para a utilização de algoritmos de aprendizagem de máquina supervisionada com dados de telemetria de computação em nuvem, detectando ataques que afetam a disponibilidade de sistemas hospedados em uma plataforma *OpenStack*. Tal detecção é feita através da coleta de dados de telemetria da nuvem, estas produzidas pelo serviço *Ceilometer* e recolhidas através de uma biblioteca Python que faz a comunicação com a API do Gnocchi, um banco de dados responsável pelo registro destas métricas de telemetria. A avaliação deste sistema se deu por dados reais obtidos em um ambiente de experimentação, semelhando a requisições maliciosas reais de SQL *injection*, causando negação de serviço em uma aplicação *web* vulnerável.

Mediante os resultados apresentados na subseção 4.4, é possível verificar que os métodos propostos alcançaram um ótimo desempenho, funcionando de forma eficiente na detecção dos ataques DoS na camada de aplicação, especificamente quanto a um serviço Web. Embora certamente um resultado de 95,15% seja de grande performance, os modelos obtidos ainda não são comparáveis a resultados mais recentes da literatura, como o estudo feito por Kachavimath et al. em 2020.

Esta diferença pode ser explicada pela natureza dos dados empregados em outros trabalhos, visto que são utilizadas informações vindas de tráfego de rede. Um fator interessante seria a utilização destes outros tipos de dados, como pacotes e fluxo de rede, para auxiliar no treinamento dos classificadores e possivelmente obter um melhor desempenho.

A utilização de outros conjuntos de dados que simulam outros ataques, como um *HTTP flood*, podem ser de grande valia para generalização do modelo treinado e para melhor comparação com outros trabalhos. É possível também que haja uma limitação dos algoritmos utilizados, mas é necessário uma investigação mais profunda em trabalhos futuros para se determinar isso, aplicando outras técnicas de processamento e *machine learning*.

REFERÊNCIAS BIBLIOGRÁFICAS

- ABURADA, K.; ARIKAWA, Y.; USUZAKI, S. Use of access characteristics to distinguish legitimate user traffic from DDoS attack traffic. **Artificial Life and Robotics**, v. 24, p. 318–323, jan. 2019.
- ARBOR NETWORKS, CISCO. **Cisco Annual Internet Report (2018-2023) White Paper**. 2018. Disponível em: <https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf>. Acesso em: 19 ago. 2020.
- BHOSALE, K. S.; NENOVA, M.; ILIEV, G. The Distributed Denial of Service Attacks (DDoS) Prevention Mechanisms on Application Layer. *In*: 2017 13th International Conference on Advanced Technologies, Systems and Services in Telecommunications (TELSIKS), 2017, Nis. **Proceedings** [...]. Nis, p. 136-139, out. 2017. Disponível em: <https://ieeexplore.ieee.org/document/8246247/>. Acesso em: 18 abr. 2021.
- CYBERSECURITY & INFRASTRUCTURE SECURITY AGENCY. **Understanding Denial-of-Service Attacks**. 2009. Disponível em: <https://us-cert.cisa.gov/ncas/tips/ST04-015>. Acesso em: 26 ago. 2020.
- DAYAL, N.; SRIVASTAVA, S. An RBF-PSO based approach for early detection of DDoS Attacks in SDN. *In*: 10th International Conference on Communications Systems & Networks (COSMSNETS), 2018, Bangalore. **Proceedings** [...]. Bangalore, p. 17-24, 2018. Disponível em: <https://ieeexplore.ieee.org/document/8328175/>. Acesso em: 15 set. 2020.
- DHINGRA, A; SACHDEVA, M. DDoS detection and discrimination from flash events: a compendious review. *In*: 2018 First International Conference on Secure Cyber Computing and Communication (ICSCCC), 2018, Jalandhar. **Proceedings** [...]. Jalandhar, p. 518-524, dez. 2018. Disponível em: <https://ieeexplore.ieee.org/document/8703335/>. Acesso em: 02 abr. 2021.
- FENG, Y.; LI, J.; NGUYEN, T. Application-Layer DDoS Defense with Reinforcement Learning. *In*: 2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS), Hang Zhou, 2020. **Proceedings** [...]. Hang Zhou, p. 1-10, 2020. Disponível em: <https://ieeexplore.ieee.org/document/9213026/>. Acesso em: 15 mar. 2021.
- GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. Cambridge: MIT Press, 2016. Disponível em: <https://www.deeplearningbook.org/>. Acesso em: 12 ago. 2020.
- HUNTER, J. D. Matplotlib: A 2D graphics environment. **Computing in Science & Engineering**, v. 9, n. 3, p. 90-95, 2007.
- HUSSAIN, K.; HUSSAIN, S. J.; HUMAYUN, M.; JHANJHI, NZ. SYN Flood Attack Detection based on Bayes Estimator (SFADBE) For MANET. *In*: 2019 International Conference on Computer and Information Sciences (ICCIS), 2019, Sakaka. **Proceedings** [...]. Sakaka, p. 1-4, 2019. Disponível em: <https://ieeexplore.ieee.org/document/8716416/>. Acesso em: 15 mar. 2021.

KACHAVIMATH, A. V.; NAZARE, S. V.; AKKI, S. S. Distributed Denial of Service Attack Detection using Naïve Bayes and K-Nearest Neighbors for Network Forensics. *In: International Conference on Innovative Mechanisms for Industry Applications, 2020, Bangalore. Proceedings [...]. Bangalore, p. 711-717, 2020. Disponível em: <https://ieeexplore.ieee.org/document/9074929/>. Acesso em: 19 mar. 2021.*

MA, L.; ZHAO, D.; GAO, Y; ZHAO, C. Research on SQL injection Attack and Prevention Technology Based on Web. *In: 2019 International Conference on Computer Network, Electronics and Automation (ICCNEA), 2019, Xi'an. Proceedings [...]. Xi'an, p. 176-179, 2019. Disponível em: <https://ieeexplore.ieee.org/document/8912016/>. Acesso em: 11 ago. 2020.*

MCKINNEY, W. Data structures for statistical computing in Python. *In: PYTHON IN SCIENCE CONFERENCE, 9., 2010, Austin. Proceedings [...]. Austin, p. 51-56, 2010. Disponível em: <https://conference.scipy.org/proceedings/scipy2010/pdfs/mckinney.pdf>. Acesso em: 04 abr. 2021.*

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. **The NIST Definition of Cloud Computing**. Gaithersburg: National Institute of Standards and Technology, 2011. Disponível em: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf>. Acesso em: 18 ago. 2020.

NEWMAN, L. H. **Github Survived the Biggest DDoS Attack Ever Recorded**. San Francisco: Wired, 2018. Disponível em: <https://www.wired.com/story/github-ddos-memcached/>. Acesso em: 11 set. 2020.

OPEN WEB APPLICATION SECURITY PROJECT. **OWASP Top Ten**. 2017. Disponível em: <https://owasp.org/www-project-top-ten/>. Acesso em: 17 set. 2020.

OPENSTACK FOUNDATION. **OpenStack Open-Source Cloud Computing Software**. Disponível em: <https://www.openstack.org/software/>. Acesso em: 03 fev. 2021.

OSANAIYE, O.; RAYMOND, K.; DLODLO, M. Distributed denial of service (DDoS) resilience in cloud: Review and conceptual cloud DDoS mitigation framework. **Journal of Network and Computer Applications**, v. 67, p. 147-165, 2016.

PEDREGOSA, F. Scikit-learn: Machine Learning in Python. **Journal of Machine Learning Research**, v. 12, p. 2825-2830, 2011.

PEPPLE, K. **Understanding Nova: Deploying OpenStack**. 1. ed. Newton: O'Reilly Media, 2011. Disponível em: <https://www.oreilly.com/library/view/deploying-openstack/9781449311223/ch04.html>. Acesso em: 07 mar. 2021.

PRASEED, A.; THILAGAM, P. S. DDoS Attacks at the Application Layer: Challenges and Research Perspectives for Safeguarding Web Applications. **IEEE Communications Surveys & Tutorials**, v. 21, n. 1, p. 661-685, 2019.

PYTHON SOFTWARE FOUNDATION. **Python 3.8.2 documentation**. 2020. Disponível em: <https://docs.python.org/release/3.8.2/>. Acesso em: 15 mar. 2020.

ROEMPLUK, T.; SURINTA, O. A Machine Learning Approach for Detecting Distributed Denial of Service Attacks. *In*: 2019 Joint International Conference on Digital Arts, Media and Technology with ECTI Northern Section Conference on Electrical, Electronics, Computer and Telecommunications Engineering (ECTI DAMT-NCON), 2019, Nan. **Proceedings** [...]. Nan, p. 146-149, 2019. Disponível em: <https://ieeexplore.ieee.org/document/8692243/>. Acesso em: 27 abr. 2021.

ROHR, A. **Sites da olimpíada do Rio sofreram ataque de 500 Gbps, diz empresa**. Rio de Janeiro: Grupo Globo, 2016. Disponível em: <http://g1.globo.com/tecnologia/blog/seguranca-digital/post/sites-da-olimpiada-do-rio-sofreram-ataque-de-500-gbps-diz-empresa.html>. Acesso em: 11 set. 2020.

SHOREY, T.; SUBBAIAH, D.; GOYAL, A.; SAKXENA, A.; MISHRA, A. K. Performance Comparison and Analysis of Slowloris, GoldenEye and Xerxes DDoS Attack Tools. *In*: 2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI), 2018, Bangalore. Proceedings [...]. Bangalore, pp. 318-322, 2018. Disponível em: <https://ieeexplore.ieee.org/document/8554590>. Acesso em: 20 mai. 2021.

SOFTLINE GROUP. **IaaS, PaaS e SaaS: entenda os modelos de nuvem e suas finalidades**. 2017. Disponível em: <https://brasil.softlinegroup.com/sobre-a-empresa/blog/iaas-paas-saas-nuvem>. Acesso em: 25 jul. 2020.

TANENBAUM, A. S; WETHERAL, D. **Redes de Computadores**. 5. ed. São Paulo: Pearson Prentice Hall, 2011.

VECCHIA, E. D. **Perícia Digital: Da Investigação à Análise Forense**. 2. ed. Campinas: Millennium Editora, 2020.

VERAS, M. **Computação em Nuvem: Nova Arquitetura de TI**. 1. ed. Rio de Janeiro: Brasport Livros e Multimídia, 2015. Disponível em: <https://plataforma.bvirtual.com.br/Acervo/Publicacao/160695>. Acesso em: 15 abr. 2021.

YAN, Q.; YU, F. Distributed Denial of Service Attacks in Software-Defined Networking with Cloud Computing. **IEEE Communications Magazine**, v. 53, n. 4, p. 52-59, 2015.

YAN, Q.; YU, F. R.; GONG, Q.; LI, J. Software-Defined Networking (SDN) and Distributed Denial of Service (DDoS) Attacks in Cloud Computing Environments: A Survey, Some Research Issues and Challenges. **IEEE Communications Surveys & Tutorials**, v. 18, n. 1, p. 602-622, 2016.

ZHIJUN, W.; WENJING, L.; LIANG, L.; MENG, Y. Low-Rate DoS Attacks, Detection, Defense, and Challenges: A Survey. **IEEE Access**, v. 8, p. 43920-43943, 2020.

APÊNDICE A – CÓDIGO PARA TREINAMENTO E MÉTRICAS

```

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from sklearn.metrics import accuracy_score, precision_score, f1_score,
recall_score, plot_roc_curve, plot_confusion_matrix
from sklearn.model_selection import cross_val_score, train_test_split

from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier

from sklearn.model_selection import RandomizedSearchCV

# carregando arquivo .csv para um dataframe do Pandas
df = pd.read_csv('/home/kmilach/tcc/dadosNormalizados.csv') #retorna um
dataframe Pandas

# criando series X e Y
X = df.loc[:, df.columns[0:8]] #dataframe com as características
Y = df.label #dataframe com os labels (0 - sem ataque, 1 - com ataque)

# dividindo a serie para treinamento e para testes
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.3,
random_state=13) #dataframes para treino e testes

classifiers = [
    DecisionTreeClassifier(),
    GaussianNB(),
    KNeighborsClassifier(),
    MLPClassifier(),
    RandomForestClassifier()
]

hyperp_grid = [
    #DecisionTree
    {
        'criterion':['gini',"entropy"],
        'splitter':['best',"random"],
        'max_depth':[None]+[int(x) for x in np.linspace(10, 120,
num = 12)],
        'min_samples_split':[2, 5, 10],
        'min_samples_leaf':[1, 2, 4]
    },
    #GaussianNB
    {
        'var_smoothing':[1e-20,1e-19,1e-18,1e-17,1e-16,1e-15,1e-
14,1e-13,1e-12,1e-11,1e-10,1e-9,1e-8,1e-7,1e-6,1e-5,1e-4,1e-3,1e-2,1e-1,1]
    },
    #KNN
    {
        'n_neighbors':[int(x) for x in np.linspace(5, 40, num=8)],
        'weights':['uniform',"distance"],
        'algorithm':['ball_tree", "kd_tree"],

```

```

        'leaf_size':[int(x) for x in np.linspace(30, 60, num=7)]
    },
    #MLP
    {
        'hidden_layer_sizes':[(int(x),) for x in np.linspace(10,
30, num=5)],
        'activation':['logistic',"tanh","relu"],
        'learning_rate':['constant',"invscaling","adaptive"],
        'max_iter':[int(x) for x in np.linspace(200, 600, num=9)],
        'learning_rate_init':[x for x in np.linspace(1e-3, 3e-3,
num=3)]
    },
    #RandomForest
    {
        'n_estimators':[int(x) for x in np.linspace(80, 160,
num=9)],
        'criterion':['gini',"entropy"],
        'max_depth':[None]+[int(x) for x in np.linspace(10, 120,
num = 12)],
        'min_samples_split':[2, 5, 10],
        'min_samples_leaf':[1, 2, 4]
    }
]

count = 0
best_classifiers = []

for clf in classifiers:
    print("-----")
    print("Procurando hiperparâmetros para "+ str(type(clf)).split(".")[-
1][:-2])
    print("-----")
    clf_random = RandomizedSearchCV(estimator = clf, param_distributions =
hyperp_grid[count], n_iter = 200, cv = 3, verbose=2, random_state=13,
n_jobs = -1)
    clf_random.fit(x_train, y_train)
    y_pred = clf_random.predict(x_test)
    print("-----")
    print(str(type(clf)).split(".")[-1][:-2])
    print("Acurácia: %.2f%%" % (100*accuracy_score(y_test, y_pred)))
    print("Precisão: %.2f%%" % (100*precision_score(y_test, y_pred)))
    print("Recall: %.2f%%" % (100*recall_score(y_test, y_pred)))
    print("F1 score: %.2f%%" % (100*f1_score(y_test, y_pred)))
    print(clf_random.best_params_)
    plot_confusion_matrix(clf_random, x_test, y_test, cmap=plt.cm.Blues)
    plt.title(str(type(clf)).split(".")[-1][:-2])
    plt.savefig('confusion_matrix_' + str(count+1) + '.png')
    best_classifiers.insert(count, clf_random)
    count += 1

fig, ax = plt.subplots()
for clf in best_classifiers:
    plot_roc_curve(clf, x_test, y_test, name=str(type(clf)).split(".")[-
1][:-2], ax=ax)

ax.plot([0, 1], [0, 1], linestyle='--', lw=2, color='r', label="Sorte",
alpha=.8)
plt.savefig('all_roc_curves.png')

```



```

granularity=granularity)
    print(meters)

    if len(meters) == 0:
        return -1
    sum = 0
    for item in meters:
        sum += item[2]

    return sum/len(meters)

def
get_metric_network_incoming(self, resource_id, start, stop, granularity):
    meters =
self.gnocchi_client.metric.get_measures('network.incoming.bytes.rate',
                                        start=start,
                                        stop=stop,

resource_id=resource_id,

granularity=granularity)
    if len(meters) == 0:
        return -1
    sum = 0
    for item in meters:
        sum += item[2]

    return sum/len(meters)

def
get_metric_network_outgoing(self, resource_id, start, stop, granularity):
    meters =
self.gnocchi_client.metric.get_measures('network.outgoing.bytes.rate',
                                        start=start,
                                        stop=stop,

resource_id=resource_id,

granularity=granularity)
    if len(meters) == 0:
        return -1
    sum = 0
    for item in meters:
        sum += item[2]

    return sum/len(meters)

def
get_metric_network_incoming_packets(self, resource_id, start, stop, granularity
):
    meters =
self.gnocchi_client.metric.get_measures('network.incoming.packets.rate',
                                        start=start,
                                        stop=stop,

resource_id=resource_id,

granularity=granularity)
    if len(meters) == 0:

```

```

        return -1
    sum = 0
    for item in meters:
        sum += item[2]

    return sum/len(meters)

def
get_metric_network_outgoing_packets(self, resource_id, start, stop, granularity
):
    meters =
self.gnocchi_client.metric.get_measures('network.outgoing.packets.rate',
                                        start=start,
                                        stop=stop,

resource_id=resource_id,

granularity=granularity)
    if len(meters) == 0:
        return -1
    sum = 0
    for item in meters:
        sum += item[2]

    return sum/len(meters)

def get_metric_disk_read(self, resource_id, start, stop, granularity):
    meters =
self.gnocchi_client.metric.get_measures('disk.device.read.requests.rate',
                                        start=start,
                                        stop=stop,

resource_id=resource_id,

granularity=granularity)
    if len(meters) == 0:
        return -1
    sum = 0
    for item in meters:
        sum += item[2]

    return sum/len(meters)

def get_metric_disk_write(self, resource_id, start, stop, granularity):
    meters =
self.gnocchi_client.metric.get_measures('disk.device.write.requests.rate',
    #meters =
self.gnocchi_client.metric.get_measures('disk.write.requests.rate',
                                        start=start,
                                        stop=stop,

resource_id=resource_id,

granularity=granularity)
    if len(meters) == 0:
        return -1
    sum = 0
    for item in meters:
        sum += item[2]

```

```

        return sum/len(meters)

    def get_metric_swap_in(self, resource_id, start, stop, granularity):
        meters = self.gnocchi_client.metric.get_measures('memory.swap.in',
                                                       start=start,
                                                       stop=stop,

resource_id=resource_id,
granularity=granularity)

        if len(meters) == 0:
            return -1
        sum = 0.0
        for item in meters:
            sum += item[2]

        return sum/len(meters)

    def get_metric_swap_out(self, resource_id, start, stop, granularity):
        meters = self.gnocchi_client.metric.get_measures('memory.swap.out',
                                                       start=start,
                                                       stop=stop,

resource_id=resource_id,
granularity=granularity)

        if len(meters) == 0:
            return -1
        sum = 0.0
        for item in meters:
            sum += item[2]

        return sum/len(meters)

    def get_list_meters(self, resource_id):
        list_meters =
self.gnocchi_client.resource.get(resource_type='generic', resource_id=resource_id)
        return list_meters['metrics']

    def get_resource_network(self, resource_inst_id):
        list_meters =
self.gnocchi_client.resource.list(resource_type='instance_network_interface')
        #print resource_inst_id
        for it in list_meters:
            #print resource_inst_id[0:len(resource_inst_id)+3]
            #if it['original_resource_id'][0:len(resource_inst_id)] ==
resource_inst_id:
                if str(it['original_resource_id']).find(str(resource_inst_id))
!= -1:
                    return it['id']
                #print 'foi'

    def get_resource_disk(self, resource_inst_id):
        list_meters =
self.gnocchi_client.resource.list(resource_type='instance_disk')
        #print resource_inst_id

```

```

    for it in list_meters:
        #print resource_inst_id[0:len(resource_inst_id)+3]
        #if it['original_resource_id'][0:len(resource_inst_id)] ==
resource_inst_id:
        if str(it['original_resource_id']).find(str(resource_inst_id))
!= -1:
            return it['id']
            #print 'foi'
'''List Resources '''
def get_list_resource_type(self):
    list_resource = self.gnocchi_client.resource_type.list()
    #print resource_inst_id
    for it in list_resource:
        #print resource_inst_id[0:len(resource_inst_id)+3]
        if(it['name'] == 'instance_network_interface'):
            print(it)

'''Create Resource type
input: name_resource_type '''
def create_resource_type(self,name):
    resource_type = {u'attributes': {u'instance_id': {u'required':
True, u'type': u'uuid'},
        u'name': {u'min_length': 0, u'max_length': 255, u'type':
u'string', u'required': True}},
        u'name': name}
    print(resource_type)
    self.gnocchi_client.resource_type.create(resource_type)

def find_resource_type(self,resource_type):
    #resource_type = {u'attributes': {u'instance_id': {u'required':
True, u'type': u'uuid'},
        # u'name': {u'min_length': 0, u'max_length': 255, u'type':
u'string', u'required': True}},
        # u'name': name}
    #print(resource_type)
    list_resource = self.gnocchi_client.resource_type.list()
    #print resource_inst_id
    for it in list_resource:
        #print resource_inst_id[0:len(resource_inst_id)+3]
        if(it['name'] == resource_type):
            return True
    return False

def create_resource(self,resource_type,resource):
    x = uuid.uuid1()
    resource = {#u'created_by_user_id':
u'38a0b60e8bdc428eb5b2145cfab70c93',
        #u'metrics': {},
        #u'started_at': u'2018-09-18T15:49:58.288111+00:00',
        #u'revision_start': u'2018-09-18T15:49:58.288135+00:00',
        #u'revision_end': None,
        #u'creator':
u'38a0b60e8bdc428eb5b2145cfab70c93:bf39e317e8774bb895f7c571bfd147ba',
        #u'created_by_project_id': u'bf39e317e8774bb895f7c571bfd147ba',
        u'id': x.hex,
        u'instance_id': u'f614e58a-62b9-4441-86c6-d0b34e1126a1',
        #u'original_resource_id': u'instance-00000056-59511a9e-1b60-4d34-
b8d5-0ef67fbfee1e-tap15478fc9-58',
        #u'user_id': u'f4f081892fa845c3a2afae8ef5c17b46',
        u'project_id': u'9b5c44449bbb41f398126f0bb1dfb45d',

```

```

        #u'type': u'wireless',
        u'name': resource}
        #print(resource)
        resource =
self.gnocchi_client.resource.create(resource_type,resource)
        print(resource[u'instance_id'])

    def find_resource(self,resource_type,resource):
        #resource_type = {u'attributes': {u'instance_id': {u'required':
True, u'type': u'uuid'},
        # u'name': {u'min_length': 0, u'max_length': 255, u'type':
u'string', u'required': True}},
        # u'name': name}
        #print(resource_type)
        list_resource =
self.gnocchi_client.resource.list(resource_type=resource_type)
        #print(list_resource)
        for it in list_resource:
            #print resource_inst_id[0:len(resource_inst_id)+3]
            #print(it[u'name'])
            if(it[u'name'] == resource):
                #print(it[u'instance_id'])
                return (it[u'id'])
        return False

    def create_metrics(self,name,resource_id):
        resource = {#u'created_by_user_id':
u'38a0b60e8bdc428eb5b2145cfab70c93',
        #u'metrics': {},
        #u'started_at': u'2018-09-18T15:49:58.288111+00:00',
        #u'revision_start': u'2018-09-18T15:49:58.288135+00:00',
        #u'revision_end': None,
        #u'creator':
u'38a0b60e8bdc428eb5b2145cfab70c93:bf39e317e8774bb895f7c571bfd147ba',
        #u'created_by_project_id': u'bf39e317e8774bb895f7c571bfd147ba',
        u'id': u'dde47cc6-72a5-55ca-9d12-13f3623d69b21',
        u'instance_id': u'f614e58a-62b9-4441-86c6-d0b34e1126a1',
        #u'original_resource_id': u'instance-00000056-59511a9e-1b60-4d34-
b8d5-0ef67fbfee1e-tap15478fc9-58',
        #u'user_id': u'f4f081892fa845c3a2afae8ef5c17b46',
        u'project_id': u'9b5c44449bbb41f398126f0bb1dfb45d',
        #u'type': u'wireless',
        u'name': u'current_Speed_linear'}
        #print(resource)
        self.gnocchi_client.metric.create(name=name,
archive_policy_name='high', resource_id=resource_id, unit=None)
        #print(resource[-1])
        #

    def add_metrics(self,name,resource_id,d):
        resource = {#u'created_by_user_id':
u'38a0b60e8bdc428eb5b2145cfab70c93',
        #u'metrics': {},
        #u'started_at': u'2018-09-18T15:49:58.288111+00:00',
        #u'revision_start': u'2018-09-18T15:49:58.288135+00:00',
        #u'revision_end': None,
        #u'creator':
u'38a0b60e8bdc428eb5b2145cfab70c93:bf39e317e8774bb895f7c571bfd147ba',
        #u'created_by_project_id': u'bf39e317e8774bb895f7c571bfd147ba',

```

```

        u'id': u'dde47cc6-72a5-55ca-9d12-13f3623d69b21',
        u'instance_id': u'59511a9e-1b60-4d34-b8d5-0ef67fbfee1e',
        #u'original_resource_id': u'instance-00000056-59511a9e-1b60-4d34-
b8d5-0ef67fbfee1e-tap15478fc9-58',
        #u'user_id': u'f4f081892fa845c3a2afae8ef5c17b46',
        u'project_id': u'2953ca0b12a749329d805d237f118f7a',
        #u'type': u'wireless',
        u'name': name}
        self.gnocchi_client.metric.add_measures(metric=name, measures=d,
resource_id=resource_id)

def save_meter_in_json(name_meter, list_meter, directory, fps, name_VM):

    dic_memory_usage = {}
    list = []
    if not os.path.exists(directory + fps + '/' + str(name_VM) + '/'):
        os.makedirs(directory + fps + '/' + str(name_VM) + '/')

    # Storage informations in files json, for example vm1_memory_usage.json
    for it_memory in list_meter:
        # Transform datetime for timestamp
        dic_memory_usage = {}
        dic_memory_usage['timestamp'] =
str(datetime.datetime.fromtimestamp(float(it_memory[0].strftime('%s'))))
        dic_memory_usage['value'] = it_memory[2]
        dic_memory_usage['granularity'] = it_memory[1]
        list.append(dic_memory_usage)
    dir = str(directory + fps + '/' + str(name_VM) + '/' + name_meter + '.json')

    with open(dir, 'wb') as outfile:
        json.dump(list, outfile, indent=4, separators=(',', ': '))

def main():

    auth_plugin =
auth.GnocchiBasicPlugin(user="70b32de45953459f85b6cbff60aedecf", endpoint="h
ttp://10.64.0.15:8041")
    gnocchi = client.Client(session_options={'auth': auth_plugin})

    cloud = Gnocchi('openstack')

    with open('metrics.txt', 'a') as f:

f.write('cpu,memory,memory_swap_in,memory_swap_out,disk_request_read,disk_r
equest_write,interface_bytes_in,interface_bytes_out,interface_packets_in,in
terface_packets_out,timestamp,label,label2\n')

    try:
        while True:
            stop = datetime.datetime.utcnow()
            start = (stop - datetime.timedelta(seconds=int(sys.argv[2])))

            #infos requisitadas de telemetria
            cpu = cloud.get_metric_cpu_utilization(sys.argv[1],
start.isoformat(), stop.isoformat(), int(sys.argv[2]))
            memory = cloud.get_metric_memory_usage(sys.argv[1],
start.isoformat(), stop.isoformat(), int(sys.argv[2]))
            try:
                memory_swap_in = cloud.get_metric_swap_in(sys.argv[1],
start.isoformat(), stop.isoformat(), int(sys.argv[2]))

```

```

        memory_swap_out = cloud.get_metric_swap_out(sys.argv[1],
start.isoformat(), stop.isoformat(), int(sys.argv[2]))
    except:
        memory_swap_in = 0
        memory_swap_out = 0
        id_disk = cloud.get_resource_disk(sys.argv[1])
        disk_request_read = cloud.get_metric_disk_read(id_disk,
start.isoformat(), stop.isoformat(), int(sys.argv[2]))
        disk_request_write = cloud.get_metric_disk_write(id_disk,
start.isoformat(), stop.isoformat(), int(sys.argv[2]))
        id_interface = cloud.get_resource_network(sys.argv[1])
        interface_bytes_in =
cloud.get_metric_network_incoming(id_interface, start.isoformat(),
stop.isoformat(), int(sys.argv[2]))
        interface_bytes_out =
cloud.get_metric_network_outgoing(id_interface, start.isoformat(),
stop.isoformat(), int(sys.argv[2]))
        interface_packets_in =
cloud.get_metric_network_incoming_packets(id_interface, start.isoformat(),
stop.isoformat(), int(sys.argv[2]))
        interface_packets_out =
cloud.get_metric_network_outgoing_packets(id_interface, start.isoformat(),
stop.isoformat(), int(sys.argv[2]))

    cpu = round(cpu,2)
    memory = round(memory,2)
    memory_swap_in = round(memory_swap_in,2)
    memory_swap_out = round(memory_swap_out,2)
    disk_request_read = round(disk_request_read,2)
    disk_request_write = round(disk_request_write,2)
    interface_bytes_in = round(interface_bytes_in,2)
    interface_bytes_out = round(interface_bytes_out,2)
    interface_packets_in = round(interface_packets_in,2)
    interface_packets_out = round(interface_packets_out,2)

    print("CPU: %f" % cpu)
    print("Memory: %f" % memory)
    print("Memory Swap In: %f" % memory_swap_in)
    print("Memory Swap Out: %f" % memory_swap_out)
    print("Disk request read: %f" % disk_request_read)
    print("Disk request write: %f" % disk_request_write)
    print("Bytes in: %f" % interface_bytes_in)
    print("Bytes out: %f" % interface_bytes_out)
    print("Packets in: %f" % interface_packets_in)
    print("Packets out: %f" % interface_packets_out)
    print("Timestamp: %s" % start.isoformat())
    print("-----")

    with open('metrics.txt','a') as f:

f.write(str(cpu)+' '+str(memory)+' '+str(memory_swap_in)+' '+str(memory_swa
p_out)+' '+str(disk_request_read)+' '+str(disk_request_write)+' '+str(inter
face_bytes_in)+' '+str(interface_bytes_out)+' '+str(interface_packets_in)+'
'+str(interface_packets_out)+' '+str(start.isoformat())+' '+str(sys.argv[3
])+'\n')

        time.sleep(int(sys.argv[2]))
except KeyboardInterrupt:
    exit(1)

```

```
if __name__ == "__main__":
    if len(sys.argv) < 4 or sys.argv[1] == "-h":
        print(" - Usage: ./agent-gnocchi.py [OpenStack-VM-ID] [time-of-
capture-in-seconds] [label]")
        print(" - Result writer in metrics.txt. Metrics sequence in
archive:
CPU,Memory,MemorySwapIn,MemorySwapOut,HDRequestRead,HDRequestWrite,Interfac
eBytesIn,InterfaceBytesOut,InterfacePacketsIn,InterfacePacketsOut,TimeStamp
,label")
        exit(1)
    main()
```


APÊNDICE C – CÓDIGO PARA GERAÇÃO DE DADOS

```

from paramiko import SSHClient
import paramiko
import time
import os

# Lembrar as seguintes regras para labels:
# "0" => sem ataque
# "1" => com ataque (SQL Injection)

def monitoramento(label):
    os.system("pkill -9 -f agent-gnocchi.py")
    os.system("pkill -9 -f agent-gnocchi.py")
    os.system("pkill -9 -f agent-gnocchi.py")
    os.system("pkill -9 -f agent-gnocchi.py")
    print("Monitorando: %s" % label)
    os.system("nohup ./agent-gnocchi.py f614e58a-62b9-4441-86c6-
d0b34e1126a1 1 %s &" % label)

def ataqueFrac0():
    ssh_attack1.exec_cmd("nohup ./ataqueFrac0 </dev/null &>/dev/null &")

def ataqueMedio():
    ssh_attack1.exec_cmd("nohup ./ataqueMedio </dev/null &>/dev/null &")

def ataqueForte():
    ssh_attack1.exec_cmd("nohup ./ataqueForte </dev/null &>/dev/null &")

def clienteFrac0():
    ssh_client1.exec_cmd("nohup ./clienteFrac0 </dev/null &>/dev/null &")

def clienteMedio():
    ssh_client1.exec_cmd("nohup ./clienteMedio </dev/null &>/dev/null &")

def clienteForte():
    ssh_client1.exec_cmd("nohup ./clienteForte </dev/null &>/dev/null &")

def restart():
    ssh_server.exec_cmd("reboot")
    ssh_client1.exec_cmd("reboot")
    ssh_attack1.exec_cmd("reboot")

def pararTudo():
    ssh_attack1.exec_cmd("pkill -9 java")
    ssh_client1.exec_cmd("pkill -9 java")
    ssh_server.exec_cmd("./pararSQLi")
    ssh_server.exec_cmd("pkill -9 tcpdump")

def pararAtaque():
    ssh_attack1.exec_cmd("pkill -9 java")
    ssh_server.exec_cmd("./pararSQLi")

def pararCliente():
    ssh_client1.exec_cmd("pkill -9 java")

class SSH:
    def __init__(self, ip):
        self.ssh = SSHClient()

```

```

self.ssh.load_system_host_keys()
self.ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
self.ssh.connect(hostname=ip,username='root',password='toor')

def exec_cmd(self,cmd):
    stdin,stdout,stderr = self.ssh.exec_command(cmd)
    print(stdout.read())
    #if stderr.channel.recv_exit_status() != 0:
    #    print(stderr.read())
    #else:
    #    print(stdout.read())

if __name__ == '__main__':
    ssh_client1 = SSH('192.0.2.12') #clientes-1 (cliente)
    print('Aberta sessao SSH com clientes-1')
    ssh_attack1 = SSH('192.0.2.39') #atacante-2 (atacante)
    print('Aberta sessao SSH com atacante-2')
    ssh_server = SSH('192.0.2.8') #dvwa-server (vitima)
    print('Aberta sessao SSH com dvwa-server')

    #Iniciando contagem
    tempo_inicial = time.time()
    pararTudo()
    time.sleep(10)

# ----- INICIO DO MONITORAMENTO -----

#Cliente fraco
monitoramento('0')
time.sleep(10)
clienteFraco()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Cliente medio + ataque fraco
monitoramento('1')
time.sleep(10)
clienteMedio()
ataqueFraco()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Cliente forte
monitoramento('0')
time.sleep(10)
clienteForte()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Cliente forte + ataque forte

```

```
monitoramento('1')
time.sleep(10)
clienteForte()
ataqueForte()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Cliente medio
monitoramento('0')
time.sleep(10)
clienteMedio()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#ataque medio
monitoramento('1')
time.sleep(10)
ataqueMedio()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Cliente forte
monitoramento('0')
time.sleep(10)
clienteForte()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Ataque forte
monitoramento('1')
time.sleep(10)
ataqueForte()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Cliente fraco
monitoramento('0')
time.sleep(10)
clienteFraco()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()
```

```
#Cliente fraco + ataque fraco
monitoreamento('1')
time.sleep(10)
clienteFraco()
ataqueFraco()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Cliente forte
monitoreamento('0')
time.sleep(10)
clienteForte()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Cliente forte + ataque fraco
monitoreamento('1')
time.sleep(10)
clienteForte()
ataqueFraco()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Cliente forte
monitoreamento('0')
time.sleep(10)
clienteForte()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Cliente forte + ataque medio
monitoreamento('1')
time.sleep(10)
clienteForte()
ataqueMedio()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Cliente medio
monitoreamento('0')
time.sleep(10)
clienteMedio()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
```

```

pararCliente()
pararAtaque()

#Cliente fraco + ataque medio
monitoreamento('1')
time.sleep(10)
clienteFrac()
ataqueMedio()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Cliente fraco
monitoreamento('0')
time.sleep(10)
clienteFrac()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Ataque Fraco
monitoreamento('1')
time.sleep(10)
ataqueFrac()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Cliente fraco
monitoreamento('0')
time.sleep(10)
clienteFrac()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Cliente fraco + ataque forte
monitoreamento('1')
time.sleep(10)
clienteFrac()
ataqueForte()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Cliente medio + ataque forte
monitoreamento('1')
time.sleep(10)
clienteMedio()
ataqueForte()

```

```

#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Cliente medio
monitoramento('0')
time.sleep(10)
clienteMedio()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Cliente medio + ataque medio
monitoramento('1')
time.sleep(10)
clienteMedio()
ataqueForte()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

#Cliente medio
monitoramento('0')
time.sleep(10)
clienteMedio()
#Sleep
print("Dormindo 60 minutos")
time.sleep(3600)
pararCliente()
pararAtaque()

# ----- FIM DO MONITORAMENTO -----

#Parar monitoramento
os.system("pkill -9 -f agent-gnocchi.py")

#ssh_server.exec_cmd("pkill -9 -f tcpdump")

tempo_final = time.time()

print('A execucao do algoritmo levou: %d segundos' %(tempo_final-
tempo_inicial))

```