DISSERTAÇÃO DE MESTRADO PROFISSIONAL

Proposta de um framework para detecção de intrusão em clusters de orquestração de contêineres utilizando machine learning para identificação de anomalias em system calls

Sávio Levy Rocha

Brasília, Dezembro de 2022

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

A FRAMEWORK PROPOSAL FOR INTRUSION DETECTION IN CONTAINER ORCHESTRATION CLUSTERS USING MACHINE LEARNING FOR ANOMALY IDENTIFICATION IN SYSTEM CALLS

PROPOSTA DE UM FRAMEWORK PARA DETECÇÃO DE INTRUSÃO EM CLUSTERS DE ORQUESTRAÇÃO DE CONTÊINERES UTILIZANDO MACHINE LEARNING PARA IDENTIFICAÇÃO DE ANOMALIAS EM SYSTEM CALLS

SÁVIO LEVY ROCHA

ORIENTADOR: GEORGES DANIEL AMVAME NZE, Ph.D
COORIENTADOR: FÁBIO LÚCIO LOPES DE MENDONÇA, Ph.D

DISSERTAÇÃO DE MESTRADO PROFISSIONAL EM ENGENHARIA ELÉTRICA

PUBLICAÇÃO: PPEE.MP.031
BRASÍLIA/DF, DEZEMBRO - 2022

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

DISSERTAÇÃO DE MESTRADO PROFISSIONAL

Proposta de um framework para detecção de intrusão em clusters de orquestração de contêineres utilizando machine learning para identificação de anomalias em system calls

Sávio Levy Rocha

Dissertação de Mestrado Profissional submetida ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Mestre em Engenharia Elétrica

Banca Examinadora

Prof. Georges Daniel Amvame Nze, Ph.D, FT/UnB _____
Orientador

Prof. Rafael Rabelo Nunes, Ph.D, FT/UnB _____
Examinador Interno

Luiz Fernando Sirotheau Serique Junior, Ph.D, TJDFT _____
Examinador Externo

FICHA CATALOGRÁFICA

ROCHA, SÁVIO LEVY
Proposta de um framework para detecção de intrusão em clusters de orquestração de contêineres utilizando machine learning para identificação de anomalias em system calls [Distrito Federal] 2022.
xvi, 102 p., 210 x 297 mm (ENE/FT/UnB, Mestre, Engenharia Elétrica, 2022).
Dissertação de Mestrado Profissional - Universidade de Brasília, Faculdade de Tecnologia.
Departamento de Engenharia Elétrica

1. Detecção de Intrusão	2. Contêiner
3. SOC	4. Segurança Cibernética
I. ENE/FT/UnB	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

ROCHA, S.L. (2022). *Proposta de um framework para detecção de intrusão em clusters de orquestração de contêineres utilizando machine learning para identificação de anomalias em system calls*. Dissertação de Mestrado Profissional, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 102 p.

CESSÃO DE DIREITOS

AUTOR: Sávio Levy Rocha
TÍTULO: Proposta de um framework para detecção de intrusão em clusters de orquestração de contêineres utilizando machine learning para identificação de anomalias em system calls.
GRAU: Mestre em Engenharia Elétrica ANO: 2022

Sávio Levy Rocha
Depto. de Engenharia Elétrica (ENE) - FT
Universidade de Brasília (UnB)
Campus Darcy Ribeiro
CEP 70919-970 - Brasília - DF - Brasil

DEDICATÓRIA

Dedico este trabalho ao Deus Eterno Todo-Poderoso. Porque dele, e por meio dele, e para ele são todas as coisas. A ele, pois, a glória eternamente. (Rom 11:36)

AGRADECIMENTOS

Agradeço à Deus pela sua fidelidade e infinita bondade para comigo. Até aqui nos ajudou o Senhor. (1Sm 7:12)

Agradeço ao meu orientador Georges Daniel Amvame Nze, e coorientador Fábio Lúcio Lopes de Mendonça, pelo tempo dedicado à mim, ao apoio e direcionamentos valiosos.

Agradeço à minha família e noiva pelo incentivo e compreensão nos momentos de dificuldade.

Agradeço à todo corpo docente do PPEE e colegas durante essa caminhada.

Agradeço ao Tribunal de Justiça do Distrito Federal e Territórios pela oportunidade que me foi concedida e aos colegas de trabalho pela compreensão nos momentos de afastamento.

Agradeço ao suporte da ABIN TED 08/2019.

RESUMO

A computação em nuvem introduziu novas tecnologias que permitiram a construção de um modelo mais ágil de integração e entrega contínua (CI/CD) no pipeline de desenvolvimento de aplicações. Uma dessas tecnologias é a utilização de contêineres em substituição às tradicionais máquinas virtuais. Além dos benefícios trazidos pelo uso dos contêineres, ameaças e riscos de ataques voltados para esses ambientes cresceram em igual proporção à sua adoção. Sistemas de Detecção de Intrusão (IDS) têm sido empregados para garantir a segurança de ambientes de nuvem, contudo, as características inerentes a esses ambientes têm apresentados novos desafios para alcançar bons resultados na detecção de intrusão. Estritamente, no que diz respeito à detecção de intrusão em ambientes de contêineres, poucos estudos foram conduzidos até o momento visando o seu aprimoramento. Neste trabalho, é proposto um *framework* contendo uma arquitetura composta por cinco camadas e suas ferramentas para implementação de um IDS baseado em *Host* (HIDS) voltado para plataformas de orquestração de contêineres através da identificação de anomalias em *system calls*. O *framework* implementado em uma topologia de rede corporativa funcional emulada no *software* GNS3 foi testado com um *dataset* de *system calls* público demonstrando a viabilidade de seu funcionamento. Através do experimento realizado, foi possível validar a integração entre as camadas do *framework* e os resultados de detecção obtidos utilizando um modelo de *machine learning* não supervisionado superaram os do trabalho que originou o *dataset* público utilizado. Os *datasets* são carregados, transformados e extraídos de uma plataforma gratuita e aberta com *front-end* para visualização de alertas de detecção de anomalias que podem ser analisados pela equipe do SOC em um *dashboard* criado para monitoramento do IDS e auxílio na tomada de decisão.

ABSTRACT

Cloud computing has introduced new technologies that have enabled a more agile continuous integration and continuous delivery (CI/CD) model to be built into the application development pipeline. One such technology is the use of containers in replacement to the traditional virtual machines. In addition to the benefits brought by the use of containers, threats and risks of attacks aimed at these environments have grown in equal proportion to their adoption. Intrusion Detection Systems (IDS) have been employed to secure cloud environments, however, the inherent characteristics of these environments have presented new challenges to achieving good intrusion detection results. Strictly, regarding intrusion detection in container environments, few studies have been conducted so far aiming at its improvement. In this work, a framework containing an architecture composed of five layers and its tools is proposed to implement a Host-based IDS (HIDS) aimed at container orchestration platforms through the identification of anomalies in system calls. The framework implemented in a functional corporate network topology emulated in the GNS3 software was tested with a public dataset of system calls demonstrating the viability of its operation. Through the experiment, it was possible to validate the integration between the layers of the framework

and the detection results obtained using an unsupervised machine learning model surpassed those of the work that originated the public dataset used. The datasets are loaded, transformed and extracted from a free and open platform with front-end for visualization of anomaly detection alerts that can be analyzed by the SOC team in a dashboard created for IDS monitoring and decision making support.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

Unidades de medida

Gbps	Giga bit por segundo
ZB	Zeta byte
GHz	Giga Hertz
MHz	Mega Hertz
GB	Giga Byte

Siglas

CI/CD	*Continous Integration and Continuous Delivery* (Integração e Entrega Contínua)
VM	*Virtual Machine* (Máquina Virtual)
TI	Tecnologia da Informação
SO	Sistema Operacional
IDS	*Intrusion Detection System(s)* (Sistema(s) de Detecção de Intrusão)
HIDS	*Host-based Intrusion Detection System(s)* (Sistema(s) de Detecção de Intrusão baseado(s) em Host)
NIDS	*Network Intrusion Detection System(s)* (Sistema(s) de Detecção de Intrusão de Rede
CIDS	*Collaborative Intrusion Detection System(s)* (Sistema(s) de Detecção de Intrusão Colaborativo(s))
SIDS	*Signature-based Intrusion Detection System(s)* (Sistema(s) de Detecção de Intrusão baseado(s) em Assinatura)
AIDS	*Anomaly-based Intrusion Detection System(s)* (Sistema(s) de Detecção de Intrusão baseado(s) em Anomalia)
IPS	*Intrusion Prevention System(s)* (Sistema(s) de Prevenção de Intrusão)
USAF	*United States Air Force* (Força Aérea dos Estados Unidos)
SIEM	*Security Information and Event Management* (Gerenciamento de Eventos e Informações de Segurança)
SOC	*Security Operations Center* (Centro de Operações de Segurança)
Pentest	*Penetration Test* (Teste de Pentração)
BIA	*Business Impact Analysis* (Análise de Impacto ao Negócio)
RA	*Risk Analysis* (Análise de Riscos)
DR	*Detection Rate* (Taxa de Detecção)
FPR	*False Positive Rate* (Taxa de Falsos Positivos)
TNR	*True Negative Rate* (Taxa de Verdadeiros Negativos)
COV	*Coverage* (Cobertura)
FAR	*False Alarm Rate* (Taxa de Alarmes Falsos)
PaaS	*Platform as a Service* (Plataforma como Serviço)
SaaS	*Software as a Service* (*Software* como Serviço)

IaaS	*Infrastructure as a Service* (Infraestrutura como Serviço)
XaaS	*Anything as a Service* (Qualquer coisa como Serviço)
DevOps	*Software Development and IT Operations* (Desenvolvimento de Aplicações e Operações de TI)
IoT	*Internet of Things* Internet das Coisas
ML	*Machine Learning* Aprendizado de máquina
BoSC	*Bag of System Calls* (Bolsas de System Calls)
STIDE	*Sequence Time-Delaying Embedding*
NN	*Neural Network* (Rede Neural)
ANN	*Artificial Neural Network* (Rede Neural Artificial)
DNN	*Deep Neural Network* (Rede Neural Profunda)
AE	*AutoEncoder* (Autoencoder)
DAE	*Deep AutoEncoder* (Autoencoder Profundo)
RNN	*Recurrent Neural Network* (Rede Neural Recorrente)
LSTM	*Long Short-Term Memory* (Memória de Longo e Curto Prazo)
AUC	*Area Under the Curve* (Área abaixo da Curva)
ROC	*Receiver Operating Characteristic* (Característica Operacional do Receptor)
DL	*Deep Learning* (Aprendizado Profundo)
API	*Application Programming Interface* (Interface de Programação de Aplicação)
REST	*Representational State Transfer* (Transferência de Estado Representacional)
HTTP	*HyperText Transfer Protocol* (Protocolo de Transferência de Hypertexto)
JSON	*JavaScript Object Notation* (Notação de Objeto do Javascript)
GPU	*Graphical Processor Unit* (Unidade de Processamento Gráfico)
NLP	*Natural Language Processing* (Processamento de Linguagem Natural)
VLAN	*Virtual Local Area Network* (Rede de Área Local Virtual)
DMZ	*Demilitarized Zone* (Zona Desmilitarizada)
DNS	*Domain Name System* (Sistema de Nomes de Domínio)
NAT	*Network Address Translation* (Tradução de Endereços de Rede)
CSV	*Comma-separated Values* (Valores Separados por Vírgula)
SEI	(Sistema Eletrônico de Informações)

1 INTRODUÇÃO

Tecnologias de computação em nuvem têm permitido a construção de um modelo mais ágil de Integração e Entrega Contínua (CI/CD) no *pipeline* de desenvolvimento de sistemas. Uma dessas tecnologias é a utilização de contêineres em substituição ao tradicional modelo de virtualização implementado através das Máquinas Virtuais (VMs) como ambiente para execução de aplicações. O uso de contêineres e outras ferramentas associadas à metodologia ágil de desenvolvimento têm causado uma mudança de cultura na forma de trabalho das equipes de Tecnologia da Informação (TI) no âmbito corporativo.

Nessa esteira, para atender às necessidades decorrentes desse novo processo de construção, implantação e gerência do ciclo de vida de aplicações a tecnologia de contêineres têm crescido em adoção não apenas em grandes provedores de nuvem, mas também em diversas empresas e órgãos governamentais. Plataformas para gerência e administração também foram introduzidas com a finalidade de orquestração de um volume massivo de contêineres executando em diversos computadores distribuídos em *clusters* com grande poder computacional. Desse modo, essa mudança que abrange desde o ciclo de desenvolvimento de aplicações até a infraestrutura tecnológica que sustenta essa operação tem feito com que as instituições se adéquem para se tornarem verdadeiros provedores de nuvem privada, que conforme [15], provém serviços que possuem características essenciais como: *self-service* sob demanda; amplo acesso por rede; *pool* de recursos; rápida elasticidade e medição de serviço.

Devido a características de funcionamento como o compartilhamento de recursos de um mesmo Sistema Operacional (SO) hospedeiro entre múltiplos contêineres, ataques que exploram falhas no isolamento entre contêineres frequentemente resultam em vazamento de informações. Ademais, em estudo publicado por [16] foi demonstrado que em média mais de 180 vulnerabilidades foram encontradas nas versões das imagens de contêineres oficiais e da comunidade hospedadas no DockerHub, um dos principais repositórios para armazenamento de imagens de contêineres disponível na Internet.

Ao longo do tempo, Sistemas de Detecção de Intrusão (IDS) têm sido empregados visando garantir um nível de segurança adequado aos ambientes computacionais e são uma importante fonte de informação para atuação da equipe de um Centro de Operações de Segurança (SOC). Contudo, com a evolução desses ambientes e o emprego de novas tecnologias, falhas de segurança, ameaças e riscos de ataques voltados para essas tecnologias surgem como novos desafios para a detecção de intrusão.

Diante desse cenário, medidas de precaução ligadas à segurança cibernética são necessárias e exigem que os administradores de redes e equipes de segurança proponham soluções atuais, de maneira a detectar e remediar possíveis violações de segurança em um ambiente computacional heterogêneo e em constante evolução.

1.1 MOTIVAÇÃO

O emprego de plataformas de orquestração de contêineres foi observado como sendo uma das vertentes em órgãos da Administração Pública Federal e a preocupação com o aprimoramento da segurança nesses ambientes foi o principal motivador para a condução deste trabalho. Apesar da forte adoção dessa tecnologia e a consequente necessidade de aprimoramento da detecção de intrusão em ambientes que fazem uso dessa tecnologia, avanços nesse contexto ainda estão em um estágio inicial e permanecem não completamente explorados. Diante do exposto, como principais limitações que carecem de aprimoramento no contexto da detecção de intrusão em ambientes de contêineres as seguintes podem ser elencadas:

- Abordagens para contêineres têm sido negligenciadas e avanços nesse contexto ainda são esparsos [17, 18, 19];
- Alto custo computacional decorrente do grande volume de *system calls* para realizar a detecção de intrusão baseada em *host* de maneira tradicional [20];
- Pouca ênfase tem sido dada para se fazer a implementação de um *Host-based* IDS (HIDS) de tempo real [21];
- Escassez de *datasets* atuais e relevantes envolvendo *system calls* de contêineres [22, 23];
- Taxa de Falso Positivo (FPR) e Precisão de Detecção (DR) de anomalias precisam ser melhoradas em HIDS baseados em *system calls* [20, 24].

1.2 OBJETIVOS

Diante das limitações apresentadas como principais fatores motivacionais de pesquisa, este trabalho tem como objetivo principal a proposta de um *framework* que implementa uma arquitetura para um HIDS distribuído e possibilita a análise de *system calls* para detecção de anomalias no comportamento de sistemas executando em um *cluster* de orquestração de contêineres Kubernetes.

Como objetivos específicos os seguintes foram estabelecidos:

1. Reduzir o *overhead* de processamento causado pela detecção de intrusão nos nós de um *cluster* Kubernetes;
2. Desenvolver uma arquitetura de referência para implementação de um HIDS distribuído voltado para plataformas de orquestração de contêineres com ferramentas gratuitas;
3. Possibilitar a geração de *datasets* com *system calls* de aplicações conteinerizadas;
4. Prover alertas para tomada de decisão do SOC na ocorrência de anomalias em aplicações conteinerizadas detectadas através de técnicas de *machine learning*.

1.3 METODOLOGIA

A metodologia utilizada para condução desse trabalho teve início com um levantamento bibliográfico do estado da arte no que diz respeito aos Sistemas de Detecção de Intrusão e sua aplicação aos ambientes conteinerizados. A partir da literatura relacionada, limitações e lacunas ainda não exploradas pelos estudos foram identificadas e algumas delas definidas como alvos de pesquisa a serem abordados por esse trabalho. As principais implementações de IDS e técnicas de detecção de intrusão do estado da arte foram estudadas e serviram como referência para o projeto de um *framework* contendo uma arquitetura para um HIDS baseado em análise de anomalias em *system calls* com foco em plataformas de orquestração de contêineres, endereçando algumas das limitações existentes no campo de pesquisa.

Para implementação do *framework*, o *software* de emulação de redes GNS3 foi utilizado e a topologia de um ambiente corporativo construída contendo um *cluster* de orquestração de contêineres Kubernetes, múltiplas VLANs, além de um *firewall*, roteadores e *switches*. A arquitetura proposta apresenta um HIDS com componentes distribuídos que viabiliza a elasticidade do sistema conforme a necessidade e foi implementada utilizando ferramentas gratuitas. Um *pipeline* foi então construído compreendendo cinco camadas principais iniciando com a coleta dos dados até a geração de alertas de anomalias e análise de dados na última camada. Cada camada do *pipeline* foi construída com ferramentas configuradas para propósitos específicos e que possibilitam a integração entre as camadas adjacentes.

Testes preliminares foram realizados com um *dataset* de *system calls* público para treinamento e avaliação de algoritmos de *machine learning* na camada de detecção de anomalias. Essa abordagem permitiu que o funcionamento do *framework* e da arquitetura fossem testados com alterações mínimas do *pipeline* na camada de captura de dados, simulando uma aplicação em execução no *cluster* monitorado. Com base nos resultados dos testes de alguns algoritmos de *machine learning* do estado da arte utilizando o *dataset* de teste, o modelo que apresentou o melhor desempenho na detecção de anomalias foi selecionado para emulação do funcionamento do HIDS e implementado no módulo de *machine learning* da camada quatro da arquitetura proposta.

Para validação do funcionamento do IDS construído, modelos de *machine learning* foram treinados com diferentes parâmetros utilizando um *dataset* público de *system calls* com comportamento normal de uma determinada aplicação. Um subconjunto do *dataset* público contendo alguns ataques selecionados foi então utilizado simulando a exploração de falhas de segurança e ataques direcionados à aplicação em execução no *cluster* monitorado. Para cada *dataset* representando um ataque, a eficácia de detecção foi mensurada e avaliada comparando os algoritmos de *machine learning* treinados. Na ocorrência de detecção de anomalias, alertas são gerados e apresentados em um *dashboard* construído para acompanhamento pela equipe do SOC, que pode então prosseguir com a análise do eventual incidente e aplicar as ações cabíveis conforme preconizado pela política de segurança.

1.4 PRINCIPAIS CONTRIBUIÇÕES

O trabalho desenvolvido apresenta um *framework* contendo uma arquitetura de referência para implementação de um HIDS voltado para plataformas de contêineres. Técnicas e abordagens utilizadas em estudos do estado da arte no campo de pesquisa foram estudadas e aplicadas para desenvolvimento do *framework*. Como principais contribuições desse *framework* destaca-se o seguinte:

- Realização da coleta remota de *system calls* em diferentes nós de um *cluster* de orquestração de contêineres, possibilitando o aprendizado de anomalias de maneira colaborativa;
- Redução do *overhead* de processamento causado pela detecção de intrusão nos nós de um *cluster* de orquestração de contêineres através de uma arquitetura de HIDS com componentes distribuídos;
- Arquitetura escalável implementada com ferramentas gratuitas em um ambiente corporativo emulado e funcional;
- Capacidade de construção de *datasets* de *system calls* próprios e possibilidade de compartilhamento com a comunidade;
- Geração de alertas de detecção de anomalias em aplicações para apoio ao SOC através da análise de *system calls*;
- Análise dos dados através de interface web contendo *datasets* e anomalias indexadas;
- Possibilidade de implementação de diferentes algoritmos de *machine learning* e abordagens para detecção de anomalias em *system calls* (frequência, sequência, argumentos e outros dados) visando uma maior eficácia na detecção;
- Capacidade de integração do *framework* com outras ferramentas, aprimorando a segurança colaborativa.

Configurações relacionadas às ferramentas utilizadas na arquitetura do *framework* além dos códigos em Python construídos para implementação do módulo de *machine learning* foram disponibilizados como apêndices deste trabalho. Ainda, como resultado deste trabalho um artigo científico apresentando uma versão inicial do *framework* proposto foi publicado nos Anais da 17ª Conferência Ibérica de Sistemas e Tecnologias de Informação – CISTI 2022 [13]:

- S. L. Rocha, G. Daniel Amvame Nze and F. L. Lopes de Mendonça, "Intrusion Detection in Container Orchestration Clusters : A framework proposal based on real-time system call analysis with machine learning for anomaly detection,"2022 17th Iberian Conference on Information Systems and Technologies (CISTI), 2022, pp. 1-4, doi: 10.23919/CISTI54924.2022.9820103.

1.5 ORGANIZAÇÃO

O presente trabalho está estruturado em seis capítulos, sendo o capítulo 1 a introdução. O capítulo 2 discorre sobre os trabalhos correlatos com suas abordagens, principais resultados e limitações. No capítulo 3 é feita uma fundamentação teórica abordando conceitos fundamentais ligados à tecnologia de contêineres, detecção de intrusão e *machine learning*. O capítulo 4 apresenta a proposta de um *framework* para um HIDS voltado para plataformas de contêineres com uma arquitetura em camadas e integração entre elas. No capítulo 5 os experimentos realizados são descritos e resultados obtidos através da implementação do *framework* em um ambiente simulado são analisados. Por fim, no capítulo 6 a conclusão é apresentada acerca do trabalho conduzido, seus resultados, limitações e possibilidades de trabalhos futuros.

2 TRABALHOS CORRELATOS

Ao longo dos últimos anos, diversos estudos têm sido conduzidos visando o aprimoramento dos Sistemas de Detecção de Intrusão e a segurança de ambientes de computação em nuvem. Contudo, muitos desafios ainda se encontram sem uma solução definitiva e carecem de avanços que enderecem as lacunas existentes nesse campo de pesquisa. Devido à capacidade de detecção de novos ataques pelos IDS baseados em anomalia, a literatura do estado da arte acerca das técnicas de detecção de anomalia utilizadas em HIDS será explorada nesse trabalho.

Em um estudo de revisão sobre o desenvolvimento de IDS baseados em comportamento, [25] afirma que os IDS de ambientes de nuvem atuais sofrem com a baixa Precisão de Detecção (DR), alta Taxa de Falsos Positivos (FPR) e custos de operação. De acordo com [25], uma das principais limitações da tecnologia de detecção de intrusão atual é a necessidade de se reduzir a Taxa de Alarmes Falsos (FAR) para que as equipes de segurança não se confundam com esse tipo de informação. Nesse estudo, foi feito um levantamento de técnicas recentes para detecção de intrusão baseadas em comportamento anômalo com as vantagens e desvantagens dos algoritmos utilizados.

Em outro trabalho de revisão bibliográfica realizado por [10], desafios enfrentados pelos IDS atuais são discutidos. Dentre os principais desafios, limitações como a baixa DR, alta FPR e FAR, também são evidenciados. Nesse mesmo trabalho, uma taxonomia para classificação dos diferentes tipos de IDS, métodos de detecção de intrusão, problemas com *datasets* públicos existentes e técnicas de evasão também são analisados. De acordo com [10], a evolução dos *malwares* e a sofisticação cada vez maior dos ataques visando contornar os mecanismos de proteção existentes representa um desafio crítico no projeto e implementação dos IDS.

Em [8], foi realizado um estudo sobre detecção de intrusão, discutindo ataques populares, examinando os problemas associados à sua detecção e explorando possíveis soluções. Conforme o estudo, a detecção de ataques de *zero day*, as Taxas de Verdadeiros Negativos (TNR) e FPR, o *overhead* computacional e a detecção em tempo real continuam sendo um problema para os IDS. Assim, [8] destaca que é importante que pesquisadores continuem trabalhando para desenvolver melhores sistemas e algoritmos para minimizar os efeitos desses problemas, além de manter um bom desempenho. Do mesmo modo, [23] ressalta que pesquisas para otimizar os IDS visando o aumento da segurança e redução do *overhead* são necessárias.

Corroborando o entendimento de [25] e [8], [23] afirma que a principal desvantagem da detecção por anomalia é a precisão, mais notavelmente, essas técnicas sofrem com uma quantidade maior de alarmes falsos. Também é ressaltado por [20] que a velocidade e a precisão na detecção geralmente são difíceis de serem bem balanceadas. Outro aspecto levantado por [20] e [11] é que as métricas de avaliação de desempenho das técnicas utilizadas em muitas pesquisas não são consistentes. Ou seja, diferentes métricas e *datasets* são utilizados e dificultam a comparação dos resultados entre os estudos uma vez que não há uma padronização.

No artigo de revisão bibliográfica acerca dos HIDS baseados em *sytem calls*, [20] não aborda especificamente o cenário de detecção de intrusão em ambientes de contêineres. Contudo, importantes contri-

buições e direcionamentos podem ser obtidos do estudo realizado. Dentre elas, uma discussão sobre as principais limitações e indicações de trabalhos futuros nessa área de pesquisa são levantadas. Como principais tendências de pesquisa no contexto de HIDS, [20] relaciona as seguintes problemáticas: redução da FAR, melhoria da DR e aprimoramento da segurança colaborativa.

No que tange aos ambientes de contêineres, [17] e [18], afirmam que a detecção de intrusão tem sido amplamente utilizada em múltiplos contextos, contudo, sua aplicação a contêineres ainda é esparsa e tem sido negligenciada. A mesma constatação é destacada por [19], que afirma que enquanto muitos IDS foram introduzidos para garantir a segurança de ambientes de nuvem baseados em VM, poucos esforços têm sido direcionados para a área de segurança de contêineres em nuvem. De acordo com [26], a natureza dinâmica do desenvolvimento e operações de aplicações conteinerizadas dificulta a segurança desses ambientes. Assim, [26] considera o desenvolvimento de Sistemas de Detecção de Intrusão com técnicas de ML para a detecção ativa de contêineres e sistemas de análise como uma área para pesquisas futuras.

Em dois estudos, [27] e [28], publicados pelos mesmos autores, foi demonstrado que fatores como a dificuldade de se alterar um contêiner para monitorá-lo, especialmente quando aplicações críticas são executadas, fazem com que um HIDS seja o sistema de detecção mais apropriado para contêineres. Ademais, características *multi-tenancy* dos ambientes de contêiner, onde o *kernel* de um mesmo SO é compartilhado por múltiplos contêineres, fazem com que o *host* seja alvo potencial de ataques. Como fruto desses trabalhos, um HIDS foi implementado para realizar a detecção de anomalias no comportamento de contêineres monitorando as *system calls* entre o processo do contêiner e o *kernel* do *host*. Para tanto, a técnica conhecida como *Bag of System Calls* (BoSC), proposta por [29], foi utilizada para detecção de anomalias.

Dando continuidade ao estudo realizado por [28], em [19] foi apresentado um IDS Resiliente (RIDS) que implementou um sistema de resolução para nuvens baseadas em contêineres. O RIDS desenvolvido, se baseia em um mecanismo de monitoramento de comportamento inteligente de tempo real para detectar contêineres maliciosos e uma abordagem de defesa que consegue realizar a migração de contêineres em tempo de execução para uma zona de quarentena, visando minimizar a dispersão de um ataque. Resultados mostraram que esse RIDS foi capaz de detectar e migrar eficientemente contêineres de aplicações maliciosas garantindo uma operação segura de contêineres, sendo essa característica de defesa de alvo em movimento (*Moving Target Defense* - MTD), uma contribuição inovadora do estudo realizado.

No trabalho de [17], foi realizado um estudo onde algoritmos de detecção de anomalia foram analisados em ambientes conteinerizados. A abordagem utilizou os algoritmos *Sequence Time-Delaying Embedding* (STIDE) e BoSC, e ambos demostraram potencial para serem aplicados no contexto de detecção de intrusão. Contudo, a análise se restringiu ao desempenho de aprendizado dos algoritmos, e portanto, testes de avaliação dos algoritmos do estado da arte para detecção de intrusão foram recomendados para trabalhos futuros. Segundo os autores de [17]: “Esses experimentos ainda estão em sua infância visto que não há um conjuntos de dados (*datasets*) disponíveis para a detecção de intrusão em contêineres”. Para condução desse trabalho o artigo cita apoio do projeto Atmosphere, fundado pelo ministro da Ciência e Tecnologia do Brasil e tem participação da Universidade de Brasília (UnB) [30].

Um problema recorrente e citado em grande número de trabalhos como [20], [21], [31], [32], [14] e [33], diz respeito ao *overhead* de processamento ligado à detecção de anomalias. O grande volume de dados e a complexidade das técnicas envolvidas no processo de análise, muitas vezes acarreta em uma

penalidade de desempenho dos sistemas monitorados ou compromete a eficácia da detecção.

De acordo com [20], devido ao rápido desenvolvimento de técnicas e instalações de *data centers*, recentemente, os HIDS têm sofrido do bem conhecido desafio de *Big Data*. A alta FPR representa um desafio para os HIDS e novos registros de *system calls* estão sendo gerados com o surgimento de novas aplicações [20]. Dessa forma, métodos de mineração e sistemas de gerenciamento de banco de dados tradicionais em um *host* único podem não ser capazes de lidar com a quantidade massiva de *system calls* de maneira eficiente [20]. Ainda conforme [20], HIDS tradicionais estão em sua maioria realizando a análise de intrusão em um *host* independente com um *software* de detecção *standalone* instalado e não há interação entre os HIDS instalados nos diferentes *hosts* [20]. Como HIDS possuem apenas banco de dados de comportamentos normais ou conhecidos, novos conjuntos de *system calls* normais que não estão em conformidade com os bancos de dados ou modelos podem ser erroneamente reportados [20].

Na pesquisa conduzida por [24], vários impeditivos para implementar um IDS *standalone* foram encontrados. De acordo com a pesquisa, as altas FPR deixaram o sistema incapaz de detectar ataques de *zero-day* em tempo real. De acordo com [24] e [20], HIDS tradicionais baseados em *system calls* não conseguem alcançar um desempenho robusto como esperado. Assim, vários trabalhos nessa área estão em progresso e têm amplo escopo para futuro desenvolvimento [24].

Outra limitação existente na área envolve a defasagem dos *datasets* disponíveis atualmente para desenvolvimento dos IDS. De acordo com [22], [23], [10], [14], [33] e [11], os principais *datasets* conhecidos apresentam inúmeras deficiências tais como: obsolescência por antiguidade, falta de volume, ausência de dados complementares, além de a maioria não ser voltada para *system calls* de contêineres. No trabalho desenvolvido por [23], alguns dos principais *datasets* públicos disponíveis com dados de *hosts* úteis para um HIDS foram relacionados e organizados. A descrição, características e limitações de cada *dataset* foram compiladas para facilitar a utilização e pesquisas na área. No intuito de sanar essa lacuna envolvendo os *datasets*, [22] propõe uma metodologia para geração de novos *datasets* que podem ser úteis no contexto da detecção de intrusão.

Entre os trabalhos mais recentes sobre HIDS voltados para anomalias em *system calls*, muitas são as abordagens utilizadas para identificação de intrusões. A detecção de intrusão através da sequência de *system calls* foi apresentada em 1996 por [34] e ainda é utilizada em conjunto com técnicas de análises probabilísticas em trabalhos como [32], bem como redes neurais em [33]. Outros trabalhos como [19], [21], [31] e [35] fazem uso de abordagens baseadas na análise de frequência de *system calls*, e a utilização conjunta com redes neurais também é possível, conforme [36].

No estudo conduzido por [32], um HIDS de tempo real foi proposto para detecção de aplicações maliciosas executando em contêineres Docker. Para a detecção também foi utilizada a análise de *system calls*, mas utilizando uma abordagem com n-gramas de *system calls* e sua probabilidade de ocorrência. Nessa pesquisa, a escolha da análise sequencial de *system calls* foi preferível no lugar da análise baseada em frequência utilizada pelo BoSC, acreditando-se que essa última seria menos precisa. Nos experimentos realizados nesse estudo, foi atingida uma precisão de 87-97% na detecção de anomalias.

O algoritmo BoSC faz uso de uma abordagem baseada na análise da frequência de *system calls* e exige menor esforço computacional. Contudo, tendências gerais indicam que apesar de modelos baseados em sequência de *system calls* serem computacionalmente mais onerosos, eles oferecem uma melhor detecção

[23]. Também é notável que com a popularização de *datasets* rotulados diversas pesquisas têm surgido comparando IDS e testando muitos algoritmos de *machine learning* [23]. Diferentes técnicas de *machine learning* têm sido aplicadas nos HIDS para aprimorar o desempenho da detecção visando o aumento da precisão e a redução da FPR. Contudo, menos ênfase é dada na implementação prática de um HIDS para detecção em tempo real [21].

No que tange a cenários onde plataformas de orquestração de contêineres são utilizadas como ambientes de produção, poucos são os estudos de implementação de HIDS baseados em anomalias de *system calls* desenvolvidos até o momento [13]. No trabalho de [36], um *framework* para aprendizado distribuído foi desenvolvido visando a construção de modelos de detecção por aplicação através de redes neurais. Contudo, é sabido que o sistema implementado em cada *host* da plataforma de contêineres gera um *overhead* computacional que compete com a carga de trabalho real de aplicações e esse *overhead* não foi considerado [13].

Em [37], foi proposto um HIDS voltado para um *cluster* Kubernetes com detecção de anomalias através de redes neurais com aprendizado supervisionado e quatro categorias de *system calls*. Apesar do sistema ser capaz de monitorar os diversos *hosts* do *cluster* e realizar a detecção em componente externo, as regras de filtragem a partir de um conjunto limitado de *system calls* podem restringir o escopo de detecção de ataques [13]. Outro aspecto diz respeito à limitação acerca da falta de análise das anomalias reportadas e a necessidade de desenvolvimento de subcomponentes próprios como um portal web e um serviço de API *Restful* para implementação do IDS [13].

Nesse trabalho, um *framework* contendo uma arquitetura de referência para implementação de um HIDS voltado para plataformas de contêineres é proposto. Questões como o *overhead* computacional, o aprimoramento da eficácia de detecção através do emprego de *machine learning*, a construção de novos *datasets* e o monitoramento e análise de anomalias com ferramentas de apoio são apresentados como pontos-chave da arquitetura proposta.

3 FUNDAMENTAÇÃO TEÓRICA

3.1 VIRTUALIZAÇÃO

O conceito de virtualização surgiu nos anos 1960 com [38], no entanto seu grande impulso se deu com a utilização de servidores, próximo dos anos 2000. A virtualização é uma tecnologia que permite um melhor aproveitamento e compartilhamento dos recursos de infraestrutura de TI. Através dela é possível que o mesmo servidor físico possa ser compartilhado por diferentes Sistemas Operacionais simultaneamente. Além do mais, a virtualização traz uma série de benefícios como maior mobilidade das cargas de trabalho, aumento do desempenho e da disponibilidade dos recursos, automação das operações, simplificação do gerenciamento de TI e redução de custos operacionais [39].

Tecnologias de virtualização têm tido um papel preponderante durante os últimos anos devido ao crescente número de soluções em *software* em surgimento [40]. De acordo com [41], um dos fatores mais importantes e responsáveis pela baixa eficiência de consumo energético nos *data centers* é a má utilização de servidores físicos. Nesse sentido, a virtualização é principalmente utilizada por provedores de nuvem para aprimorar a utilização dos recursos computacionais, aumentar a segurança e a portabilidade, além de reduzir o custo total do consumo de energia dos *data centers* [42]. Em razão de tais benefícios, empresas de variados portes também têm feito uso da virtualização para melhor aproveitamento de recursos e retorno sobre o investimento em tecnologia.

3.1.1 Máquinas Virtuais

Para prover o compartilhamento de recursos de um mesmo servidor físico, uma abstração do *hardware* físico é criada através de uma camada de *software* chamada *hypervisor*. As Máquinas Virtuais, também conhecidas como VMs, são implementações de um sistema completo e executam de maneira isolada sobre o mesmo servidor físico interagindo com o *hypervisor* instalado. Um mesmo servidor físico pode abrigar várias VMs, onde cada VM pode fazer uso de um Sistema Operacional diferente e independente de outras VMs, bem como do servidor físico hospedeiro [42]. Diversas plataformas de virtualização estão disponíveis hoje como alternativas para implementação de Máquinas Virtuais. Para citar apenas algumas, pode-se destacar: Vmware, KVM, XEN e Hyper-V.

Apesar das vantagens oferecidas pelo uso de VMs, ambientes que fazem uso dessa tecnologia enfrentam desafios como baixa resiliência, dificuldade na alocação de recursos compartilhados e degradação de performance [43]. Além disso, devido ao fato de cada VM necessitar a instalação de um Sistema Operacional completo, incluindo *drivers*, sistema de arquivos, além da virtualização do *hardware*, o *overhead* causado no desempenho não pode ser desprezado [44]. Deste modo, alternativas como contêineres passaram a ser cada vez mais utilizadas em substituição às Máquinas Virtuais em diversos contextos.

3.1.2 Contêineres

A virtualização implementada através de contêineres também é conhecida como virtualização leve ou virtualização de Sistema Operacional. A principal diferença entre VMs e contêineres é que cada contêiner normalmente executa apenas uma única aplicação [45]. Para seu funcionamento os contêineres compartilham o *kernel* do Sistema Operacional hospedeiro em um mesmo *host* e encapsulam as dependências necessárias para o funcionamento da aplicação executada em uma estrutura de empacotamento conhecida como imagem. Recursos disponíveis no *kernel* Linux como *namespaces*, *cgroups* e *chroot*, possibilitam o controle e isolamento de recursos. Essas características dão impressão de se ter uma máquina própria com desempenho próximo ao nativo e sem nenhum *overhead* adicional de virtualização [46].

O Sistema Operacional hospedeiro utilizado para execução de um contêiner pode estar instalado diretamente sobre *bare metal* ou ser até mesmo virtualizado. Essa flexibilidade permite que os contêineres sejam utilizados em conjunto com VMs, usufruindo dos benefícios dessa tecnologia e agregando funcionalidades trazidas pelos contêineres. A Figura 3.1 ilustra a diferença de arquitetura entre VMs, contêineres sobre *bare metal* e contêineres em VMs. Originalmente desenvolvidos em sistemas Unix, posteriormente foram suportados no Linux, e atualmente já é possível executar contêineres em diferentes versões e famílias de Sistemas Operacionais.

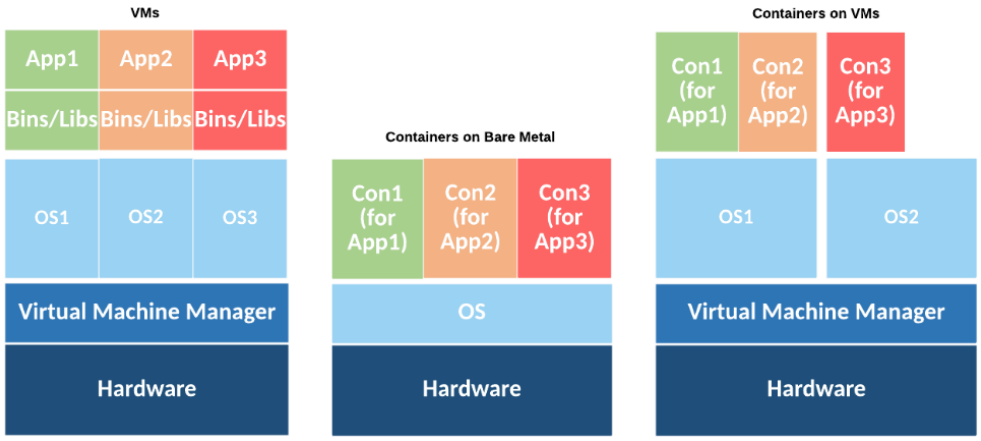

Figura 3.1: VM, contêiner em *Bare Metal*, contêiner em VM [1].

O termo contêiner é entendido como uma analogia aos contêineres usados em navios de carga, que fornecem uma forma padronizada de agrupar conteúdos díspares enquanto os isolam uns dos outros [45]. Uma das principais motivações para a crescente adoção dos contêineres é a conveniência para encapsular, implantar e isolar aplicações, além da flexibilidade e eficiência no compartilhamento de recursos [43]. Dessa forma, recentemente o uso de contêineres têm se tornado popular devido aos avanços obtidos pela sua utilização e maior foco na agilidade de desenvolvimento de aplicações como um benefício chave [1].

Outra característica fundamental de um contêiner é a sua natureza imutável. Definido através de uma imagem composta de um sistema de arquivos em camadas identificáveis através de *hash*, cada contêiner pode ser considerado como uma instância de uma imagem específica em execução. Para garantir a escrita e persistência de dados, um contêiner pode acessar um sistema de arquivos externo para essa finalidade. Devido ao fato de um contêiner não necessitar de um SO completo em sua imagem, mas conter essencialmente as bibliotecas necessárias para uma aplicação, isso faz com que seu processo de inicialização seja

significativamente mais rápido quando comparado a uma VM. Dessa forma, caso necessário, um contêiner pode ser terminado e um novo contêiner criado em substituição ao antigo com certa agilidade, tornando essa operação mais viável do que o modelo tradicional de aplicações em VMs. Essas e outras características de funcionamento e operação de contêineres possibilitou o surgimento de um novo paradigma de computação.

3.2 PLATAFORMAS DE ORQUESTRAÇÃO DE CONTÊINERES

Um dos blocos de construção centrais das plataformas de nuvem são os contêineres Linux, que simplificam a implantação e gerenciamento de aplicações e agregam escalabilidade [46]. Em ambientes de larga escala, com dezenas ou centenas de *hosts* e milhares de contêineres a gerência e administração individual de cada *host* se torna impraticável de ser realizada com eficiência. Para resolver esse problema, uma ferramenta conhecida como orquestrador possibilita o gerenciamento de múltiplos *hosts* de contêineres em um *cluster*, monitorando o consumo de recursos, a execução de tarefas agendadas e o controle dos contêineres em execução [45]. Algumas das principais tecnologias de orquestração de contêineres disponíveis atualmente são: Kubernetes, Mesos e Nomad.

Orquestradores ou plataformas de orquestração de contêineres, possibilitam que tarefas do *pipeline* de DevOps sejam executadas permitindo que o ciclo de desenvolvimento de uma aplicação possa ocorrer de maneira automatizada, desde a disponibilização em ambiente de testes até sua implantação em ambiente de produção. Um orquestrador permite que as equipes de DevOps especifiquem configurações de execução de contêineres a partir de arquivos descritores contendo a imagem utilizada, recursos necessários para cada contêiner, bem como o número de réplicas desejado. A partir dessas informações o orquestrador gerencia a execução dos contêineres com as configurações definidas alocando-os nos *hosts* ou nós de um *cluster*, conforme a disponibilidade de recursos de cada *host*, mantendo a carga de trabalho do *cluster* balanceada. Também é possível que um orquestrador reinicie automaticamente um contêiner em caso de falha e movimente contêineres entre os diferentes nós de um *cluster* conforme a necessidade. Para que a comunicação via rede entre os contêineres em execução no diferentes nós de um *cluster* seja possível, muitos orquestradores fazem uso de redes definidas por *software* (SDN), que permitem o isolamento da comunicação entre aplicações e o compartilhamento da mesma rede física [45].

Para extrair o melhor benefício das soluções tecnológicas empregadas em um ambiente computacional é comum que haja a integração entre a plataforma de orquestração de contêineres e outras ferramentas. A Figura 3.2 contém uma relação de ferramentas que frequentemente são utilizadas pelas equipes de DevOps e fazem uso da plataforma de contêineres para execução de cargas de trabalho. Outras soluções de infraestrutura de TI para armazenamento de dados, balanceamento de carga, *backup* e controle do tráfego de rede também são integradas à plataforma de contêineres provendo funcionalidades para o funcionamento de sistemas e aplicações conteinerizadas.

[26] afirma que uma área para pesquisas futuras é o desenvolvimento de Sistemas de Detecção de Intrusão com técnicas de ML para a detecção ativa de contêineres e sistemas de análise. De acordo com [26], a natureza dinâmica do desenvolvimento e operações de aplicações conteinerizadas dificulta a segurança

desses ambientes.

Devido à evolução das plataformas de orquestração de contêineres, *clusters* contendo um número cada vez maior de *hosts* têm surgido podendo compreender um alto poder computacional agregado. Empresas como a Google[1], podem executar bilhões de contêineres na plataforma de orquestração desenvolvida por ela [47]. Diversos provedores de nuvem pública e privada também têm fornecido modelos de serviços alicerçados sobre plataformas de contêineres. Outras funcionalidades como alta disponibilidade, escalabilidade e automação de procedimentos do *pipeline* de desenvolvimento de *software*, têm feito com que haja uma convergência cada vez maior para as plataformas de orquestração de contêineres à reboque de uma mudança de cultura tecnológica organizacional. Desse modo, é fundamental que essas plataformas atendam a requisitos de segurança e também estejam no radar das equipes do SOC.

3.3 COMPUTAÇÃO EM NUVEM

Em meados do ano de 2009 ainda não havia uma definição padronizada do conceito de computação em nuvem. Deste modo, [48] definiu o termo da seguinte forma: "Nuvens, ou *clusters* de computadores distribuídos, fornecem recursos e serviços sob demanda através de uma rede, geralmente a Internet, com a escala e confiabilidade de um *data center*".

De acordo com [48], as nuvens podem ser classificadas em dois tipos: as que fornecem instâncias de computação (*on-demand computing instances*) e as que fornecem capacidade de computação (*on-demand computing capacity*). Sendo que ambas utilizam o mesmo tipo de máquinas, mas o primeiro tipo é projetado para expandir fornecendo instâncias de computação adicionais, enquanto que o segundo é projetado para suportar aplicações que fazem uso intensivo de processamento ou de dados [48]. Ainda conforme [48], as nuvens podem fornecer dois tipos de serviço: SaaS (*Software as a Service*) e PaaS (*Plataform as a Service*), mencionando então dois termos bem conhecidos e utilizados ainda hoje para classificação de nuvens. Características essenciais presentes em nuvens de computação como escalabilidade, simplicidade e uma política de preços voltada para uso (*pay as you go*) também são citadas por [48], além dos modos de implementação de nuvem como sendo: privada (uso interno e exclusivo a uma empresa) ou pública (empresa que fornece serviços de nuvem para vários clientes através de recursos compartilhados).

No ano de 2011, [15] publicaram pelo NIST (*National Institute of Standards and Technology*) o que viria a ser uma referência para definição de computação em nuvem de maneira a fornecer um *baseline* para discussão acerca de conceitos e melhor utilização da computação em nuvem. A definição de nuvem proposta por [15] é a seguinte: "A computação em nuvem é um modelo que fornece um acesso ubíquo, conveniente e sob demanda através da rede a um conjunto compartilhado de recursos de computação configuráveis (como: redes, servidores, armazenamento, aplicativos e serviços) que possam ser rapidamente provisionados e liberados com o mínimo esforço de gerenciamento ou interação com o provedor de serviços.". Ainda de acordo com [15], esse modelo de computação em nuvem é composto por cinco características essenciais, três modelos de serviço e quatro formas de implementação. Quais são:

- Características essenciais: *self-service* sob demanda; amplo acesso por rede; *pool* de recursos; rápida

[1] <https://www.google.com/>

elasticidade e medição de serviço;

- Modelos de serviço: SaaS (*Software as a Service*); PaaS (*Plataform as a Service*) e IaaS (*Infrastructure as a Service*);
- Modelos de implementação: nuvem privada; nuvem comunitária; nuvem pública e nuvem híbrida.

Alguns dos conceitos propostos por [15] em relação às características essenciais de nuvens de computação bem como acerca dos modelos de serviço e de implementação de nuvens já eram utilizados anteriormente como visto em [48]. Contudo, novas classificações como o modelo de serviço IaaS, além dos modelos de nuvem híbrida e comunitária foram apresentados. A publicação de [15] introduziu definições que facilitaram o entendimento e a classificação de nuvens de computação de uma forma abrangente e que foi bem aceita como um padrão que não existia até o momento de sua publicação. Devido ao fato da computação em nuvem ser uma tecnologia que emergiu e continua em evolução, novas classificações e conceitos ligados ao assunto começaram a surgir desde o ano de 2011 e a definição e classificação proposta pelo NIST passou a ser complementada para melhor retratar o cenário da computação em nuvem atual. Esse é o tema principal abordado por [49], que questiona se a definição de nuvem proposta pelo NIST ainda é válida.

De acordo com [49], uma das primeiras empresas a utilizar o termo nuvem nos anúncios dos seus produtos foi a Amazon, com AWS, no ano de 2006. Desde então, a computação em nuvem têm aparecido nos relatórios da Gartner com seu próprio ciclo de *hype*. Ainda conforme [49], desde a definição de computação em nuvem apresentada pelo NIST em 2011, a indústria tem evoluído rapidamente para incluir diversas tecnologias como contêineres e a computação *serverless*. Uma evolução dos modelos de serviço SaaS, PaaS e IaaS apresentada por [49] é o XaaS (*Anything as a Service*), que tem por definição "uma tecnologia entregue pela Internet que costumava ser entregue *on site*". Provedores de nuvem são capazes de oferecer outras tecnologias como serviço devido ao rápido, confiável e crescente acesso à Internet [49]. Deste modo, alguns dos mais recentes tipos de XaaS são mencionados por [49]: *Blockchain PaaS*; BpaaS (*Business Process as a Service*); DBaaS (*Database as a Service*); FaaS (*Function as a Service*); *Malware as a Service*; e *Windows as a Service*.

Em [49] também é citado um novo modelo para computação em nuvem criado por Johan den Haan, CTO da Mendix, que é mais interativo e um *framework* mais granular, e pode enquadrar melhor as tecnologias que os provedores comerciais oferecem hoje. Esse modelo foi construído a partir da definição pré-existente do NIST além de outros relacionados e os integra em um *framework* composto por 8 camadas (*Hardware, Software Defined, Foundational* PaaS, PaaS, *Serveless Computing, Built-up PaaS, App Services* e SaaS) [49]. Esse modelo traz a virtualização, SDN (*Software Defined Network*) e armazenamento como serviços base; o modelo de serviço PaaS é particionado em subcamadas para melhor alinhamento com a evolução dos serviços; e também inclui o conceito de computação *serverless*, também conhecido como Função como Serviço [49]. Ainda de acordo com [49], a arquitetura *serverless* irá eliminar ou substituir o modelo de IaaS em alguns casos e o novo modelo mostra claramente que "existe muito mais do que apenas SaaS, PaaS e IaaS no cenário atual de nuvens de computação como originalmente conceitualizado e documentado pelo NIST".

Dos conceitos apresentados por [48] em 2009 até a publicação de [49] em 2018, é possível notar uma

evolução acerca dos conceitos e terminologias utilizadas na computação em nuvem, principalmente após a padronização proposta por [15] em 2011, que serve como um verdadeira referência conceitual sobre o assunto. Contudo, conforme visto em [49], apesar da definição de computação em nuvem do NIST permanecer como a mais simples, precisa e abrangente, espera-se que em breve haja uma atualização para melhor acompanhar a crescente evolução das tecnologias em uso na computação em nuvem.

Nesse trabalho, o modelo de serviço PaaS será escopo de análise no que diz respeito à segurança de plataformas de orquestração de contêineres, mais especificamente, endereçando a detecção de intrusão nesse tipo de ambiente de computação em nuvem que têm ganhado proeminência nos últimos anos.

3.4 DESENVOLVIMENTO DE APLICAÇÕES E OPERAÇÕES DE TI

O Desenvolvimento de aplicações e Operações de TI ou DevOps, do termo em inglês, diz respeito a um conjunto de práticas que visa a integração entre o desenvolvimento e implantação de aplicações enfatizando uma colaboração mais próxima entre as equipes de desenvolvimento e operações de TI [45]. Assim, pode-se dizer que o DevOps é uma mudança de paradigma organizacional, onde ao invés de se ter grupos divididos em silos executando funções separadamente, equipes multidisciplinares aceleram a resolução de problemas e trabalham em entregas contínuas de maneira integrada [2].

As práticas de DevOps visam entregar valor continuamente no provimento de soluções por parte das equipes de TI através de uma cultura de trabalho composta por metodologias ágeis, automação e tecnologias como a computação em nuvem e contêineres. Ferramentas são um fator mandatório para possibilitar as práticas de DevOps e a escolha das ferramentas certas é uma importante questão para um projeto [2]. Diversas ferramentas podem ser utilizadas nas fases do ciclo DevOps. Contudo, a escolha das ferramentas mais apropriadas deve levar em consideração o ambiente e a arquitetura de produção utilizada [2]. A Figura 3.2 lista diferentes ferramentas usadas para automação nas fases de DevOps e traz outros detalhes como modelo de licenciamento, linguagem e tipo de cada ferramenta.

Uma das práticas de DevOps que traz mais agilidade ao *pipeline* de desenvolvimento de *software* é a Entrega Contínua (CD). Através da Entrega Contínua uma nova versão de um *software* pode ser rapidamente implantada em ambiente de produção de maneira automatizada pelas equipes de desenvolvimento sem a necessidade de intervenção manual de outras equipes, em contraste com o processo de desenvolvimento tradicional de *software*, onde há forte dependência das equipes de operação nesse processo. A prática de CD permite que correções de *bugs* e falhas de segurança em uma aplicação possam ser corrigidas rapidamente, mas também possibilita que versões vulneráveis sejam colocadas em produção com maior facilidade caso não haja um processo de verificação de segurança no *pipeline* de desenvolvimento.

A arquitetura de microsserviços é outra tecnologia que tem se estabelecido e está inserida no contexto de DevOps. Devido à sua característica onde um *software* complexo é decomposto em muitos serviços fracamente acoplados, ela se torna bem adaptada para execução em ambientes de contêiner [22]. A migração da arquitetura de grandes aplicações monolíticas para microsserviços traz muitos benefícios como a flexibilidade de migração de tecnologia, menor tempo para entrada em produção e melhor estruturação das equipes de desenvolvimento em torno de serviços [50]. Desse modo, a utilização de microsserviços

Tool	DevOps phase	Tool type	Configuration format	Language	License
Ant	Build	Build	XML	Java	Apache
Maven	Build	Build	XML	Java	Apache
Rake	Build	Build	Ruby	Ruby	MIT
Gradle	Build	Build	Based on Groovy	Java and a Groovy-based domain-specific language (DSL)	Apache
Jenkins	Build	Continuous integration	UI	Java	MIT
TeamCity	Build	Continuous integration	UI	Java	Commercial
Bamboo	Build	Continuous integration	UI	Java	Commercial
Puppet	Deployment	Configuration management	DSL similar to JSON (JavaScript Object Notation)	Ruby	Apache
Chef	Deployment	Configuration management	Ruby-based DSL	Ruby	Apache
Ansible	Deployment	Configuration management	YAML (YAML Ain't Markup Language)	Python	GPL (GNU General Public License)
Loggly	Operations	Logging	—	Cloud based	Commercial
Graylog	Operations	Logging	—	Java	Open source
Nagios	Operations	Monitoring	—	C	Open source and GPL
New Relic	Operations	Monitoring	—	—	Commercial
Cacti	Operations	Monitoring	—	PHP	GPL

Figura 3.2: Ferramentas de automação utilizadas nas fases de DevOps [2].

permite a implantação efetiva da cultura DevOps promovendo a importância de pequenas equipes [50].

Apesar de muitas empresas serem beneficiadas pela cultura DevOps e aumentarem sua eficiência, a mudança de cultura nas organizações é algo que não pode ser subestimado pois apresenta desafios [2]. Quando as equipes de desenvolvimento de *software* embarcam na jornada rumo ao DevOps é preciso adotar uma abordagem *full-stack*, onde é preciso ter um conhecimento que vai além do código e se assume a responsabilidade peles testes e implantações [2]. Da mesma forma, as equipes de infraestrutura e operações devem colaborar com as equipes de desenvolvimento e prover as ferramentas, acessos e registros necessários para que as práticas de DevOps

3.5 SEGURANÇA CIBERNÉTICA

Dada a arquitetura e abrangência da Internet, sendo um meio de comunicação compartilhado entre muitos países para finalidades que vão desde o comércio eletrônico até as redes sociais, o ciberespaço é

composto por muitas tecnologias globalmente distribuídas e interconectadas através da Internet. Tecnologias como o 5G, a computação em nuvem, Internet das Coisas (IoT), dentre outras, têm possibilitado a ampla conectividade de dispositivos à Internet. Conforme o número de sistemas conectados à Internet cresce, a superfície de ataque consequentemente aumenta, levando a maiores riscos de ataques [11].

Incidentes envolvendo crimes cibernéticos de alto nível demonstram a facilidade com que ameaças cibernéticas podem se espalhar internacionalmente, uma vez que um simples comprometimento pode causar disrupção de serviços ou instalações críticas [10]. De acordo com [51], o espaço cibernético tem se tornado uma zona de guerra em uma nova era de combates ideológicos. Desse modo, o espaço cibernético é também uma nova dimensão onde grupos rivais, organizações criminosas e até mesmo nações se enfrentam travando guerras através dos meios tecnológicos. Cada vez mais países têm investido na formação de equipes especializadas em combates cibernéticos formando verdadeiros exércitos cibernéticos. Acordos cooperativos para aprimoramento das defesas cibernéticas como os realizados entre o Japão e Vietnã, e entre a Austrália e Singapura também tem crescido [51].

A segurança cibernética é formada pela coleção de políticas, tecnologias e processos que juntos trabalham para proteger a confidencialidade, integridade e disponibilidade de recursos computacionais, redes, *softwares* e dados contra ataques [11]. Técnicas empregadas por usuários maliciosos têm se tornado cada vez mais sofisticadas, abrangendo desde *exploits* do tipo *zero-day* a *malwares* para contornar medidas de segurança implementadas, possibilitando que eles persistam por longos períodos sem que sejam notados [11].

De acordo com o relatório *X-Force Threat Intelligence Index 2022* da empresa IBM, entre 2021 e 2022 houve um aumento de 33% no número de incidentes causados por exploração de vulnerabilidades. Quatro das cinco primeiras vulnerabilidades mais exploradas em 2021 são novas [52]. Outros dados do mesmo relatório indicam um aumento de 146% de *malwares* do tipo *ransonware* com código inédito voltados para o SO Linux no ano de 2021.

Em outro relatório de segurança *Sophos 2022 Threat Report* da empresa Sophos, em referência ao ano de 2021, foi constatado que praticamente em todas as semanas do ano houve sérios ataques cibernéticos que ameaçaram a infraestrutura de milhares de grandes empresas e organizações na Internet. No ano de 2021, vulnerabilidades em *softwares* contribuíram para ataques massivos contra a infraestrutura que sustenta alguns dos serviços básicos de Internet e causaram muitos problemas para administradores de TI lidarem com a variedade de ataques ocorridos [53].

O relatório de segurança da empresa SANS, *SANS 2021 Top New Attacks and Threat Report*, indica que mais de 131 milhões de pessoas foram impactados por vazamentos de dados individuais apenas nos três primeiros meses do ano de 2020. Nesse mesmo período, foram feitas 239 divulgações de dados de empresas de diversos setores, sendo que 37% dos vazamentos foram de empresas do setor de saúde [54]. O mesmo relatório expõe um aumento significativo de ataques ligados à cadeia de suprimentos entre os anos de 2020 e 2021, causando impactos complexos a empresas e negócios [54].

Com a rápida propagação da informação pela Internet, o compartilhamento sem fronteiras de informações acerca dos mais diversos domínios do conhecimento se tornou possível, abrangendo inclusive o domínio da segurança cibernética. Desse modo, a disseminação de *malwares*, ferramentas e *exploits* para exploração de vulnerabilidades, bem como uma vasta biblioteca com procedimentos detalhados para exe-

cução de ataques também pode ser encontrada na Internet sem grandes esforços. Isso possibilitou que muitos ataques pudessem ser reproduzidos por outros usuários de maneira facilitada, ganhando assim popularidade. A Figura 3.3 ilustra a evolução da sofisticação dos ataques em relação ao nível de conhecimento técnico, demonstrando assim a queda do conhecimento técnico necessário para execução de ataques complexos ao longo dos anos.

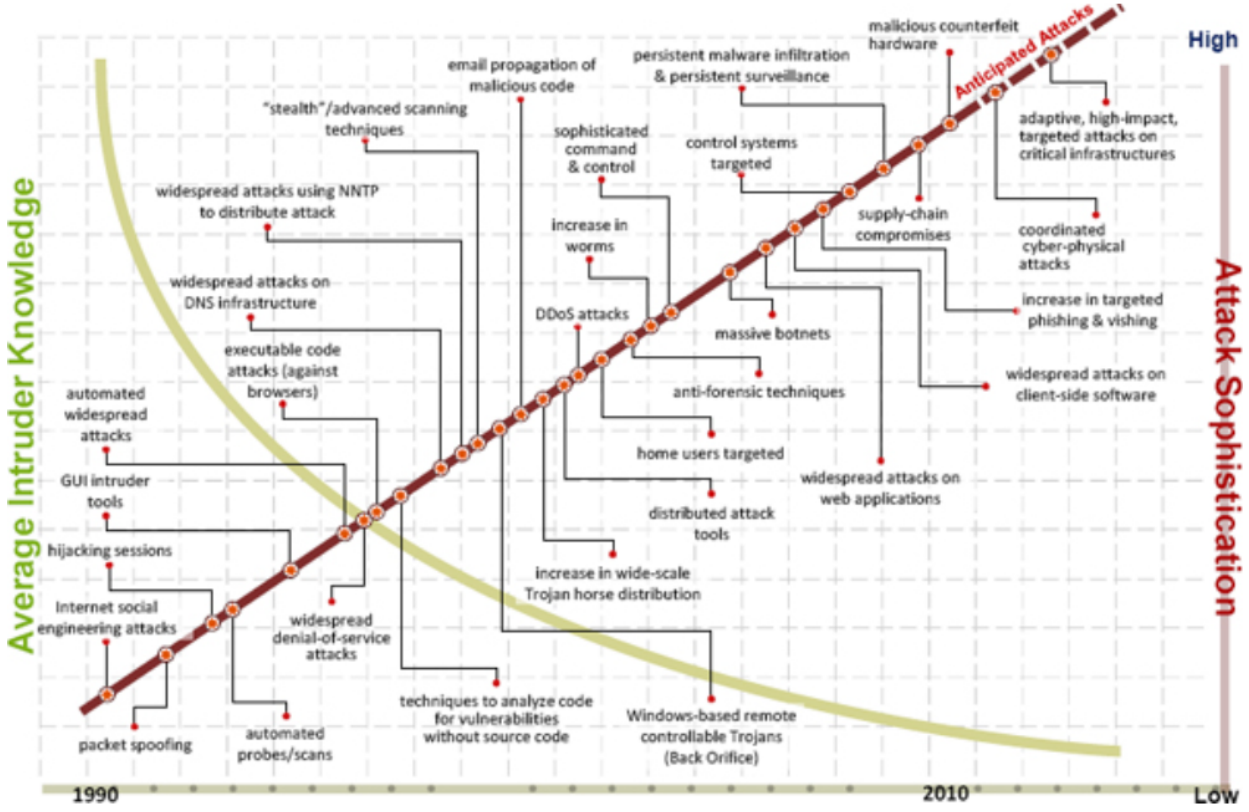

Figura 3.3: Evolução da sofisticação dos ataques e conhecimento técnico necessário [3].

Devido à crescente utilização de contêineres em ambientes corporativos, essa tecnologia tem despertado cada vez mais interesse no contexto da segurança cibernética e plataformas de contêineres têm sido alvos de ataques. De acordo com o relatório da empresa Red Hat, *State of Kubernetes Security Report*, aproximadamente 94% das empresas que participaram de uma pesquisa acerca da segurança da plataforma de orquestração de contêineres Kubernetes reportaram que identificaram pelo menos um incidente de segurança nos últimos doze meses nessa plataforma ou outros ambientes de contêiner. A principal causa identificada para os incidentes foi a falha de configuração [4]. A Figura 3.4 descreve as principais outras causas de incidentes reportadas pelo relatório.

Ainda, o relatório de ameaças *ENISA Threat Landscape 2022*, elaborado pela ENISA, afirma que o advento da COVID-19 têm acelerado a adoção de serviços de nuvem em apoio aos processos de negócios das organizações. Como os cibercriminosos seguem as tendências em tecnologias, não é surpresa que estejam tendo como alvo ambientes de nuvem [55]. Ainda de acordo com o mesmo relatório, como formas de ataque aos ambientes de nuvem têm-se a exploração de imagens de contêineres mal configuradas, além de contêineres Docker e *clusters* Kubernetes mal configurados.

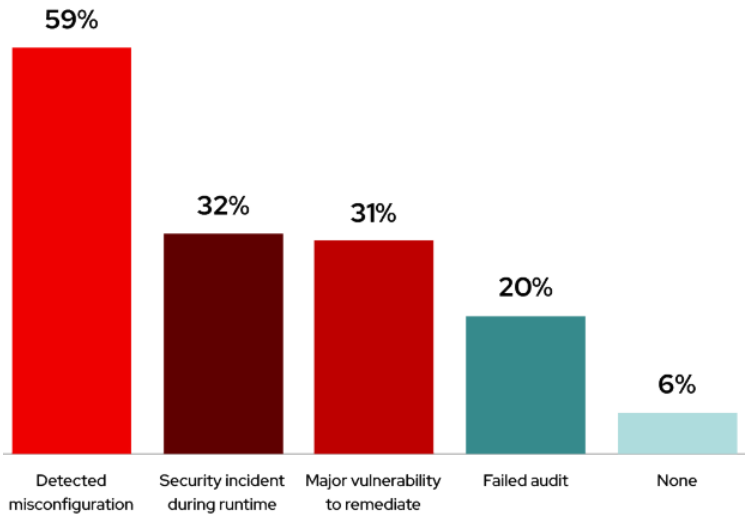

Figura 3.4: Principais causas de incidentes em plataformas de orquestração de contêineres [4].

Outros riscos relacionados aos componentes fundamentais das tecnologias de contêineres podem ser enquadrados em duas categorias principais [45]:

- Comprometimento de uma imagem ou contêiner
- Mau uso de um contêiner servindo como meio para execução de outros ataques contra: outros contêineres, *hosts* do *cluster*, etc.

Como referência adicional sobre os ataques voltados para contêiner, a Figura 3.5 representa uma matriz contendo as técnicas e táticas utilizadas contra as tecnologias de contêineres identificadas pelo MITRE. Nessa figura, muitas das técnicas elencadas são decorrentes de falhas de configuração na imagem de contêineres, *clusters* de orquestração e mau uso de contêineres em execução, conforme citado anteriormente. Ainda sobre a Figura 3.5, as técnicas representadas com um grifo lateral contendo o símbolo II podem ser desmembradas em sub-técnicas que podem ser consultadas em [5].

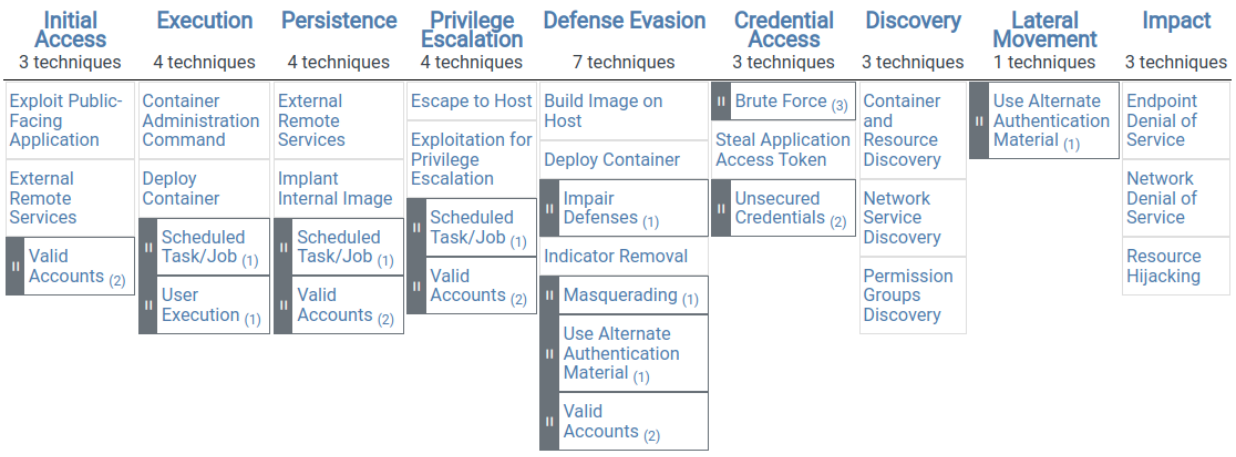

Figura 3.5: Matriz de técnicas e táticas empregadas contra as tecnologias de contêineres mapeadas pelo MITRE [5].

3.6 CENTRO DE OPERAÇÕES DE SEGURANÇA

Centros de operações surgiram como consequência de avanços tecnológicos ao longo dos anos. Normalmente o termo é associado a um esteriótipo onde se tem uma sala com diversos especialistas que operam computadores e outros equipamentos, monitorando informações críticas projetadas em um *videowall* [56]. Muitos tipos de centro de operações podem existir, sendo que em todos eles há um propósito geral de execução de tarefas específicas além do acesso e troca de informações por um grupo de pessoas que interagem entre si [56].

Um Centro de Operações de Segurança (SOC) é um tipo de centro de operações que atua como uma equipe composta por profissionais especializados que executam procedimentos operacionais definidos e apoiados por tecnologias de segurança que funcionam de maneira integrada [6]. Como escopo de análise do SOC estão informações ligadas à segurança da informação oriundas de equipamentos e ferramentas como antivírus, *firewalls*, IDS, balanceadores de carga, *logs* de auditoria de sistemas, dentre outras fontes, normalmente concentradas em uma ferramenta de Gerenciamento de Eventos e Informações de Segurança (SIEM). Desse modo, o propósito de um SOC é prover serviços de detecção e reação a incidentes de segurança, atuando em operações de geração, coleta, armazenamento, análise e reação a eventos de segurança [57].

Um SOC incorpora uma mistura de pessoas, tecnologias, processos e governança, para identificar, detectar e mitigar eficientemente ameaças idealmente antes que danos ocorram [58]. No que diz respeito à composição da equipe do SOC, é necessário que profissionais especializados sejam alocados em diferentes funções e níveis de atuação de maneira que um processo colaborativo e hierárquico de trabalho possa ser desempenhado. A Figura 3.6 ilustra os componentes de um SOC com suas respectivas funções conforme proposto por [6].

Ainda de acordo com [6], as atividades desempenhadas dentro de um SOC podem ser agrupadas em cinco áreas principais conforme descrito na Tabela 3.1.

Tabela 3.1: Áreas de atividades de um SOC [6].

Tipo de função	Descrição da atividade
Inteligência	Análise de incidentes de segurança Análise de padrões de ataques
Baseline de segurança	Varreduras de conformidade Varreduras de vulnerabilidades
Monitoramento	Observação Correlação e seleção de logs
Pentest	Execução de testes de penetração
Forense	Análise de logs Investigações

Um requisito para uma correta operação de um SOC é a definição de uma hierarquia funcional responsável pela segurança da informação institucional que atue de acordo com a missão e objetivos de segurança corporativos [6]. A definição de políticas e requisitos de segurança da informação também devem estar alinhadas com métodos e processos como a Análise de Impacto ao Negócio (BIA) e Análise de Riscos (RA) da organização [6]. Assim sendo, um SOC pode prover uma solução abrangente para detectar e mitigar ataques, se implementado corretamente [58].

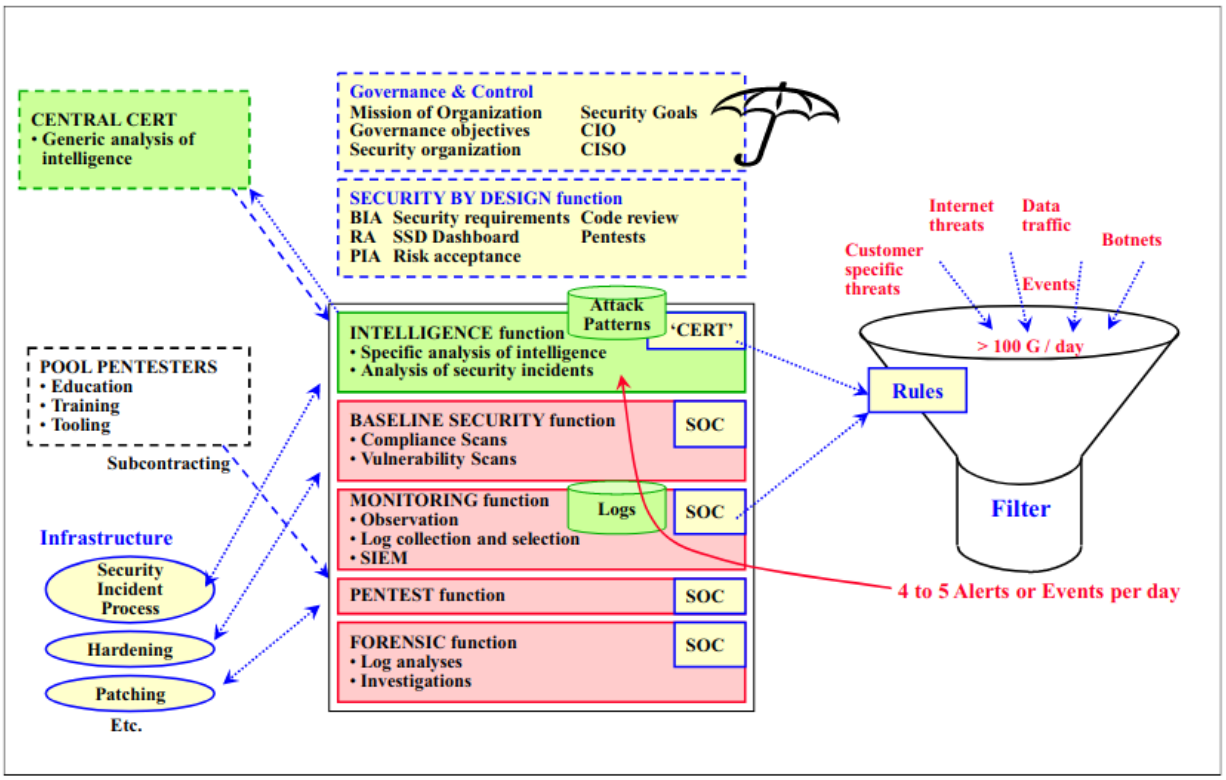

Figura 3.6: Componentes e funções de um SOC [6].

Com o surgimento crescente de novas aplicações e tecnologias que fazem uso da infraestrutura de TI, um volume de dados cada vez maior é trafegado em uma rede corporativa. Ademais, redes modernas têm adquirido ampla conectividade com a Internet a velocidades de 100 Gbps ou mais, computando um volume médio de dados existentes em torno de 44ZB até 2022 [59]. Consequentemente, a quantidade de informações analisadas por um SOC aumenta e atividade de análise se torna uma tarefa onerosa.

Um analista de segurança cibernética experiente desempenha um importante papel analisando cautelosamente eventos de segurança que chegam ao SOC bem como alertas gerados por ferramentas devido a violações de regras e parâmetros configurados. Técnicas de evasão avançadas e outros ataques sofisticados podem passar desapercebidos por ferramentas de detecção, mas podem ser percebidos por investigadores experientes, gerando assim uma dependência do fator humano para identificar certos tipos de ameaças [6]. Ainda, um grande número de alertas normalmente é gerado por um SIEM e chegam ao SOC com uma alta taxa de falso positivos (FPR), podendo chegar a centenas de milhares de ocorrências em um dia e excedendo a capacidade de investigação do SOC [60].

De acordo com o relatório de segurança [53], publicado pela empresa Sophos no ano de 2022, acredita-se que nos próximos anos tecnologias de *Machine Learning* irão tornar as ferramentas de segurança mais intuitivas. Assim, um SOC apoiado por Inteligência Artificial poderá desempenhar suas funções com maior facilidade e eficiência em comparação aos SOCs atuais [53].

Diante dessa problemática, soluções que agreguem inteligência na análise de eventos de segurança e que ajudem a reduzir o alto número de falsos alertas têm se tornado cada vez mais importantes para otimizar o trabalho da equipe do SOC.

3.7 SISTEMAS DE DETECÇÃO DE INTRUSÃO

Um dos recursos mais utilizados visando garantir um nível de segurança adequado aos ambientes computacionais modernos é a utilização de Sistemas de Detecção de Intrusão. A detecção de intrusão é o processo de monitoramento dos eventos que ocorrem em um sistema de computador ou rede, analisando-os em busca de indícios de possíveis incidentes [1]. Uma intrusão pode ser definida como qualquer atividade não autorizada com potencial para causar danos a um sistema informatizado [10]. Ou seja, qualquer ataque que possa provocar uma possível ameaça à confidencialidade, integridade ou disponibilidade da informação [10].

O surgimento dos IDS se deu em meados de 1980, na Força Aérea dos Estados Unidos (USAF), onde os fundamentos da segurança da informação foram lançados por meio de um grupo de trabalho com foco em segurança de redes [25]. Mais tarde, uma primeira forma de IDS foi criada a partir do funcionamento dos antivírus, focando na varredura do tráfego de rede em busca de ameaças contidas em uma lista de ameaças conhecidas [25]. Avanços posteriores seguiram contribuindo para o desenvolvimento da tecnologia empregada nos IDS modernos, o que engloba técnicas manuais, algoritmos e mecanismos automatizados visando o monitoramento contínuo de ambientes computacionais em busca de sinais de comprometimento [23]. A Figura 3.7 ilustra a evolução dos Sistemas de Detecção de Intrusão nos estágios iniciais do seu surgimento.

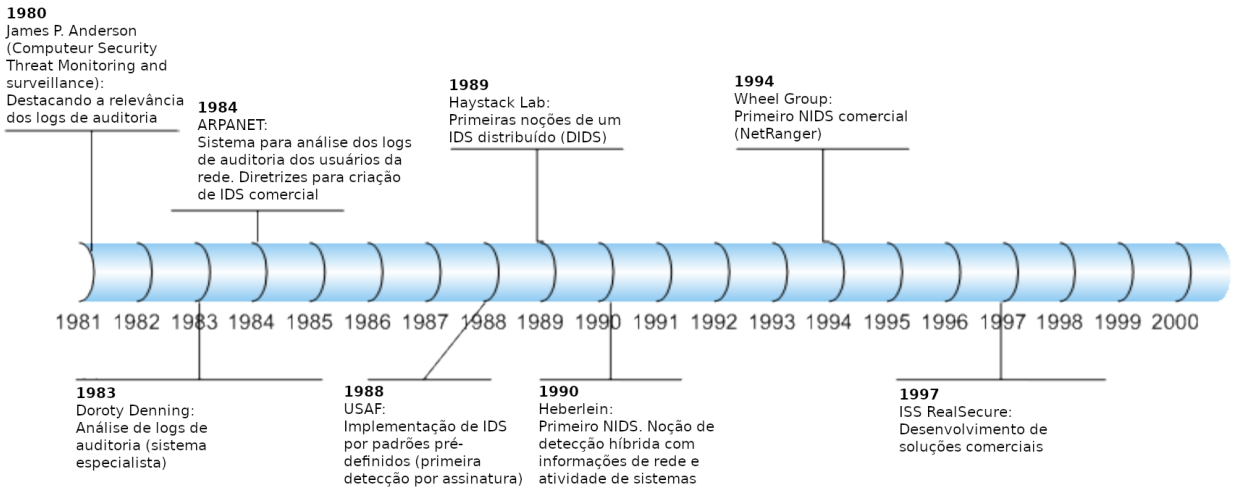

Figura 3.7: Evolução dos Sistemas de Detecção de Intrusão [7].

Pode-se dizer que o objetivo principal de um IDS é identificar um tráfego de rede ou uso malicioso de sistemas computacionais, que não podem ser identificados por um *firewall* tradicional [10]. Alguns requisitos são esperados de um bom IDS, como a habilidade de detectar uma grande variedade de ataques, incluindo ataques já conhecidos bem como desconhecidos (*zero-day*) [8]. Além disso, é esperado que um IDS tenha a flexibilidade para se adaptar e aprender novos ataques e mudanças no comportamento dos usuários, além de ser capaz de detectar ataques de origem externa e interna [8].

Mesmo com os avanços nas tecnologias de detecção de intrusão, ataques cibernéticos continuam ocorrendo e necessitam de vigilância contínua através do aprimoramento dos IDS e introdução de novas abordagens para garantir a segurança [61]. Desse modo, é impossível que sistemas de segurança, por mais bem

projetados que sejam, sejam capazes de prevenir todos as tentativas de intrusão, haja vista a quantidade de novas vulnerabilidades expostas diariamente [8]. Sendo assim, é importante identificar uma intrusão mesmo após a sua ocorrência, uma vez que essa informação pode ajudar a determinar o impacto de um ataque ocorrido e a estabelecer medidas de prevenção contra ataques futuros [8].

Sistemas de Detecção de Intrusão normalmente podem ser categorizados em dois grupos de acordo com o método de detecção utilizado: Sistemas de Detecção de Intrusão baseados em Assinatura (SIDS) e Sistemas de Detecção baseados em Anomalia (AIDS) [10]. Alguns trabalhos incluem também uma terceira categoria de IDS designada como híbrida, onde os métodos de detecção por assinatura e anomalia são utilizados em conjunto para obter melhores resultados.

Outra distinção comum entre os IDS é feita com relação à sua localização na infraestrutura de TI, sendo as abordagens mais comuns denominadas: Sistemas de Detecção de Intrusão de Rede (NIDS) e Sistemas de Detecção de Intrusão baseados em *Host* (HIDS). A combinação dessas duas abordagens juntas resulta no que é conhecido como Sistema de Detecção de Intrusão Colaborativo (CIDS). Um CIDS busca extrair o melhor dos NIDS e HIDS combinando-os colaborativamente junto com outros mecanismos de segurança para se obter uma melhor eficiência na detecção de intrusão [20]. De acordo com [20], a construção de CIDS tem se apresentado como uma tendência para o futuro. A Tabela 3.2 apresenta as formas de classificação de IDS conforme o critério utilizado.

Tabela 3.2: Critérios utilizados para classificação de IDS.

Critérios	**Método de detecção**	**Localização na infraestrutura**
Classificação	SIDS	NIDS
	AIDS	HIDS
	SIDS + AIDS (Híbrido)	NIDS + HIDS (CIDS – Colaborativo)

De maneira geral os IDS costumam ter três componentes principais [23]:

- Coletores de dados – responsáveis pela captura de um ou mais tipos de dados como tráfego de rede ou *system calls*
- Conversor de *atributos* – cria um vetor de atributos a partir dos dados coletados baseado em uma lista de atributos pré-definida
- Módulo de decisão – executa um algoritmo ou heurística para decidir se um dado representado por um vetor de atributos é considerado malicioso ou não

IDS em suas diferentes formas de implementação têm demonstrado aplicabilidade para proteção de ambientes computacionais de tamanhos e complexidades diversos. O crescimento da computação em nuvem fez com que houvesse um aumento no emprego de IDS nesses ambientes [25]. Diante desse cenário, onde a segurança de ambientes de nuvem tem demonstrado um nível de complexidade elevado para ser monitorada manualmente, os IDS se apresentam com uma das soluções de segurança mais vendidas e com tendência a ganhar cada vez mais importância [25].

3.7.1 Sistemas de Detecção de Intrusão baseados em Assinatura

Sistemas de Detecção de Intrusão baseados em Assinatura (SIDS) são um tipo de IDS onde a detecção de intrusão ocorre através da verificação do comportamento analisado com uma base de dados de comportamentos maliciosos conhecidos. Esses comportamentos contém padrões e características de ataques conhecidos e também são conhecidos como assinaturas de ataques. Em SIDS métodos de detecção são utilizados para verificar se um conjunto de dados analisado coincide com alguma das assinaturas de ataques existentes em um banco de assinaturas, e caso isso ocorra, um alerta de detecção é então gerado [10].

SIDS são bastante precisos para detectar ataques já conhecidos, contudo não são capazes de identificar ameaças cibernéticas novas, desconhecidas ou polimórficas [23]. Na ocorrência de um ataque onde a verificação de assinatura não resultar em nenhuma correspondência com a base de assinaturas de ataques conhecidos, nenhum alerta será gerado e o ataque passará desapercebido. Desse modo, SIDS são frequentemente criticados por ter uma alta Taxa de Falso Negativos (FNR), e a quantidade cada vez maior de ataques de *zero-day* tem tornado essa abordagem gradualmente obsoleta [20]. Por outro lado, a Taxa de Falso Positivos (FPR) dos SIDS costuma ser relativamente baixa. Um exemplo de SIDS popularmente conhecido é o Snort. Além de realizar a detecção de intrusões, o Snort também atua como um IPS (Sistema de Prevenção de Intrusão).

Para contornar as limitações ligadas aos SIDS, uma alternativa é combinar o uso de SIDS com AIDS, de maneira que os métodos de detecção de ambas as abordagens possam trabalhar colaborativamente [8]. Outra importante iniciativa é manter um procedimento sistemático de revisão das configurações do IDS, mantendo-o atualizado acerca das novas definições de assinaturas de ataques e alinhado com as políticas de segurança corporativas.

3.7.2 Sistemas de Detecção de Intrusão baseados em Anomalia

Sistemas de Detecção de Intrusão baseados em Anomalia (AIDS) têm despertado interesse de muitos pesquisadores devido à sua capacidade de superar as limitações dos SIDS [10]. Diferentemente dos SIDS, um AIDS não utiliza de um banco de dados contendo assinaturas de ataques conhecidos, mas é capaz de detectar comportamentos anômalos que se desviam de um padrão considerado normal, identificando-os como possíveis intrusões. Assim, a detecção de anomalias pode ser definida como o processo de encontrar padrões em um *dataset* cujo comportamento é anormal ante ao esperado, e são então denominados anomalias ou *outliers* [62].

A detecção baseada em anomalia se fundamenta no pressuposto de que atividades intrusivas geralmente não se parecem com o padrão usual de comportamento de usuários legítimos [8]. Dessa forma, um AIDS é capaz de identificar novos ataques jamais ocorridos antes. A Figura 3.8 ilustra o modelo de detecção de intrusão baseado em anomalia.

Dentre os benefícios obtidos pela utilização de um AIDS, está a capacidade de identificar atividades internas maliciosas [10]. Ou seja, na ocorrência de um ataque onde credenciais de um usuário interno sejam utilizadas para atividades atípicas não previstas pelo sistema de detecção, um alerta poderá ser gerado [10]. Todavia, a principal desvantagem dos AIDS é decorrente da sua acurácia, que normalmente sofre com altas

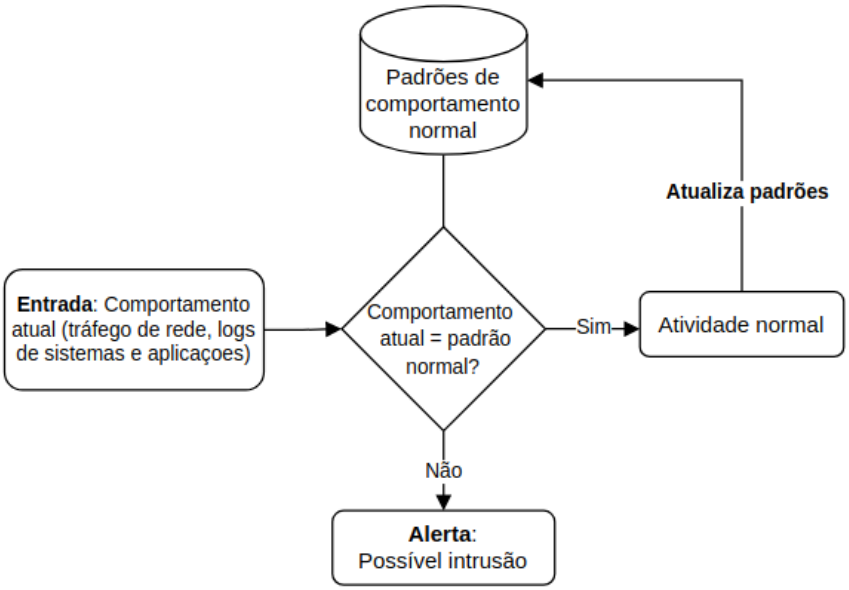

Figura 3.8: Modelo de detecção de intrusão baseado em anomalia [8].

Taxas de Alarme Falso (FA) [23].

O desenvolvimento de um AIDS normalmente é realizado em duas fases: treinamento e teste [10]. Na fase de treinamento, o comportamento considerado normal de usuários, sistemas e outros dispositivos é utilizado para construir um perfil de comportamento normal. Já na fase de teste, dados de comportamentos diferentes dos utilizados na fase de treinamento são utilizados para avaliar a eficácia do sistema de detecção e realizar novos treinamentos, caso necessário. Dados que caracterizam ataques e intrusões também são utilizados para testar o sistema de detecção na fase de teste. Como esses comportamentos diferem do perfil normal construído, devem então ser detectados como anômalos pelo AIDS. O conjunto de dados contendo comportamentos usados para treinamento e teste dos sistemas de detecção de intrusão são conhecidos como *datasets*.

Um fator que deve ser observado com cautela ao se criar um *dataset* de comportamento normal para treinamento de um AIDS é a existência de ataques que possam estar erroneamente inseridos na massa de dados considerados normais. Caso isso ocorra, o mecanismo de detecção treinado poderá considerar um padrão de ataque contido no *dataset* de treinamento como comportamento normal no momento em que ele vier a ocorrer. Da mesma forma, é importante que o *dataset* utilizado para treinamento tenha um volume de dados com boa representatividade do padrão de acessos normais para que o mecanismo de detecção seja capaz de generalizar o comportamento considerado normal. Caso o *dataset* utilizado para treinamento seja insuficiente, é provável que um acesso legítimo seja classificado como anômalo pelo mecanismo de detecção, gerando um falso positivo.

De acordo com [10], métodos de detecção de anomalia podem ser classificados em três grupos principais:

- Baseados em estatística
- Baseados em conhecimento

- Baseados em *machine learning*.

Métodos de detecção de anomalia baseados em estatística constroem um modelo contendo um perfil de comportamento normal, e então detecta eventos de baixa probabilidade e os identifica como sendo anômalos [10]. A detecção de anomalia baseada em conhecimento cria um modelo contendo um perfil normal baseado no conhecimento de um especialista, que define um conjunto de regras que identificam uma atividade considerada normal [10]. Já a detecção de anomalia através de *machine learning* faz uso de algoritmos que constroem modelos que contém um conjunto de regras, métodos e funções complexas que podem ser usadas para encontrar padrões de dados, além de reconhecer ou prever comportamentos [63]. Algoritmos de *machine learning* tem sido amplamente utilizados para detecção de anomalias e empregados nos sistemas de detecção de intrusão modernos. Isto posto, o método de detecção de anomalias através de *machine learning* receberá maior enfoque nesse trabalho.

A Tabela 3.3 traz um comparativo entre as vantagens e desvantagens dos métodos de detecção baseados em assinatura (SIDS) e anomalia (AIDS).

Tabela 3.3: Comparativo entre os métodos de detecção por assinatura e anomalia [8].

		Vantagens	**Desvantagens**
Método de detecção	**SIDS**	• Muito efetivo em identificar intrusões com o mínimo de alarmes falsos (FA). • Indentifica intrusões prontamente. • Superior para detectar ataques conhecidos. • Projeto simples.	• Precisa ser atualizado frequentemente com novas assinaturas. • Projetado para detectar ataques para assinaturas conhecidas. • Qualquer variância na assinatura pode impedir a detecção do ataque. • Incapaz de detectar ataques de zero-day. • Não adequados para detecção de ataques de múltiplas etapas. • Pouco entendimento dos detalhes do ataque.
	AIDS	• Pode ser usado para detectar novos ataques. • Pode ser usado para criar assinaturas de intrusão.	• Não consegue lidar com pacotes criptografados. • Alta taxa de falso positivos (FPR). • Difícil construir um perfil normal para sistema muito dinâmico. • Alertas não classificados. • Precisa de treinamento inicial.

3.7.2.1 Sistemas de Detecção de Intrusão baseados em Host

Um Sistema de Detecção de Intrusão baseado em Host (HIDS) monitora a atividade individual de *hosts* para detectar qualquer atividade não autorizada [24]. HIDS monitoram *logs* do SO, *logs* de aplicação, *system calls*, e outros eventos buscando encontrar padrões e extrair informações para detectar um comportamento ilegal [24]. Por serem instalados como um *software* no sistema a ser monitorado, os HIDS não exigem nenhum *hardware* adicional para seu funcionamento, e conseguem obter informações detalhadas e boa visibilidade sobre os eventos ocorridos no sistema. Por outro lado, existe pouco isolamento entre o HIDS e o sistema monitorado, podendo se tornar alvo de ataque para impedir o seu funcionando [23].

O contexto dessa pesquisa envolve a detecção de intrusão em *clusters* de orquestração de contêineres. Devido às características desses ambientes e do funcionamento dos HIDS, uma abordagem de detecção através do monitoramento do comportamento individual de contêineres em execução nos *hosts* do *cluster*

foi utilizada. Dessa maneira, conforme o escopo de monitoramento do Sistema de Detecção de Intrusão *hosts* específicos do *cluster* de orquestração de contêineres podem ser monitorados por um HIDS instalado localmente em cada nó.

HIDS baseados em *system calls* têm ganhado atenção nos últimos vinte anos devido ao aumento crescente de ataques focados em servidores Linux e têm sido desenvolvidos para detecção de intrusão em *hosts* virtuais e plataformas embarcadas como os *smartphones* [20]. *System calls* são um recurso utilizado pelos processos em execução para comunicar com o *kernel* do SO. Assim, enquanto uma aplicação está em execução em um sistema, eventualmente fará uso de *system calls* para seu funcionamento. Uma funcionalidade específica de uma aplicação implica em um conjunto de *system calls* necessárias para solicitar recursos ao *kernel* do SO. Desse modo, um perfil de utilização normal de uma aplicação poderá ser caracterizado pelo conjunto de *system calls* invocadas durante esse uso. Em sistemas baseados em Unix, todas as aplicações que necessitam de recursos do sistema devem fazer uso de *system calls*, sendo a razão pelo qual HIDS baseados em *system calls* obtém a melhor granularidade de dados [21]. Enquanto HIDS baseados em assinatura detectam atividades anômalas no sistema através de um banco de dados com assinaturas de ataques, HIDS baseados em anomalia de *system calls* inicialmente constroem um perfil de comportamento normal de uma aplicação a partir da utilização de *system calls* e caso processos dessa mesma aplicação futuramente executados não se enquadrem nesse perfil construído, serão considerados como intrusivos [31]. Desse modo, a maioria dos HIDS propostos na literatura são baseados em anomalia [31].

Diversos estudos do estado da arte têm empregado técnicas de *machine learning* para detecção de anomalias e têm apresentado resultados promissores para esse campo de pesquisa. Em visto disso, HIDS podem fazer uso de diferentes técnicas de mineração de dados e *machine learning* como Redes Neurais Artificiais (ANN) para analisar informações do sistema monitorado e identificar intrusões [20].

Conforme [20], uma tendência para o aprimoramento da eficiência de HIDS baseados em *system calls* envolve a utilização da computação em nuvem e ferramentas de *Big Data*. Ainda de acordo com [20], pesquisadores têm trabalhado para construir um HIDS de tempo real e que seja adequado para uso em plataformas de computação em nuvem e *data centers*.

3.8 MACHINE LEARNING

Aprendizado de Máquina ou *Machine Learning* (ML), do termo em inglês, é a ciência que envolve a programação de computadores de maneira que eles possam aprender a partir de dados [9]. Outra definição popular trazida por Samuel Arthur em 1959, define ML como sendo "o campo de estudo que dá aos computadores a capacidade de aprender sem que eles sejam explicitamente programados".

Para contextualização histórica, o filtro *antispam* utilizado em correios eletrônicos é considerado como a primeira implementação amplamente difundida de *machine learning* [9]. Desde então, técnicas de *machine learning* evoluíram à medida que os benefícios de sua utilização nas mais diversas áreas foram sendo alcançados. Outras aplicações onde o ML pode ser útil envolvem: problemas que exigem soluções compostas de numerosas regras e necessidade de ajustes finos, problemas cuja abordagem tradicional para solução é incapaz de trazer bons resultados, ambientes flutuantes, além de problemas complexos com grandes

quantidades de dados [9]. As Figuras 3.9 e 3.10 detalham respectivamente, a comparação entre o método tradicional e a utilização de ML para solucionar o problema de classificação de *e-mails* como *spam* ou não.

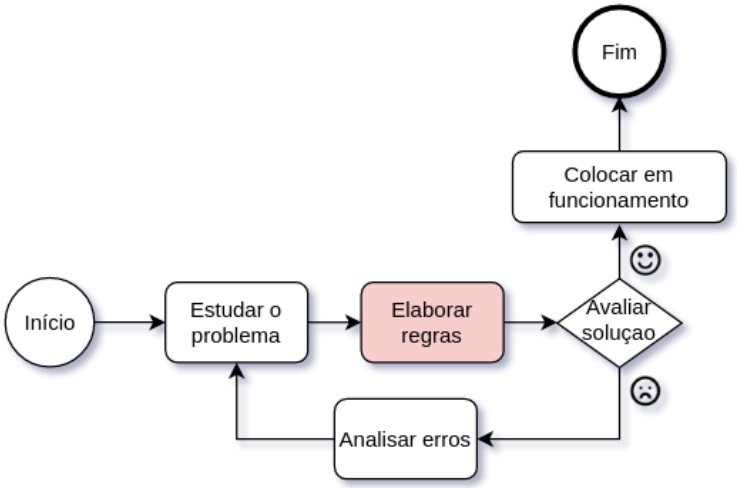

Figura 3.9: Abordagem tradicional usada para solucionar o problema de detecção de *spam* [9].

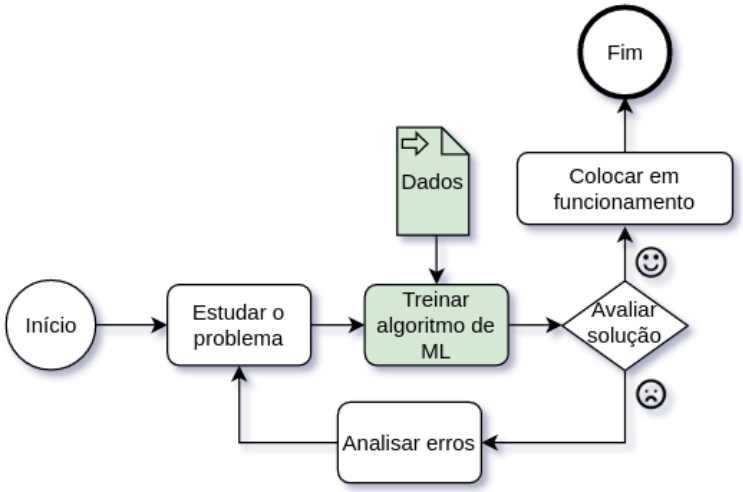

Figura 3.10: Abordagem usando ML para solucionar o problema de detecção de *spam* [9].

No contexto da segurança cibernética, técnicas de ML têm sido aplicadas extensivamente em Sistemas de Detecção de Intrusão baseados em Anomalia e a quantidade desses sistemas que utilizam essas técnicas têm aumentado nos últimos anos [10]. Desse modo, através do uso de ML busca-se construir um IDS com acurácia aprimorada e com menos dependência de conhecimento humano [10]. A figura 3.11 apresenta diferentes técnicas de ML empregadas em AIDS.

Métodos de ML podem ser classificados de acordo com os seguintes critérios, que podem ser combinados entre si [9]:

- Quanto à necessidade de supervisão humana durante o treinamento: supervisionado, semi-supervisionado, não-supervisionado ou Aprendizado por Reforço (*Reinforcement Learning*);
- Quanto à capacidade de aprendizado incremental em tempo de execução: *online* ou aprendizado em lote;

- Se o funcionamento se dá através da comparação de novos dados com dados já conhecidos ou através da detecção de padrões por um modelo preditivo: aprendizado baseado em instância ou aprendizado baseado em modelo.

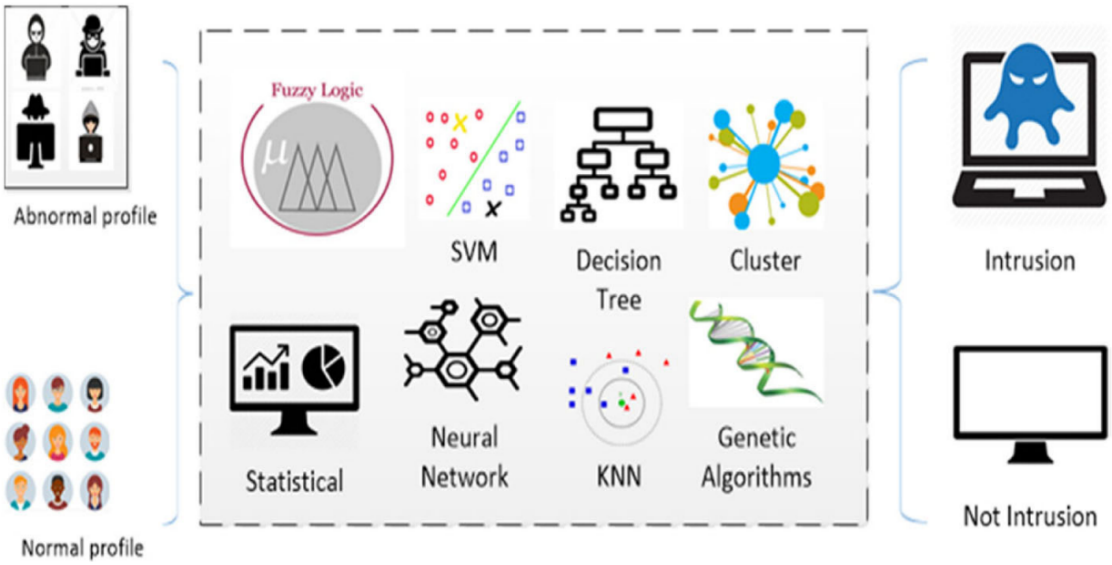

Figura 3.11: Técnicas de ML utilizadas para detecção de anomalias [10]

A implementação de técnicas de ML geralmente segue uma abordagem compreendendo uma fase de treinamento e outra de teste [64]. Para tanto, as seguintes etapas costumam ser seguidas [64]:

1. Identificar os atributos (*features*) e as classes a partir dos dados de treinamento;
2. Identificar um subconjunto de *features* necessárias para a classificação (redução da dimensionalidade);
3. Realizar o treinamento do modelo utilizando dados de treinamento;
4. Utilizar o modelo treinado para classificar dados desconhecidos.

De acordo com [61], o ML é uma das abordagens mais populares para detectar intrusões, e anomalias podem ser detectadas através de vários algoritmos. No relatório de segurança da empresa SANS [54], afirma-se que o nível de confiança depositado nas ferramentas de detecção e no algoritmo de ML em uso é crítico. Dessa forma, um algoritmo mal configurado pode levar ao pior de dois mundos: altas taxas de falsos positivos (FPR) e também de falsos negativos (FNR), simultaneamente [54]. Ademais, no relatório de segurança [53], publicado pela empresa Sophos no ano de 2022, o ano de 2021 marcou a conclusão de uma era de mudança de paradigma dentro da indústria de segurança, quando as técnicas de ML foram reconhecidas como um fator indispensável nos *pipelines* de detecção modernos. Nos anos vindouros, o fato de um fabricante utilizar ML em seus produtos de segurança não será mais um diferencial, mas um requisito [53].

Diferentes métricas podem ser utilizadas para avaliar o desempenho dos algoritmos de ML. Autores como [11] e [20] afirmam que a falta de padronização dos resultados de desempenho em muitos estudos

desenvolvidos na área acaba provocando uma dificuldade de comparação entre os resultados dos trabalhos. Diante do exposto, seria benéfico para os pesquisadores se houvesse uma convergência na forma de apresentação dos resultados das diferentes pesquisas. Ao se analisar os resultados de uma pesquisa, é imprescindível observar a metodologia utilizada para condução dos testes, bem como o *dataset* e outros parâmetros de configuração utilizados. Um algoritmo pode apresentar resultados de detecção distintos conforme os parâmetros e modificações realizadas nos *dataset* utilizados.

Em problemas de classificação binária (classe normal ou anomalia) normalmente se utiliza resultados primários da avaliação de algoritmos e outras métricas podem então ser obtidas em função desses resultados. Esses resultados podem ser representados através de uma matriz de confusão, demonstrando a capacidade de predição de classes do algoritmo, conforme a Tabela 3.4.

Tabela 3.4: Matriz de confusão com os resultados de classificação de um algoritmo.

		Valor predito	
		Falso	Verdadeiro
Valor real	Falso	Verdadeiro Negativo (**TN**)	Falso Positivo (**FP**)
	Verdadeiro	Falso Negativo (**FN**)	Verdadeiro Positivo (**TP**)

A partir dos resultados da matriz de confusão da Tabela 3.4, as seguintes métricas podem ser derivadas [64] [11]:

- **Acurácia (Acc)**: Taxa de acerto na classificação de todos os resultados da matriz de confusão.

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.1}$$

- **Precisão (Pre)**: Taxa de itens corretamente preditos como sendo de uma classe sobre o total de itens preditos como sendo desta classe (corretamente ou não).

$$Pre = \frac{TP}{TP + FP} \tag{3.2}$$

- **Taxa de Verdadeiro Positivo (TPR)**, ***Recall* (Rec)** ou **Sensibilidade**: Taxa de itens corretamente preditos como sendo de uma classe sobre todos os itens pertencentes a esta classe.

$$TPR = \frac{TP}{TP + FN} \tag{3.3}$$

- **Valor Predito Negativo (NPV)**: Taxa de itens corretamente preditos como não sendo de uma classe sobre o total de itens preditos como não sendo desta classe (corretamente ou não).

$$NPV = \frac{TN}{TN + FN} \tag{3.4}$$

- **Taxa de Verdadeiro Negativo (TNR)**, **Cobertura (COV)**, ou **Especificidade**: Taxa de itens corretamente preditos como não sendo de uma classe sobre todos os itens que não pertencem a esta classe.

$$TNR = \frac{TN}{TN + FP} \tag{3.5}$$

- **Taxa de Falso Positivo (FPR)** ou **Taxa de Alertas Falsos (FAR)**: Taxa de itens incorretamente preditos como sendo de uma classe sobre todos os itens que não são desta classe.

$$FPR = \frac{FP}{TN + FP} \tag{3.6}$$

- **F1 Score** ou **F-Score**: Média harmônica entre a Precisão (Pre) e a Taxa de Verdadeiro Positivo (TPR).

$$F1Score = 2 * \frac{Pre * TPR}{Pre + TPR} \tag{3.7}$$

Outra métrica utilizada para demonstrar graficamente os resultados de um algoritmo é conhecida como Área Abaixo da Curva (AUC). A AUC é obtida como a soma da área abaixo de uma curva Característica Operacional do Receptor (ROC), em um gráfico que possui a FPR no seu eixo horizontal e a TPR no eixo vertical [11]. Desse modo, é possível observar a relação entre a FPR e a TPR, e a curva ROC em função dessas duas métricas. A Figura 3.12 traz um exemplo de gráfico com uma curva ROC. A AUC corresponderá à área observada abaixo da curva plotada no gráfico.

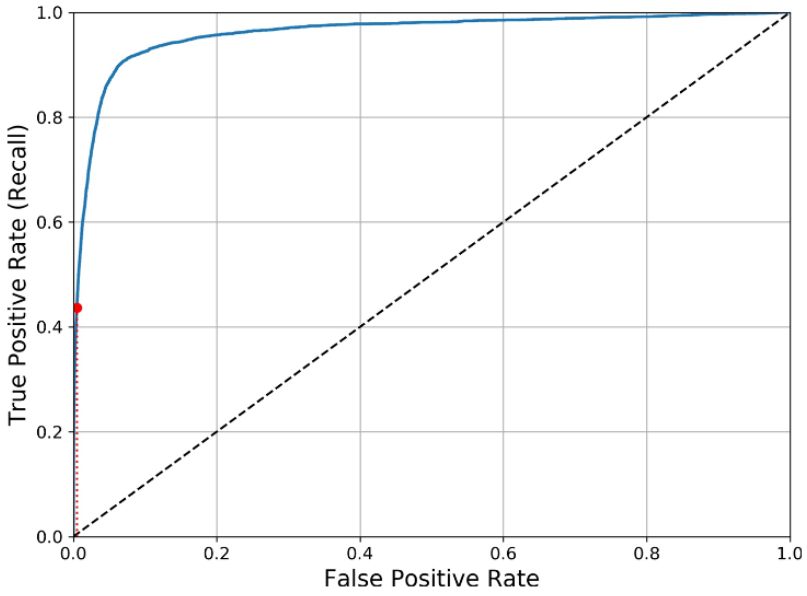

Figura 3.12: Exemplo de gráfico com AUC e Curva ROC [9].

Um algoritmo com desempenho de classificação perfeito teria a AUC igual a 1, o que corresponde à totalidade da área do gráfico [9]. Dessa forma, quanto mais próxima do vértice superior esquerdo do gráfico a curva ROC estiver, melhor o desempenho do algoritmo [10]. O ponto na curva ROC que provê a melhor detecção deve ser escolhido de acordo com o contexto do problema, o *dataset*, e o algoritmo utilizado [64]. No exemplo da Figura 3.12, o ponto na cor vermelha indica uma TPR que corresponde a 43.68%, conforme relação TPR x FPR observada pela curva ROC.

Na pesquisa de revisão bibliográfica realizada por [64], acerca dos métodos de ML utilizados na detecção de intrusão, muitos algoritmos e técnicas de ML foram detalhados. Como conclusão do estudo foi observado que não há como dizer qual o algoritmo mais eficaz para aplicação em segurança cibernética, sendo uma questão ainda não respondida. Ou seja, o desempenho de algoritmos de ML pode variar con-

forme a aplicação em problemas específicos. Portanto, bons resultados na aplicação de um algoritmo em um contexto não são garantia dos mesmos resultados em outro contexto. De maneira geral, a avaliação de desempenho dos métodos de ML deve levar em conta os diversos critérios como: acurácia, complexidade de treinamento, tempo para classificação e interpretabilidade dos resultados [64].

Ainda que a escolha de um algoritmo de ML seja dependente de sua aplicação, o relatório de segurança [53] da empresa Sophos publicado em 2022, afirma que a tecnologia de Redes Neurais de larga escala deve continuar em evidência e com potencial para causar impacto nas áreas da segurança cibernética [53]. Outro estudo desenvolvido por [11], aponta que métodos de Aprendizado Profundo (DL) têm provido novas abordagens para solucionar problemas de segurança cibernética. De acordo com [11], o uso de DL tem apresentado melhorias significantes quando comparado às soluções tradicionais baseadas em detecção por assinatura, bem como quando comparado às soluções baseadas em técnicas de ML clássicas. Além do mais, a utilização de Redes Neurais Recorrentes (RNN) em problemas da área de segurança cibernética tem se tornado popular devido ao fato de os dados relacionados à essa área terem características de problemas de séries temporais [11].

O DL é implementado através de uma Rede Neural Profunda (DNN), que possui múltiplas camadas ocultas que visam realizar o aprendizado em múltiplos níveis, correspondendo a diferentes níveis de abstração [11]. Por outro lado, o Aprendizado Raso é desempenhado por uma Rede Neural Rasa, que tem como característica a existência de apenas uma camada oculta [11]. As Figuras 3.13 e 3.14 ilustram um exemplo de Rede Neural Profunda e Rede Neural Rasa, respectivamente.

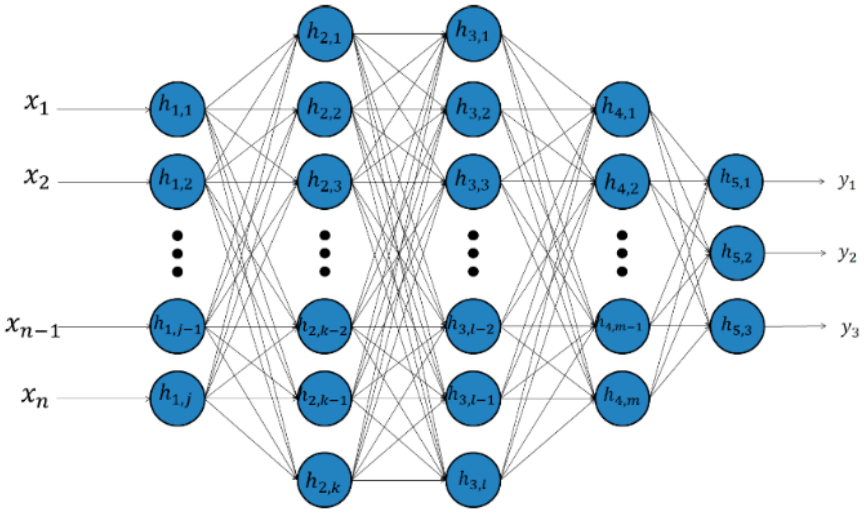

Figura 3.13: Exemplo de Rede Neural Profunda [11].

De acordo com [11], é necessário que métodos de DL sejam testados de maneira que seja possível fazer a comparação dos resultados e métricas de desempenho. Outra abordagem para alcançar melhores resultados de detecção de anomalias é a combinação de diferentes métodos e algoritmos, uma vez que apenas um algoritmo particular pode não alcançar os resultados esperados [62]. Desse modo, abordagens híbridas combinando diferentes técnicas podem ajudar a aprimorar os pontos fracos de métodos específicos e alcançar melhores resultados na detecção de anomalias [62].

Um dos aspectos mais críticos na implementação de técnicas de ML para solução de problemas diz

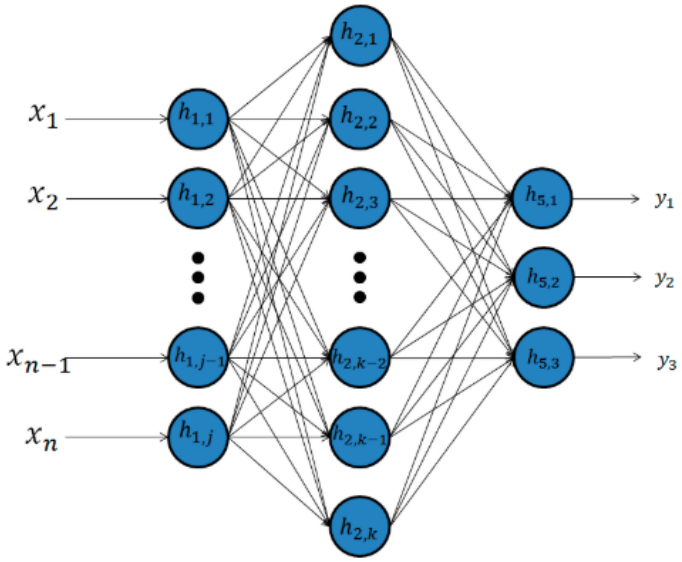

Figura 3.14: Exemplo de Rede Neural Rasa [11].

respeito aos (*datasets*) ou conjunto de dados utilizados. Devido ao fato de os métodos aprenderem através de informações contidas nos dados, é necessário ter uma boa compreensão sobre os dados utilizados [64]. Um importante fator na utilização de *datasets* é que estes estejam balanceados, ou seja, que haja boa representatividade das diversas classes de dados existentes. A quantidade de dados para treinamento, *features* utilizadas e qualidade dos dados também devem ser observadas em um *dataset*. De acordo com [20], a tendência é que à medida que *datasets* relevantes se tornem populares, muitas pesquisas se desenvolvam permitindo a comparação de resultados dos trabalhos e diversos algoritmos de ML também possam ser testados.

Conforme explanado na seção de trabalhos correlatos, as limitações quanto às características dos *datasets* públicos existentes é citada por inúmeros trabalhos recentes e têm influenciado o desenvolvimento de pesquisas na área de ML aplicado à detecção de intrusão. Visando contribuir com pesquisas futuras, este trabalho pretende disponibilizar um *dataset* de *system calls* de uma aplicação conteinerizada em funcionamento em um ambiente de produção corporativo.

3.8.1 Autoencoders e LSTM

Autoencoders (AE) são um tipo de Rede Neural não-supervisionada, onde uma estrutura característica de codificador-decodificador é formada pelas camadas da rede. Esse tipo de Rede Neural, recebe um dado como vetor de entrada e tenta reproduzir o mesmo dado na sua saída [11]. Em uma Rede Neural do tipo AE, o *dataset* utilizado não necessita conter a informação de classes (*labels* ou rótulos), permitindo sua aplicação à detecção de anomalias através de uma abordagem não supervisionada. Ao se utilizar AE para a detecção de anomalias, define-se o problema como um problema de aprendizagem de uma classe, onde os dados de uma classe são considerados como normais, e os dados de qualquer outra classe, como maliciosos [12].

Assim como em uma DNN, uma Rede Neural com *Autoencoders* pode ser configurada estabelecendo

assim, um tipo especial de DNN denominado *Deep Autoencoder* (DAE). Nesse tipo de Rede Neural, as camadas ocultas que compõe a estrutura do codificador têm a função de identificar as *features* mas representativas dos dados recebidos na entrada, descartando outras *features* consideradas menos importantes. Ao final da codificação, se obtém um estado de representação comprimida dos dados. Essa representação com as *features* mais significativas dos dados é então recebida como entrada pelas camadas do decodificador, que tenta recriar o dado original a partir da representação comprimida. A saída da última camada do decodificador é então comparada com o dado original recebido na primeira camada de codificação, calculando-se o erro entre os dois dados e ajustando os pesos da rede neural com o objetivo de minimizar o erro obtido. Ou seja, reduzir a discrepância entre o dado de entrada e saída. Esse tipo de Rede Neural é considerado como incrivelmente versátil, pois podem aprender a partir da codificação comprimida de dados de maneira não supervisionada [11]. A Figura 3.15 ilustra uma Rede Neural do tipo DAE.

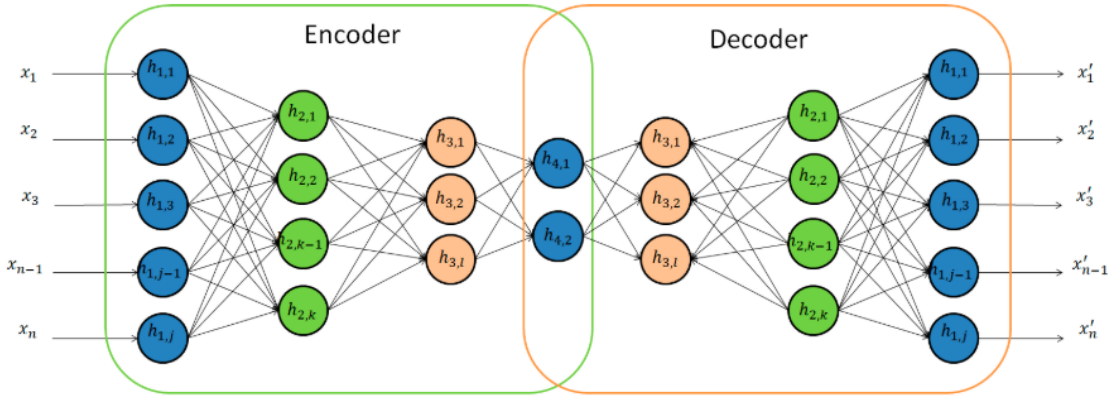

Figura 3.15: Exemplo de uma Rede Neural DAE [11].

Outro tipo de Rede Neural que pode ser utilizado em cenários de detecção de anomalia são as Redes Neurais Recorrentes. Uma RNN possui como característica principal a retroalimentação de informações nas unidades (neurônios) que compõem a rede.

Em uma Rede Neural tradicional, os dados de saída de uma unidade na camada de entrada seguem para as camadas ocultas, e destas, para a camada de saída [65]. Assim, uma Rede Neural tradicional é incapaz de lembrar de dados recebidos como entrada no passado e portanto, não consegue tomar uma decisão baseada em dados históricos [12]. Por outro lado, em uma RNN a saída atual sendo processada por uma unidade leva em consideração o dado recebido como entrada da unidade anterior a ela [65]. Deste modo, as RNNs são especialmente úteis em contextos envolvendo o processamento de informações sequenciais.

As unidades das camadas ocultas de uma RNN são capazes de manter um vetor de estado que contém uma memória de eventos passados em uma sequência, sendo possível ajustar a quantidade de eventos passados possíveis de serem lembrados [11]. A Figura 3.16, detalha um exemplo de RNN com uma camada oculta além de conexões diretas e recursivas.

As redes LSTM (*Long Short-Term Memory*), são um tipo de RNN apresentadas em 1997 por Hochreiter, com o objetivo de aprimorar as RNN existentes à época [66]. Através das redes LSTM é possível lidar com problemas que exigem memória de longo prazo fazendo uso de uma célula de memória que acumula informação e retroalimenta a unidade no passo seguinte [11]. Portas lógicas incluídas dentro de uma unidade LSTM são usadas para determinar quando é necessário lembrar o valor de entrada e quando se deve

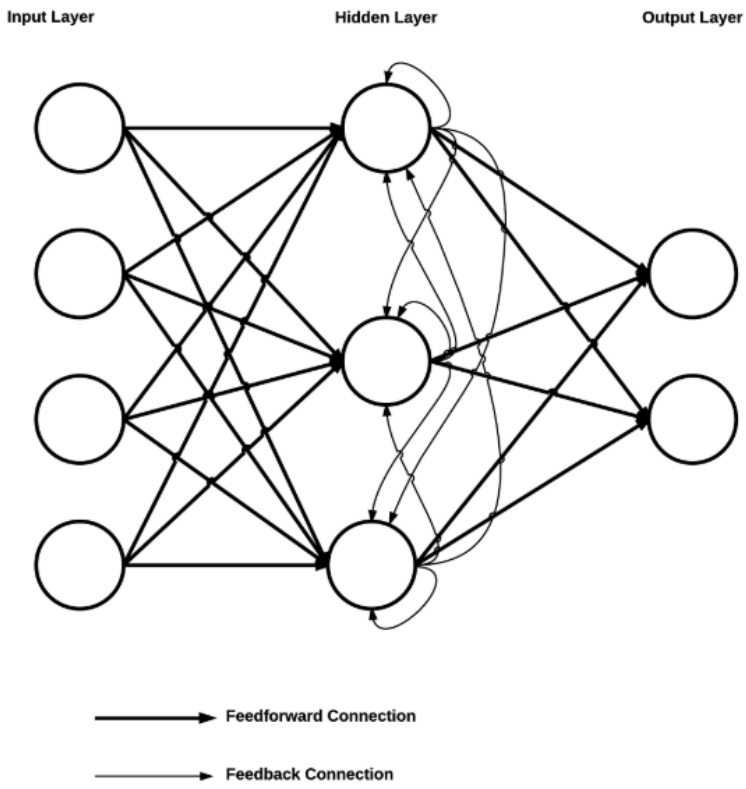

Figura 3.16: Exemplo de uma Rede Neural Recorrente [12].

reter, descartar ou propagar o valor relativo [20]. A Figura 3.17 detalha a estrutura interna de uma unidade LSTM com portas lógicas implementadas pelas funções de ativação *sigmoid* (σ) e tangente hiperbólica (tanh).

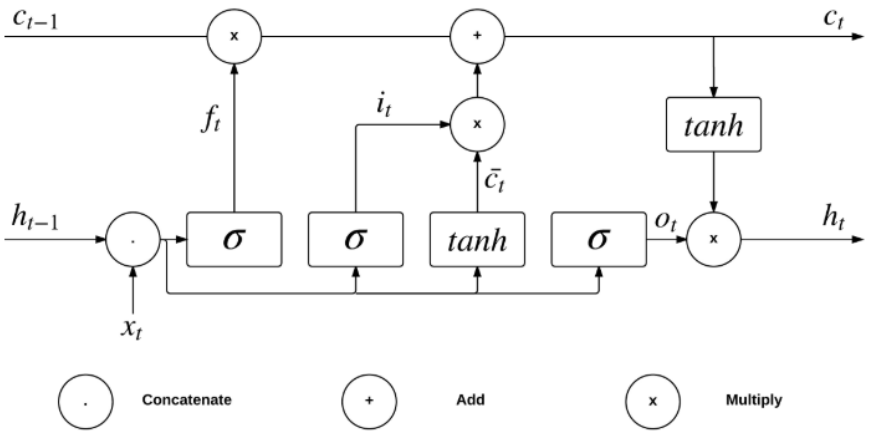

Figura 3.17: Detalhes internos de uma unidade LSTM [12].

A capacidade das redes LSTM de processar dados em série temporal tem comprovada utilidade no contexto dos Sistemas de Detecção de Intrusão, uma vez que os *datasets* de segurança cibernética são frequentemente estruturados como sequências de operações realizadas na linha do tempo [67]. Desse modo, este trabalho fará uso das redes LSTM e *Autoencoder* para implementar o mecanismo de detecção de anomalias em sequências de *system calls* invocadas por aplicações conteinerizadas.

4 FRAMEWORK PROPOSTO

O *framework* proposto é composto por conjunto de ferramentas e processos integrados em uma arquitetura de referência para implementação de um Sistema de Detecção de Intrusão baseado em *Host* em um *cluster* de orquestração de contêineres. Uma possível intrusão ou ataque à um sistema conteinerizado poderá ser identificada através da detecção de anomalias nas *system calls* realizadas pelos contêineres ao *kernel* do Sistema Operacional do *host* da plataforma de contêineres. Desse modo, o *framework* visa o aprimoramento dos Sistemas de Detecção de Intrusão em plataformas de contêineres e foi elaborado com o objetivo de contribuir com a resolução de problemas expostos pelos trabalhos relacionados como: a sobrecarga causada nas plataformas de contêineres pelos HIDS, a ausência de *datasets* atuais, e a pouca ênfase dada à detecção de intrusão em tempo real [13].

As Figuras 4.1, 4.2 e 4.3 apresentam respectivamente os fluxogramas que descrevem o funcionamento do *framework* em três fases de operação: treinamento, teste, e produção, em cinco camadas de processamento: camada 1 (captura de dados), camada 2 (*feature engineering*), camada 3 (indexação e busca), camada 4 (detecção de anomalias) e camada 5 (geração de alertas e análise de dados). Dessa forma, a operação do IDS tem início na fase de treinamento (Fig. 4.1), seguindo-se com a fase de teste (Fig. 4.2) e por fim, com a fase de produção (Fig. 4.3). Eventualmente, o ciclo de operação do IDS poderá ser reinicializado tendo em vista a necessidade de treinamento do modelo com novos dados ou ajustes de parâmetros. Devido ao fato do modelo de *machine learning* utilizado ser classificado como de aprendizado em lote, seu aprendizado é realizado de maneira *offline* [9]. Ou seja, o modelo treinado e em uso na fase de produção é utilizado apenas para detecção e não aprende simultaneamente com os dados analisados. Apesar de o treinamento de um novo modelo ter de ser feito *offline*, esse requisito não impede o funcionamento do IDS enquanto novos modelos são treinados ou testados. Dessa forma, após o treinamento de um modelo com novos parâmetros ou *datasets*, o mesmo poderá ser implantado para funcionamento em produção em substituição a outro modelo em uso. Nesse momento, o funcionamento do IDS deverá ser reiniciado para carregamento do novo modelo e retorno de sua execução na fase de produção, conforme o fluxograma da Figura 4.3.

Na fase de treinamento do *framework*, o objetivo principal é treinar um modelo de *machine learning* apresentando comportamentos benignos de uma aplicação conteinerizada. O modelo treinado com comportamentos normais poderá identificar anomalias através da discrepância entre as *system calls* invocadas na utilização normal e o uso malicioso do sistema. Para tanto, é necessário garantir que nenhum acesso malicioso ou ataque ocorra durante a captura das *system calls*. O processamento dos dados nessa fase, tem início na camada 1 do *framework* e prossegue pelas camadas intermediárias até a camada 4, onde é feito o treinamento do modelo e seu encerramento ocorre quando o modelo atinge níveis satisfatórios de detecção utilizando os dados de treinamento. Além do modelo de *machine learning* treinado que poderá ser utilizado nas fases de teste e produção do *framework*, outro subproduto da fase de treinamento são os *datasets* armazenados no Elasticsearch na camada 3, após tratamento e filtragem realizados na camada 2.

Após o treinamento do modelo de *machine learning*, na fase de testes do *framework*, *datasets* distintos do utilizado na fase de treinamento são usados para avaliação de desempenho da detecção de anomalias do

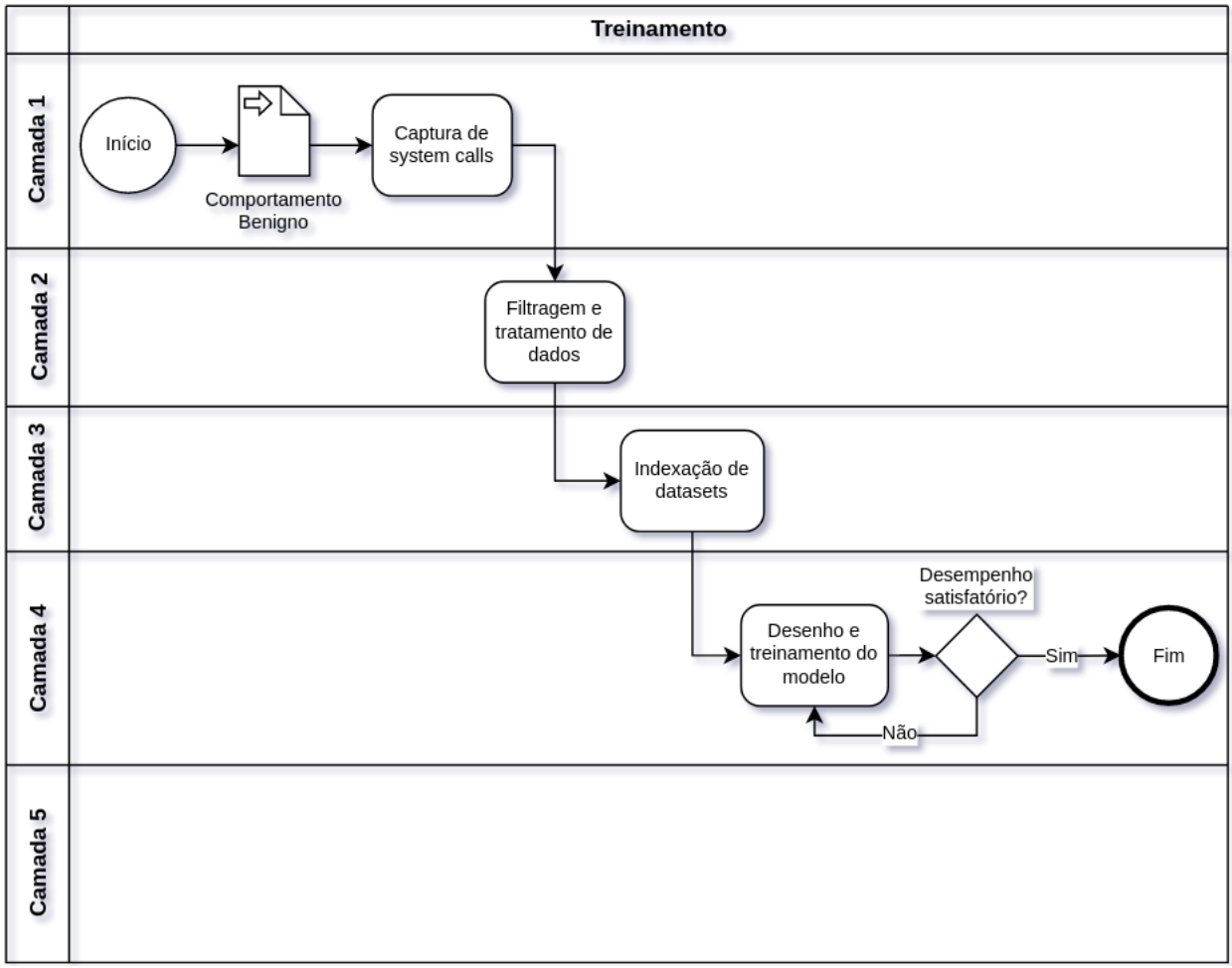

Figura 4.1: Fluxograma descrevendo o funcionamento do *framework* em fase de treinamento e cinco camadas.

modelo treinado. Esses *datasets* devem conter comportamentos benignos ou maliciosos da mesma aplicação conteinerizada. Sua construção deve ser feita de maneira íntegra, garantindo a captura sistemática das *system calls* em comportamentos identificados como normais e/ou anômalos na utilização da aplicação. Nessa fase de operação, o processamento dos dados tem início na camada 1 e permeia até a última camada do *framework*. Diferentemente da fase de treinamento, na camada 4 o modelo de *machine learning* treinado é utilizado apenas para detecção de anomalias nos *datasets* de teste. Ainda na camada 4, métricas de desempenho do modelo podem ser visualizadas, conforme citado na Seção 3.8. Os resultados de detecção do modelo também são armazenados no Elasticsearch e poderão ser consultados e analisados posteriormente. Na camada 5, é possível fazer a análise dos resultados de detecção do modelo e alertas gerados em caso de anomalias detectadas através de uma interface web do Kibana. Caso necessário, conforme o desempenho do modelo, a escolha de um novo valor de *threshold* para detecção de anomalias pelo modelo poderá ser ajustado. Se o desempenho do modelo na detecção de anomalias for insatisfatório, a operação do *framework* poderá voltar para a fase de treinamento para construção de um novo modelo de *machine learning*. Cabe ressaltar que tanto na fase de treinamento quanto na fase de produção do *framework*, uma vez que os *datasets* de treinamento e teste já estejam indexados no Elasticsearch o processamento das camadas 1 e 2 não precisa ser repetido a cada treinamento ou teste de um novo modelo com *datasets* indexados. Nesse cenário, a execução do fluxo de processamento tem início na camada 3.

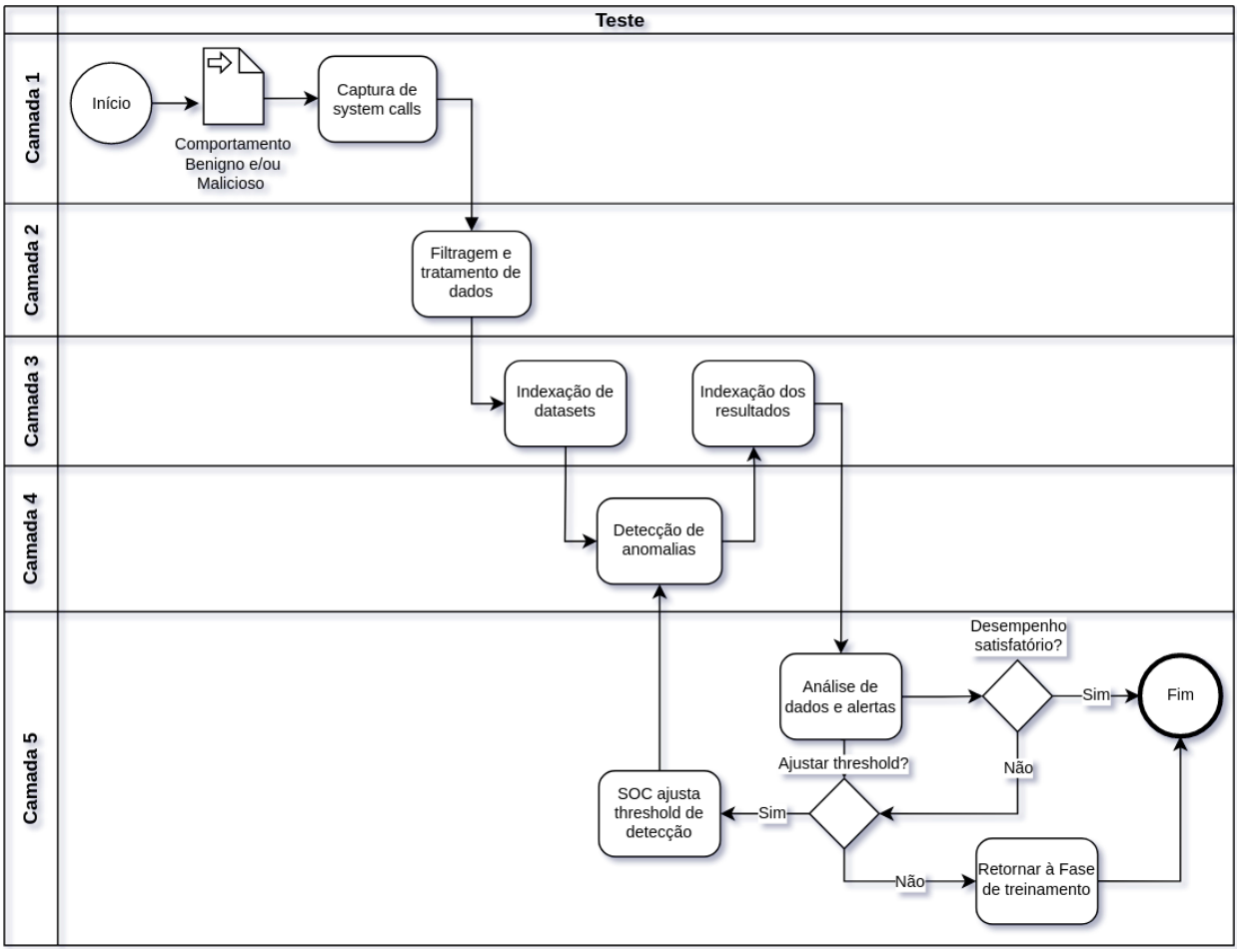

Figura 4.2: Fluxograma descrevendo o funcionamento do *framework* em fase de teste e cinco camadas.

Durante a fase de produção do *framework* o principal objetivo é a detecção de anomalias no comportamento da aplicação alvo utilizando o modelo de *machine learning* treinado e testado nas fases de operação anteriores. A captura das *system calls* ocorre na camada 1 em tempo de execução da aplicação e o processamento dos dados segue o fluxo de tratamento e indexação pelas camadas 2 e 3, respectivamente. A indexação das *system calls* no Elasticsearch é feita em índices que são criados em um intervalo de tempo fixo e que pode ser ajustado. A cada intervalo de tempo definido, um novo índice é criado com um conjunto de *system calls* executadas naquele período. Desse modo, a janela de tempo representada por um índice insere um atraso entre a ocorrência da *system call* e a análise de anomalias pelo modelo, sendo um importante fator a ser observado na configuração do *framework*. A frequência de criação de novos índices no Elasticsearch poderá ser a cada dia, hora ou mesmo frações menores de tempo, conforme a tolerância no tempo de detecção aceito pelo IDS. Após a criação de um índice, ele será utilizado como entrada para análise de anomalias pelo modelo de *machine learning* na camada 4. A leitura dos índices é realizada individualmente e de maneira sequencial, conforme horário de criação de cada índice. Na camada 5, a equipe do SOC poderá acompanhar os resultados de detecção pelo IDS em um *dashboard* criado no Kibana. Na ocorrência de uma eventual anomalia, o SOC poderá realizar uma análise mais aprofundada do evento correlacionando-o com outras fontes de informação como um SIEM, por exemplo. Além disso, o *threshold* de detecção poderá ser ajustado para permitir uma maior ou menor sensibilidade na detecção de anomalias pelo IDS. Ainda assim, caso o desempenho do IDS seja insatisfatório, a operação do *framework*

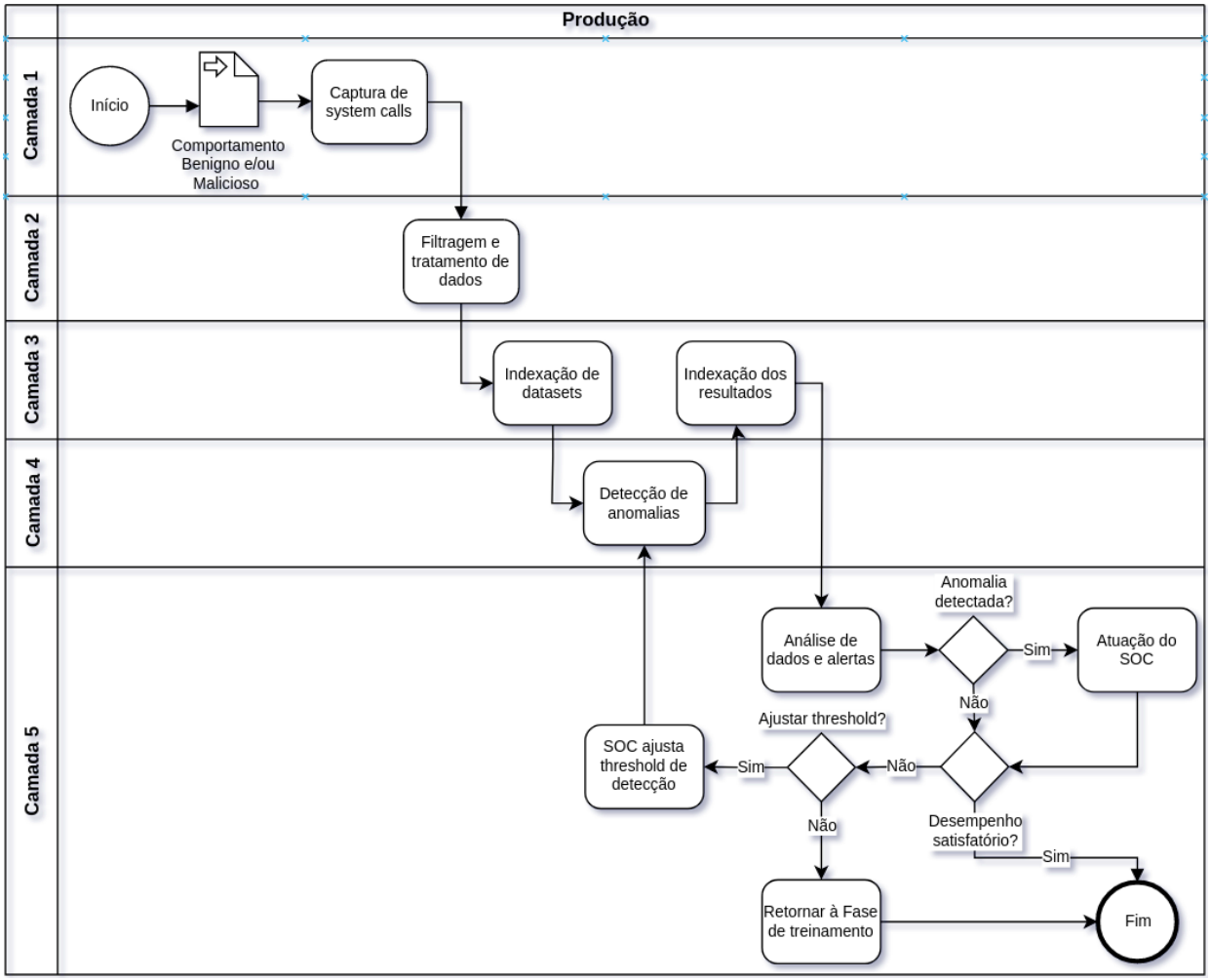

Figura 4.3: Fluxograma descrevendo o funcionamento do *framework* em fase de produção e cinco camadas.

poderá retornar para as fases anteriores para aprimoramento do modelo de *machine learning* ou eventuais ajustes em outros componentes.

4.1 ARQUITETURA DE REFERÊNCIA

A arquitetura desenvolvida apresenta um HIDS com componentes distribuídos, viabilizando a elasticidade do sistema conforme a necessidade [13]. A segregação da arquitetura em diferentes camadas permite uma identificação mais clara de funções e ferramentas envolvidas no processamento dos dados e atende ao propósito de redução da sobrecarga de processamento na plataforma de contêineres. Desse modo, o fluxo de dados do sistema de detecção permeia cinco camadas, onde ferramentas gratuitas são utilizadas para propósitos específicos, conforme a Figura 4.4 [13]. De acordo com os fluxogramas ilustrados nas Figuras 4.1, 4.2 e 4.3, diferentes tarefas podem ocorrer nas camadas 3, 4, e 5, conforme a fase de operação do *framework*. Contudo, em essência, as principais funções executadas em cada camada são comuns a todas as fases de operação. Nas sessões seguintes serão fornecidos detalhes de cada camada da arquitetura proposta.

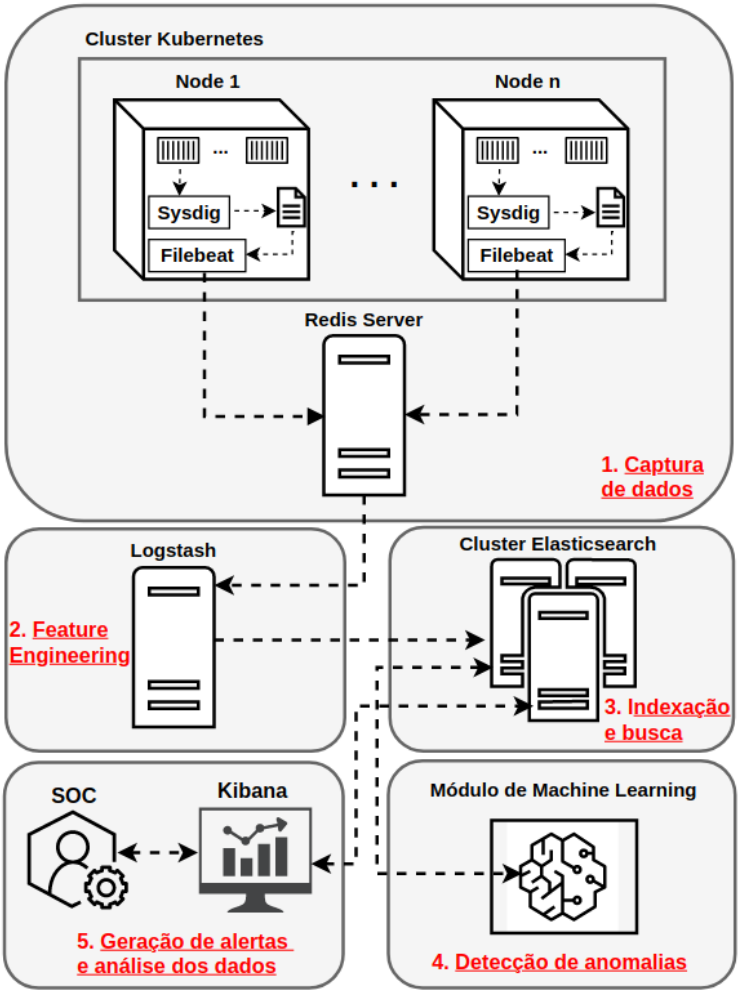

Figura 4.4: Arquitetura proposta com cinco camadas e ferramentas [13].

4.1.1 Camada 1 - Captura de dados

O fluxo de processamento na camada de captura de dados tem início através de um agente localizado em nos nós do *cluster* Kubernetes responsáveis pela execução dos contêineres de aplicação. Através de configurações específicas a nível de *cluster* também é possível fidelizar a execução de contêineres de determinadas aplicações em nós selecionados, evitando-se que o agente tenha de ser instalado em todos os nós do *cluster*, mas apenas naqueles que executarão a aplicação monitorada. Esse agente faz a coleta das *system calls* dos contêineres desejados através da ferramenta Sysdig e as grava em um arquivo local no sistema de arquivos do nó, evitando a dependência de serviços externos na camada de captura de dados [13]. Ao se fazer a captura de *system calls* de um contêiner através do Sysdig, é possível especificar quais atributos de cada *system call* invocada serão coletados através de filtros. Esses atributos serão posteriormente utilizados para construção das *features* que irão compor o *dataset* de *system calls*. O Sysdig provê uma lista de filtros disponíveis para utilização em sua documentação [68] e para o escopo deste trabalho, os filtros elencados na Tabela 4.1 foram utilizados para captura das *system calls* dos contêineres.

À medida que as *system calls* são gravadas em arquivo, um serviço provido pela ferramenta Filebeat faz a leitura de cada nova linha adicionada em arquivo e as envia para o Redis, um banco de dados em

Tabela 4.1: Atributos de *system calls* coletados pelo Sysdig.

Filtro	Descrição
evt.num	Número do evento
evt.time.iso8601	Timestamp do evento no formato iso8601 (incluindo nano segundos e fuso horário)
proc.name	Nome do executável que gerou o evento
evt.dir	Direção do evento ('>' para eventos de entrada e '<' para eventos de saída)
evt.category	Categoria do evento (ex: 'file' para operações de abertura e fechamento de arquivos, 'net' para operações de rede, etc.)
syscall.type	Nome da *system call* invocada
evt.latency	Delta entre o horário de entrada e saída de um evento em nano segundos
evt.rawres	Valor de retorno do evento
evt.info	Retorna os argumentos do evento e informações adicionais

memória externo ao *cluster* [13]. O Filebeat permite que múltiplos arquivos contendo *system calls* possam ser lidos simultaneamente, possibilitando assim o monitoramento de diversos contêineres em execução ao mesmo tempo. Para prover segurança na comunicação entre o Filebeat e o Redis, uma senha é exigida para autenticação no Redis antes do recebimento de dados por esse serviço. Essa senha deverá ser parametrizada na configuração do Filebeat. O Apêndice I.1 contém detalhes da configuração do Filebeat para leitura de múltiplos arquivos em um diretório e envio para o Redis.

Caso necessário, a captura das *system calls* de um determinado contêiner poderá ser interrompida a qualquer momento, e em caso de reinicialização da aplicação monitorada por eventuais necessidades (*deploy* de nova versão, reinicialização, etc.), um novo processo deve ser disparado pelo agente para monitoramento do novo contêiner. A Figura 4.5 ilustra o conteúdo de um arquivo criado em um nó do *cluster* Kubernetes para armazenamento das *system calls* de um contêiner em execução capturadas pelo Sysdig.

Ainda na camada de captura de dados, o Redis age como um *cache*, dando resiliência e alto desempenho ao fluxo de *system calls* capturadas em tempo real dos contêineres em execução [13]. Diferentes filas contendo as *system calls* podem ser criadas no Redis para cada contêiner monitorado. As filas permanecem no Redis até que o Logstash, um serviço da camada 2 da arquitetura, faça a leitura e remoção dos dados (operação POP) das filas. Desse modo, os dados só persistem no Redis em memória até que sejam consumidos pela camada 2, dando lugar para a entrada de novas *system calls*. Cada fila no Redis, contém documentos que representam as *system calls* capturadas em formato json, e podem ser inspecionadas no servidor Redis, conforme exemplificado na Figura 4.6.

Para evitar que essa carga de trabalho seja executada no *cluster* Kubernetes, o Redis foi configurado em um *host* dedicado a esse serviço. Caso necessário, o servidor Redis pode ser escalado horizontalmente para trabalhar como um *cluster*, oferecendo maior desempenho no fluxo dos dados.

4.1.2 Camada 2 - Feature engineering

A camada 2 da arquitetura leva o nome de *feature engineering*, do termo em inglês, pois é onde ocorre um processamento dos dados brutos de *system calls* em *features* (características) que irão compor os *datasets* de entrada para o modelo de *machine learning*. Conforme [20] e [24], a preparação dos dados e a extração de *features* são fatores decisivos e portanto, carecem de mais atenção quando comparados à escolha do melhor modelo de detecção. Desse modo, uma parte crítica em um projeto envolvendo *machine*

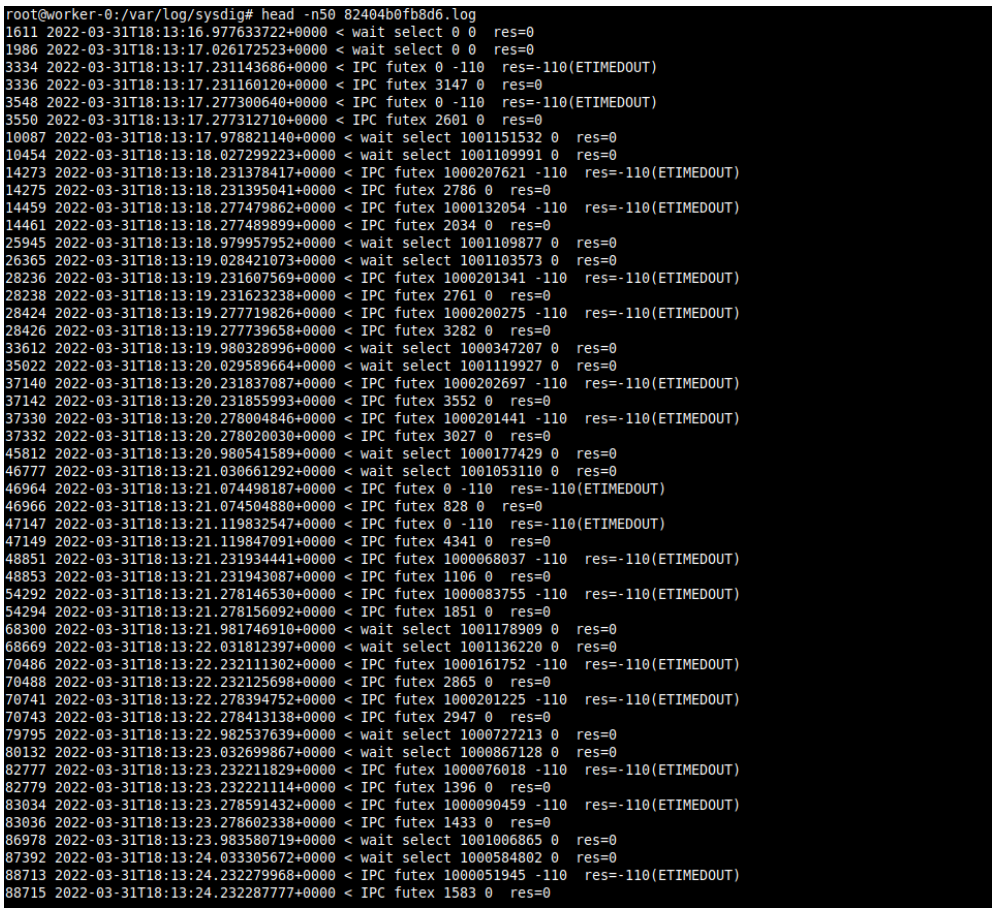

Figura 4.5: Detalhes das *system calls* capturadas de um contêiner em execução pela ferramenta Sysdig [13].

learning é a definição de um bom conjunto de *features* dos dados que serão usados para treinamento do modelo [9]. Esse processo é denominado *feature engineering* e envolve os seguintes passos [9]:

- Seleção de *features* (*Feature selection*): Seleção das *features* mais relevantes dentre as existentes
- Extração de *features* (*Feature extraction*): Combinação de *features* existentes para produção de novas *features* mais úteis
- Criação de novas *features* a partir de novos dados

De acordo com [10], a seleção de *features* é útil para diminuir a dificuldade computacional, eliminar a redundância de dados, melhorar a taxa de detecção das técnicas de *machine learning*, simplificar os dados e reduzir os falsos alarmes. Desse modo, pode-se dizer que a seleção de *features* no *framework* têm início ao se determinar quais serão os filtros utilizados no Sysdig para captura de atributos das *system calls*. Dentre muitos atributos disponíveis para captura, apenas alguns considerados mais relevantes para o escopo de identificação de anomalias no comportamento de aplicações foram selecionados conforme a Tabela 4.1. De toda forma, outros filtros poderão ser utilizados nas capturas feitas pelo Sysdig possibilitando a criação de novas *features* no *dataset*.

O Logstash é a ferramenta utilizada para consumir os dados do Redis e fazer a filtragem e tratamento dos dados brutos das *system calls* [13]. Através do Logstash, é possível manipular e fazer o tratamento

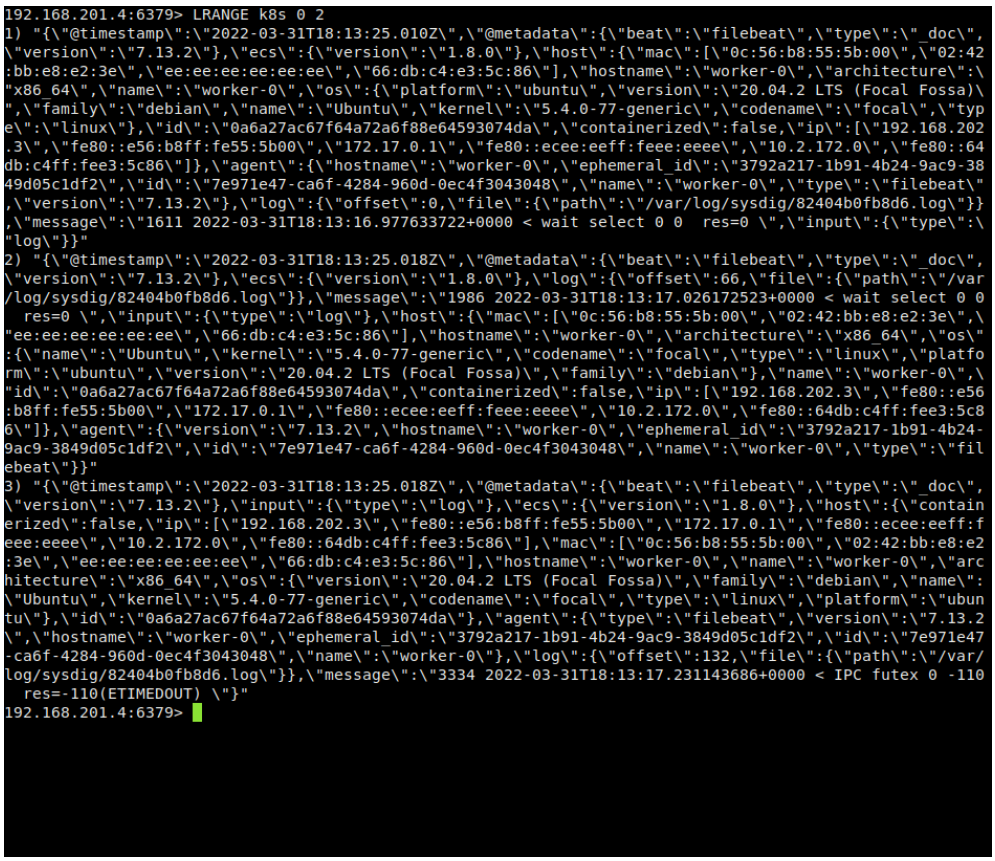

Figura 4.6: Detalhes de *system calls* recebidas em uma fila do Redis [13].

de cada atributo que compõe uma *system call*, convertendo *strings* em valores numéricos, descartando informações desnecessárias e adicionar outras informações caso necessário. Cada *system call* enfileirada no Redis é extraída e tratada individualmente em um *pipeline* de processamento composto de três etapas: entrada (*input*), filtragem (*filter*) e saída (*output*). Para evitar uma carga adicional de processamento no *cluster* Kubernetes, o Logstash foi configurado em um *host* dedicado a esse serviço e sua comunicação com o Redis ocorre por meio do protocolo TCP/IP. Caso necessário, múltiplas instâncias do Logstash podem ser configuradas na camada 2, permitindo a escalabilidade desse serviço e maior capacidade no processamento do fluxo de dados.

Em síntese, na seção de entrada, é onde são definidos as configurações para leitura das filas do Redis. A mesma senha utilizada para autenticação do Filebeat pelo Redis deverá ser configurada no Logstash para a leitura das filas do Redis. Na seção de filtragem são realizadas diversas operações para identificação de padrões através de expressão regulares, conversão de dados textuais em numéricos, remoção de campos, adição de novos atributos a cada evento, bem como outras transformações. Após o tratamento dos dados realizadas na seção de filtragem, na seção de saída, cada evento correspondente à uma *system call* invocada é indexado como um documento no Elasticsearch em índices que podem ser criados dinamicamente conforme o número de documentos indexados ou em intervalos de tempo pré-definidos. Os índices no Elasticsearch são as estruturas de dados que contém os *datasets* com as *system calls* e as suas *features* já codificadas em formato numérico, próprio para o processamento feito pelos algoritmos de *machine learning* [13]. A comunicação entre o Logstash e o Elasticsearch também exige uma autenticação baseada em

usuário e senha que devem ser criados no Elasticsearch e posteriormente configurados na seção de saída do Logstash. O Apêndice I.2 detalha a configuração realizada nas seções em cada uma dessas seções.

Por fim, como resultado do processamento realizado pelo Logstash, cada *system call* representada por um documento no índice do Elasticsearch contém os seguintes atributos, que também podem ser visualizados na Figura 4.8

- args_length – quantidade de caracteres dos argumentos da *system call*
- event_number – número do evento
- systemcall_type – categoria do evento
- @timestamp – horário do evento em formato UTC
- process_name – nome do executável que gerou o evento
- argv – vetor com os argumentos da *system call*
- systemcall – nome da *system call*
- message – evento completo conforme capturado pelo Sysdig
- event_direction – direção do evento (0 para eventos de entrada e 1 para eventos de saída)
- latency – delta entre o horário de entrada e saída de um evento em nano segundos
- argc – número de argumentos da *system call*
- timestamp – horário do evento em formato ISO8601
- return_value – valor de retorno do evento

4.1.3 Camada 3 - Indexação e busca

Na camada de indexação e busca, tem-se como principal componente o Elasticsearch. O Elasticsearch é um motor de indexação e busca de alto desempenho [69]. Sua utilização como repositório para armazenamento das *system calls* permite que haja alta disponibilidade dos dados, além de possibilitar pesquisas através de meta-dados úteis para filtragens, agregações, ordenações, etc [13]. Ele provê uma interface API (*Application Programming Interface*) baseada em REST (*Representational State Transfer*) e no protocolo HTTP ((*HyperText Transfer Protocol*)) para comunicação e consulta aos dados, que são armazenados no formato JSON (*JavaScript Object Notation*) [69]. Essas características facilitam sua integração com outras ferramentas para gravação e leitura de dados.

O Elasticsearch pode ser estruturado como um *cluster* formado por um ou mais nós instalados em *hosts* dedicados a esse serviço. Nos nós do *cluster* Elasticsearch os dados são armazenados de maneira fragmentada, e replicada, a depender do número de nós do *cluster*. A Figura 4.7 ilustra uma consulta de *status* de um *cluster* Elasticsearch através de sua API, e detalhes da configuração de um *cluster* de Elasticsearch composto por um único nó podem ser encontrados no Apêndice I.3.

Além de conter os índices criados pelo Logstash representando os *datasets*, o Elasticsearch integra com o módulo de *machine learning* da camada 4, com a ferramenta Kibana da camada 5, e também é utilizado para armazenar os resultados das análises de anomalias. Desse modo, sua operação pode dar através do provimento de dados das consultas aos *datasets* ou de resultados de detecção de anomalia, ou simultaneamente indexando novos *datasets* e resultados de detecção de anomalia pelo módulo de *machine learning*.

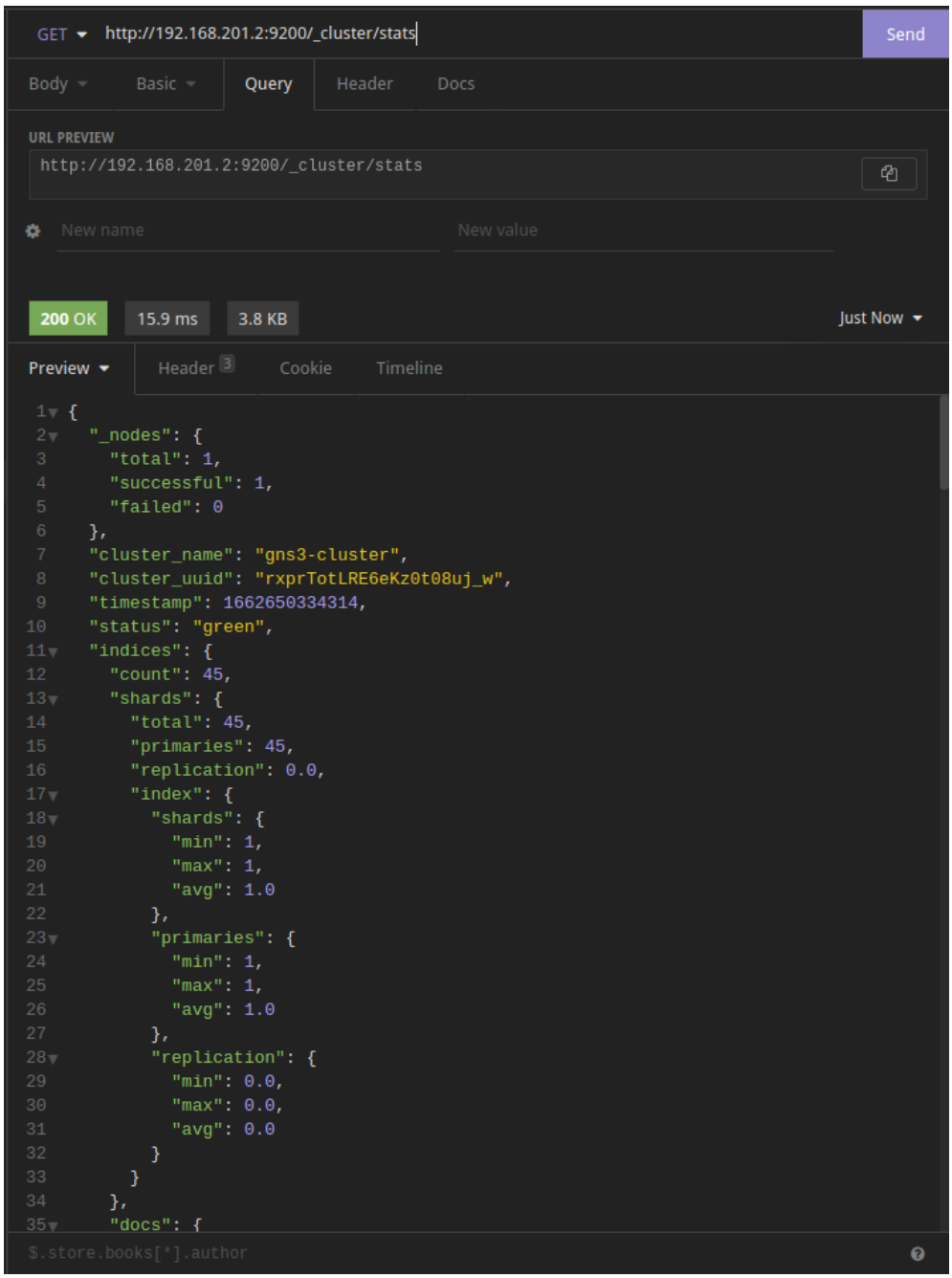

Figura 4.7: Consulta do *status* de um *cluster* Elasticsearch realizada via API.

Para cada índice representando um *dataset* no fluxo de processamento descrito na Figura **??**, após sua análise, um novo índice é criado no Elasticsearch com o escore de anomalia correspondente às janelas de *system calls* analisadas pelo módulo de *machine learning* [13]. As janelas de *system calls* consistem em uma estrutura da dados que armazenam sequências de *system calls* de múltiplos tamanhos e é uma abordagem comum para análise de *system calls* [13].

A Figura 4.8 ilustra o resultado de uma consulta feita a um índice existente no Elasticsearch através de sua API. Para correto armazenamento das informações no Elasticsearch é preciso que um *template* seja criado definindo o tipo correto de cada atributo do índice, conforme sua documentação [69]. Para o índice

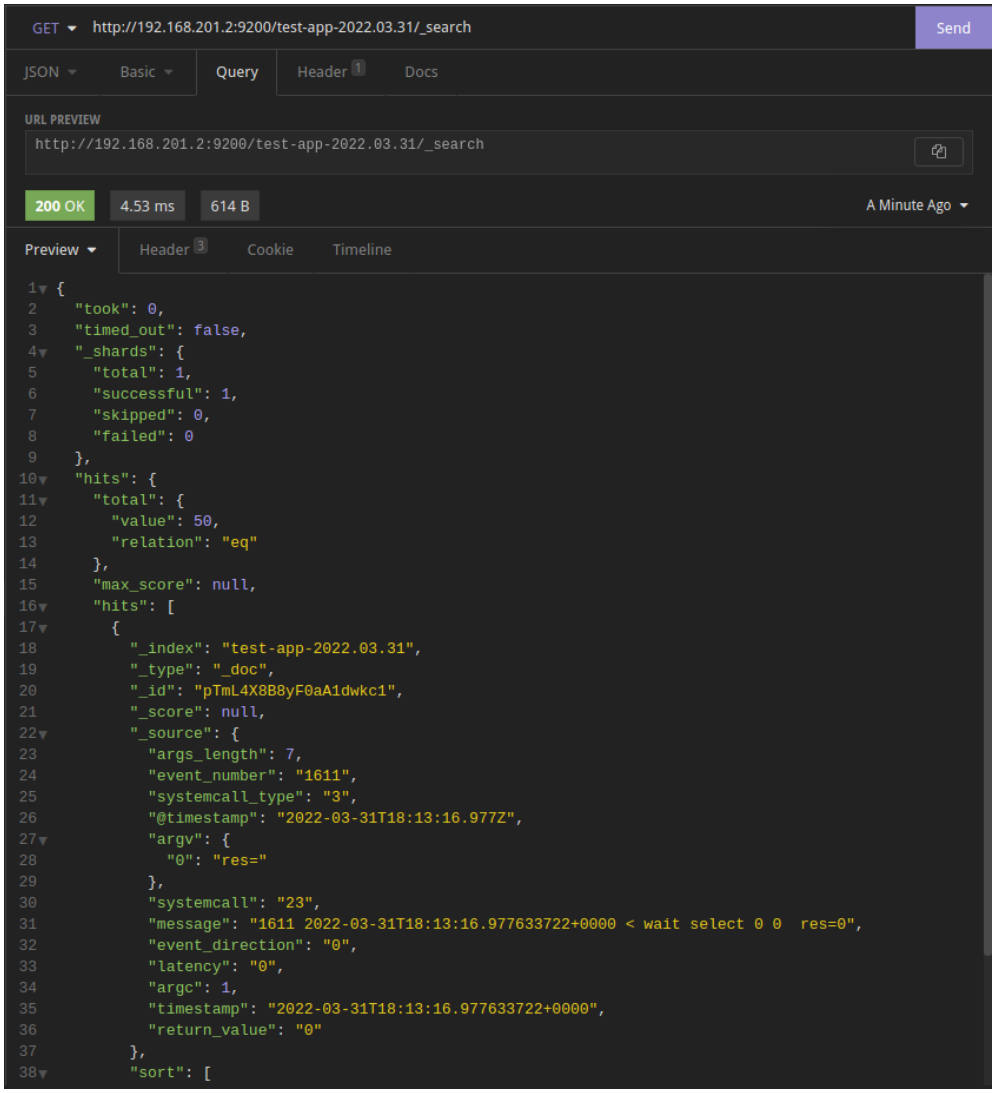

Figura 4.8: Índice criado no Elasticsearch representando o *dataset* com as *system calls* [13].

de exemplo da Figura 4.8, o *template* do Apêndice I.4 foi utilizado para formatação dos valores dos seus campos ou atributos, dentre outras configurações. Uma configuração definida no *template* e aplicada ao índice de exemplo, é a ordenação dos documentos pelo campo "*timestamp*". Dessa forma, os documentos no índice do Elasticsearch ficam armazenados de maneira cronológica conforme o horário de ocorrência do evento capturado pelo Sysdig.

O Elasticsearch é uma ferramenta desenvolvida pela mesma empresa responsável pelo Filebeat, Logstash e Kibana, portanto, a integração entre essas ferramentas é nativa. Desse modo, a visualização dos dados armazenados em índices no Elasticsearch pode ser feita através de uma interface web no Kibana. Diversas operações podem ser feitas no Elasticsearch para manipulação e armazenamento dos *datasets* em conjunto com o Kibana. Mais detalhes serão fornecidos na Seção 4.1.5.

4.1.4 Camada 4 - Detecção de anomalias

Na camada de detecção de anomalias, o módulo de *machine learning* representado na arquitetura da Figura 4.4 pode executar diversos algoritmos do estado da arte para detecção de anomalias [13]. Contudo, conforme fundamentação teórica sobre a aplicação de *Autoencoders* e LSTM à detecção de anomalias em *system calls* descrita na Seção 3.8.1, o modelo de *machine learning* implementado neste trabalho é baseado nesses tipos de Redes Neurais, que conseguem lidar com *datasets* sequenciais e sem rótulo de classe.

Com o objetivo de reduzir a sobrecarga de processamento no *cluster* Kubernetes, o módulo de *machine learning* foi configurado em um *host* dedicado a esse tipo de carga de trabalho. Devido ao grande número de operações com matrizes realizadas pelos modelos de *Deep Learning* e à alta demanda por poder computacional, processadores gráficos (GPUs) são necessários para o treinamento desses modelos em um tempo mais aceitável [11] [65]. Deste modo, um *host* com uma ou mais GPUs é recomendado para execução do módulo de *machine learning*. Na configuração do *host* usado neste trabalho, o Sistema Operacional GNU/Linux (*Kernel* 5.19.8) foi utilizado com o Python e suas dependências instaladas para execução dos modelos de Redes Neurais.

Para integração entre o módulo de *machine learning* e o *cluster* Elasticsearch na camada 3, o requisito existente é que a comunicação seja realizada através da API do Elasticsearch para leitura e gravação de dados. Em ambientes baseados em Python, existe uma biblioteca que faz a integração com o Elasticsearch [13]. Uma vez que este requisito seja atendido, é possível que outras ferramentas e algoritmos de *machine learning* sejam utilizados no *framework* visando obter o melhor desempenho conforme o problema a ser resolvido.

Diversas bibliotecas do Python foram utilizadas para implementar as funções realizadas pelo módulo de *machine learning* nesta camada. Dentre elas destacam-se as bibliotecas: Keras e Tensorflow. O Tensorflow é um projeto *open-source* da empresa Google, voltado para implementação de algoritmos de Redes Neurais Profundas. O Tensorflow tem sido largamente utilizado em pesquisas envolvendo *machine learning* aplicado a áreas da ciência da computação e outros campos, incluindo reconhecimento de fala, visão computacional, robótica, recuperação de informações, processamento de linguagem natural (NLP), extração de informações geográficas e descoberta de drogas computacionais [70]. O Keras é uma API Python utilizada para *Deep Learning* baseada no Tensorflow. Projetada com foco no usuário, a biblioteca Keras provê uma interface simples, minimizando o número de ações necessárias para implementação de casos de usos comuns, além de fornecer uma vasta documentação e guias de desenvolvimento [71]. A utilização conjunta das bibliotecas Keras e Tensorflow facilitou a implementação e treinamento de modelos de DL bem como a condução dos testes.

Outras bibliotecas como Pandas e Scikit-learn também foram usadas nas demais etapas de processamento do *pipeline* executado pelo módulo de *machine learning*. Ainda, para fazer uso do poder computacional das GPUs produzidas pela fabricante NVIDIA, é necessário que se utilize a biblioteca cuDNN disponibilizada pela empresa para otimização de desempenho no Aprendizado Profundo. Em trabalho onde a biblioteca cuDNN foi apresentada, os autores alcançaram um aumento de 36% no desempenho de um modelo de Rede Neural além da redução no consumo de memória [72].

Conforme os fluxogramas representados pelas Figuras 4.1, 4.2 e 4.3, o módulo de *machine learning*

pode operar em dois modos distintos: treinamento (fase de treinamento) e detecção (fases de testes e produção). Em ambos os modos o processamento das informações tem início através da leitura dos *índices* contendo os *datasets* no Elasticsearch.

Tanto no modo de treinamento quanto no modo de detecção, o módulo de *machine learning* interage com o *cluster* Elasticsearch obtendo os *datasets* via API. Neste trabalho, para os experimentos realizados com *dataset* próprio, as *features* elencadas na Tabela 4.2 foram utilizadas como atributos do *dataset*. Para facilitar a interação entre o módulo de *machine learning* e o *cluster* Elasticsearch, uma função em Python denominada *read_from_elastic* foi desenvolvida fazendo uso da biblioteca *elasticsearch_dsl* e foi disponibilizada através de um arquivo de funções customizadas que são utilizadas em outras operações no *pipeline* de processamento do módulo de *machine learning*. Detalhes do arquivo contendo as funções desenvolvidas podem ser encontrados no Apêndice I.5.

A função *read_from_elastic* faz a leitura de um índice do Elasticsearch e seleciona apenas as *features* definidas na Tabela 4.2 com adição do campo *timestamp* de cada documento do índice. Devido ao fato da biblioteca *elasticsearch_dsl* não retornar os documentos do índice de maneira ordenada, o campo *timestamp* é utilizado para ordenação das *system calls* conforme a sequência original de captura, e após isso, o campo é removido. Como resultado da função *read_from_elastic*, um *dataframe* Pandas é retornado contendo em suas linhas as *system calls*, ordenadas conforme a sequência temporal de captura e em suas colunas as *features*: *systemcall_type*, *systemcall*, *process_name*, *latency*, *return_value*, *args_length* e *argc*. Esse *dataframe* será então utilizado nas demais operações de pré-processamento e geração de um *dataset* Tensorflow que será recebido como entrada pelo modelo de *machine learning*.

Tabela 4.2: *Features* do *dataset* construído.

Feature	**Descrição**
systemcall_type	Categoria do evento (ex: 'file' para operações de entrada e saída, 'net' para operações de rede, etc.)
systemcall	Nome da system call
process_name	Nome do executável que gerou o evento
latency	Delta entre o horário de entrada e saída de um evento em nano segundos
return_value	Valor de retorno do evento
args_length	Quantidade de caracteres dos argumentos da system call
argc	Número de argumentos da system call

Para facilitar o entendimento sobre os modos de funcionamento do módulo de *machine learning*, os Pseudocódigos 1 e 2 representam o modo de treinamento e o modo de detecção do módulo de *machine learning*, respectivamente.

No modo de treinamento, o *dataframe* gerado contém apenas dados de comportamentos considerados benignos e é dividido em duas partes na proporção 80/20. Ou seja, 80% dos dados são utilizados para treinamento do modelo e os 20% restantes são usados para teste do modelo treinado. A proporção de divisão do *dataframe* entre o conjunto de treinamento e o de teste irá depender do tamanho do *dataset*, e essa divisão é necessária para avaliar o desempenho de predição do modelo testando-o com dados desconhecidos [9].

O pré-processamento do *dataframe* envolve operações como a normalização dos dados e a criação das janelas deslizantes contendo os agrupamentos de *system calls* em sequência. As janelas de *system calls*

Pseudocódigo 1: Processamento do módulo de *machine learning* (modo treinamento)

1: Definição dos parâmetros de conexão (index, es) com o Elasticsearch
2: $df\_benign\_data \leftarrow read\_from\_elastic(index, es)$
3: $df\_train\_data \leftarrow$ 80% de df_benign_data
4: $df\_test\_data \leftarrow$ 20% de df_benign_data
5: $window\_size \leftarrow 6$ (definido arbitrariamente)
6: $n\_features \leftarrow$ número de colunas de df_benign_data
7: $train\_data \leftarrow MinMaxScaler(df\_train\_data)$
8: $train\_data\_wz \leftarrow sliding\_window(train\_data, window\_size)$
9: $ds\_train\_full \leftarrow$ Gera dataset Tensorflow a partir de $(train\_data\_wz)$
10: $ds\_train \leftarrow$ 95% de ds_train_full
11: $ds\_validation \leftarrow$ 5% de ds_train_full
12: $modelo \leftarrow$ Definição do modelo
13: $parametros\_treinamento \leftarrow$ Parâmetros de treinamento
14: **Enquanto** $parametros\_treinamento$ forem satisfeitos **Faça:**
15: $modelo.fit(ds\_train, ds\_validation, parametros\_treinamento)$
16: **Fim Enquanto**
17: Salvar modelo treinado
18: $test\_data \leftarrow MinMaxScaler(df\_test\_data)$
19: $test\_data\_wz \leftarrow sliding\_window(test\_data, window\_size)$
20: $ds\_test \leftarrow$ Gerar dataset Tensorflow a partir de $test\_data\_wz$
21: $df\_pred \leftarrow model.predict(ds\_test)$
22: Calcular diferença de valor (erro) entre df_pred e ds_test
23: Salvar limiar de erro para os intervalos de confiança (1, 0.995, 0.99, 0.98, 0.97)

consistem em uma estrutura da dados que armazena sequências de *system calls* de múltiplos tamanhos e é uma abordagem comum para análise de *system calls* [13]. A escolha do tamanho das janelas de *system calls* é um fator que influencia diretamente nos resultados de detecção, sendo que a literatura recomenda valores na ordem de 6 a 10 para o tamanho da janela [20, 23, 73, 74]. A normalização dos dados do *dataframe* consiste em aplicar transformações nos valores numéricos de cada *feature*. Uma vez que cada *feature* pode estar representada em escalas diferentes, essa discrepância numérica pode prejudicar o desempenho da maioria dos algoritmos, salvo algumas exceções [9].

O *dataset* Tensorflow é criado a partir do *dataframe* Pandas normalizado. Uma função denominada *sliding_window* foi desenvolvida para criar um vetor resultante de três dimensões, a saber: número de linhas x tamanho da janela x número de *features*. Essa estrutura de dados com três dimensões é necessária para trabalhar com Redes Neurais LSTM. A função *sliding_window* recebe como parâmetros o *dataframe* Pandas e o tamanho da janela de *system calls* a ser utilizada, e retorna o vetor de três dimensões que será usado para criar o *dataset* Tensorflow. A função *sliding_window* está descrita no Apêndice I.5. Ao se utilizar um *dataset* Tensorflow é possível tirar proveito dos recursos oferecidos pela biblioteca Tensorflow, que é responsável implementar recursos como *multithreading*, enfileiramento, *batching*, *prefetching* e trabalhar de maneira integrada com a biblioteca Keras [9].

Quando em modo de treinamento, o *dataset* de treinamento Tensorflow também é dividido em dois conjuntos, um para treinamento e outro para validação. A partir das métricas obtidas com o conjunto de validação, será possível considerar o desempenho do modelo como satisfatório e encerrar o treinamento. Após a preparação do *dataset* Tensorflow, as definições do modelo de *machine learning* e dos parâmetros

de treinamento são realizadas. Nessa etapa do processamento, a biblioteca Keras é utilizada para construção do modelo composto por camadas LSTM em uma arquitetura de Autoencoder, conforme descrito na Seção 3.8.1. Em seguida, o ciclo de treinamento do modelo com os dados de treinamento poderá ser iniciado. O treinamento de um modelo de Rede Neural pode levar um longo período de tempo até que o seu desempenho seja satisfatório. Fatores como o *dataset* de treinamento utilizado, parâmetros do modelo, e parâmetros de treinamento utilizados, podem influenciar diretamente no tempo de treinamento, sendo uma etapa que envolve ajustes finos em uma abordagem empírica. Uma vez que o treinamento do modelo chega ao fim por alcançar o desempenho desejado ou não, o modelo treinado é salvo no formato *h5*, possibilitando o carregamento posterior sem a necessidade de treinamento.

Ainda em modo de treinamento, após o modelo treinado, o *dataframe* contendo o conjunto de teste é pré-processado e normalizado em semelhança ao *dataframe* de treinamento. O *dataset* Tensorflow é gerado para o conjunto de teste e é lido como entrada pelo modelo para predição dos valores recebidos. O erro para cada predição realizada pelo modelo é calculado e um limiar de erro aceitável (*loss threshold*) é computado conforme os seguintes intervalos de confiança: 100%, 99.5%, 99%, 98% e 97% de acordo com o algoritmo apresentado por [14] e implementado através da função *tunable_threshold* detalhada no Apêndice I.5. Os valores de erro aceitável referentes ao modelo e ao *dataset* utilizados são então registrados para uso posterior, quando o módulo de *machine learning* operar em modo de detecção.

Pseudocódigo 2: Processamento do módulo de *machine learning* (modo detecção)

1: Definição dos parâmetros de conexão (index, es) com o Elasticsearch
2: $df\_detection \leftarrow read\_from\_elastic(index, es)$
3: $window\_size \leftarrow 6$ (definido arbitrariamente)
4: $n\_features \leftarrow$ número de colunas de $df\_detection$
5: $df\_data \leftarrow MinMaxScaler(df\_detection)$
6: $df\_data\_wz \leftarrow sliding\_window(df\_data, window\_size)$
7: $ds\_data \leftarrow$ Gerar *dataset* Tensorflow a partir de (df_data_wz)
8: $modelo \leftarrow load\_model(filepath, compile = True)$
9: $df\_pred \leftarrow model.predict(ds\_data)$
10: Definir limiar de erro aceitável
11: Calcular diferença de valor (erro) entre df_pred e ds_data
12: Verificar erros que excederam o limiar aceitável
13: Gravar em disco *dataframe* com o cálculo de erro
14: Gravar no Elasticsearch resultados de detecção

Já em operação no modo de detecção, o módulo de *machine learning* fará uso do modelo já treinado para realizar a detecção de anomalias em um *dataset* contendo comportamentos de aplicação que podem ou não serem maliciosos. Desta forma, o *dataframe* resultante da função textitread_from_elastic não será divido em dois conjuntos, mas será utilizado por completo para detecção. A exemplo do que ocorre no modo de treinamento, o pré-processamento e normalização dos dados também são executados, e um *dataset* Tensorflow é gerado.

O modelo treinado previamente é carregado com seus parâmetros e pesos, e o *dataset* Tensorflow é utilizado pelo modelo apenas para predição dos dados. O erro para cada predição realizada pelo modelo é calculado e posteriormente confrontado com o limiar de erro aceitável escolhido com base no intervalo de confiança utilizado. Os limiares de erro aceitável disponíveis correspondem aos obtidos durante o

treinamento do modelo para cada intervalo de confiança: 100%, 99.5%, 99%, 98% e 97%.

A detecção de anomalias é realizada tendo como referência o limiar de erro aceitável definido. Deste modo, as predições realizadas pelo modelo que tiverem excedido o limiar são consideradas anômalas. O resultado desse cálculo é gravado em um *dataframe* em formato *pkl* e poderá ser utilizado em análise posterior para ajuste do limiar de erro. Por fim, o resultado da detecção de anomalias para cada predição do modelo é gravado em um índice criado no Elasticsearch, onde poderá ser visualizado e analisado graficamente pela equipe do SOC através da ferramenta Kibana ou através de *notebooks* elaborados em Python.

4.1.5 Camada 5 - Geração de alertas e análise de dados

Na última camada da arquitetura proposta, concentram-se as atividades de análise de dados e alertas, avaliação de desempenho das detecções realizadas pelo módulo de *machine learning*, ajuste de *threshold* de detecção (limiar de erro aceitável), além da atuação do SOC em caso de alertas. Para o desempenho dessas atividades, os *dataframes* gerados na camada anterior e o Kibana são importantes ferramentas de auxílio à equipe do SOC. Conforme os fluxogramas das Figuras 4.1, 4.2 e 4.3, as ações realizadas na camada 5 são executadas nas fases de teste e produção do *framework*.

Durante a fase de teste, não há atuação da equipe do SOC na ocorrência de alertas e eventuais anomalias detectadas. Assim, o principal objetivo da operação do *framework* nesta fase é avaliar o desempenho do IDS antes de sua entrada em produção. Após a indexação dos *datasets* e dos resultados de detecção no Elasticsearch feitos na camada 3, é possível analisar e inspecionar esses dados através do Kibana. O Kibana tem integração nativa com o Elasticsearch e fornece uma interface gráfica para análise tanto dos índices de *datasets* quanto de anomalias gerados pelo IDS [13]. As Figuras 4.9 e 4.10 ilustram respectivamente, uma interface para pesquisa e análise de *system calls* indexadas em um período de 24 horas e um *dashboard* para visualizar as características de um *dataset*.

Ainda na fase de teste, o desempenho do IDS poderá ser avaliado conforme a análise dos alertas gerados pelo módulo de *machine learning* além das métricas de detecção calculadas através das Equações 3.1 a 3.7. Conforme o desempenho do módulo de *machine learning*, é possível ajustar o limiar de erro aceitável para outros intervalos de confiança a fim de aprimorar o desempenho na detecção de anomalias. A modificação do intervalo de confiança dos erros calculados para as predições realizadas pelo modelo influencia diretamente no limiar de erro aceitável utilizado para a detecção de anomalias. Ou seja, ao se utilizar o intervalo de confiança de 100% dos erros obtidos para o cálculo do limiar de erro aceitável, o maior erro obtido em 100% das predições será utilizado. Caso um intervalo de confiança menor seja utilizado, um percentual menor do que 100% dos erros obtidos será levado em consideração para a seleção do maior erro. Como a distribuição dos erros das predições se assemelha a uma distribuição normal, a maior parte das amostras está concentrada no centro da curva. Desta forma, o limiar de erro aceitável obtido é reduzido conforme o intervalo de confiança utilizado for diminuído, e quanto menor o limiar de erro aceitável, mais sensível a detecção de anomalias o modelo se torna. A Figura 4.11 ilustra um exemplo de distribuição dos erros de predição realizados por um modelo de *machine learning*.

Diante do exposto, a sensibilidade de detecção do IDS pode ser ajustada variando-se o intervalo de

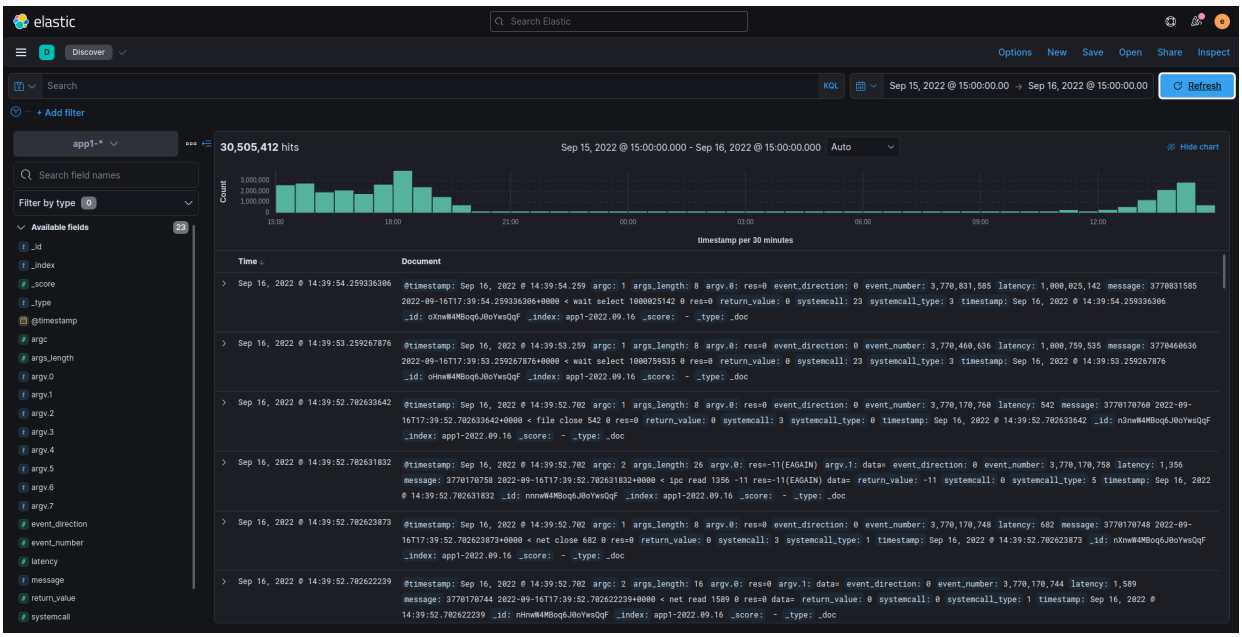

Figura 4.9: Visualização de um *dataset* de *system calls* indexado.

Figura 4.10: *Dashboard* criado no Kibana para visualizar um *dataset* de *system calls* [13].

confiança e o limiar de erro correspondente. Ainda assim, caso o desempenho do IDS ainda esteja insatisfatório após o ajuste do limiar de erro aceitável, a fase de teste poderá ser encerrada e a operação do *framework* voltar para a fase de treinamento para aprimoramento do módulo de *machine learning*.

Na operação do *framework* em fase de produção, a atividade de atuação da equipe do SOC é inserida na camada 5 do fluxograma descrito na Figura 4.3. Por conseguinte, a detecção de anomalias e alertas produzidos pelo IDS provoca uma atuação por parte da equipe do SOC para analisar o evento. A análise preliminar dos alertas e anomalias detectadas pelo IDS é realizada através da interface do Kibana, que permite que

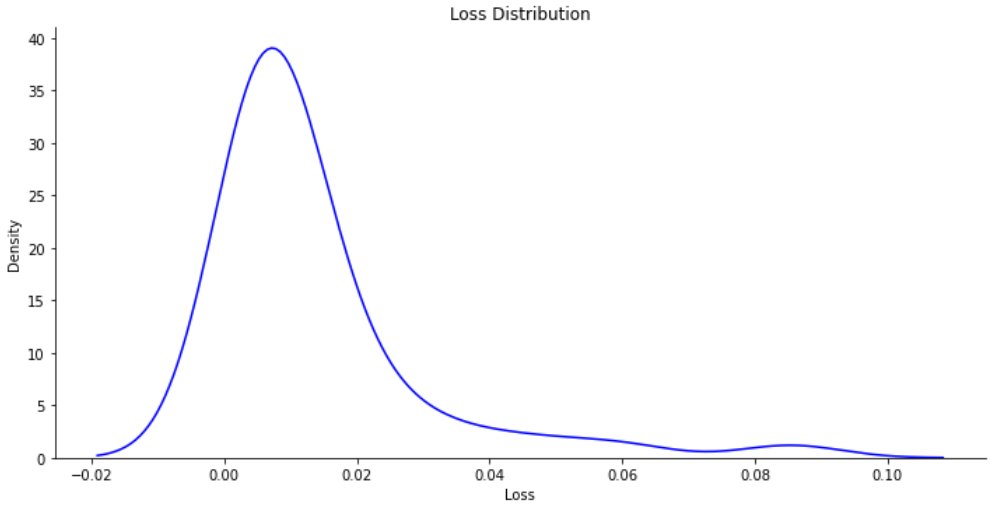

Figura 4.11: Exemplo de uma distribuição dos erros de predição realizados por um modelo.

as janelas de *system calls* com escore anômalo sejam identificadas. A anomalia detectada em determinado momento também deve ser investigada através de outras ferramentas de segurança e monitoramento do ambiente, visando a correlação do alerta com outros eventos de segurança. Como um HIDS geralmente não consegue prover uma proteção completa aos sistemas, na indústria atual, um HIDS é normalmente integrado com outros mecanismos de segurança como a análise e gerenciamento de vulnerabilidades e a resposta a incidentes [20]. Caso o desempenho do IDS esteja insatisfatório, é possível que o limiar de erro aceitável também seja ajustado para outros intervalos de confiança, alterando a sensibilidade de detecção do IDS. Não obstante, caso o desempenho do IDS não esteja adequado, o *framework* poderá voltar a operar na fase de treinamento para aprimoramento do modelo de *machine learning*.

Após a análise e investigação de uma anomalia, a equipe do SOC poderá atuar com medidas para conter a propagação de um possível ataque através de ações como: a restrição da comunicação da aplicação a contextos limitados (*sandboxing*), verificar alterações na configuração da aplicação em execução (ex: *deploy* recente de uma nova versão), acionar a equipe responsável pelo desenvolvimento da aplicação, ou até mesmo a interrupção da execução do contêiner da aplicação que gerou o alerta.

Através das ferramentas da camada de análise de dados e alertas, é possível que *dashboards* sejam criados com métricas e alertas customizados na ocorrência de anomalias em aplicações conteinerizadas. Isto posto, as informações geradas pelo IDS da plataforma de contêineres oferecem uma maior visibilidade acerca da segurança desse ambiente, representando uma relevante fonte adicional de eventos de segurança para subsidiar o monitoramento do SOC e realizar a correlação com eventos gerados por outras ferramentas.

4.2 TOPOLOGIA

Para implementação do *framework* e sua arquitetura em camadas, o *software* emulador de redes GNS3 [75] foi utilizado na versão 2.2.29, instalado em computador conforme as características abaixo:

- S.O: GNU/Linux (kernel 5.19.8)
- Tipo: Desktop (placa-mãe Asus Z97-K)
- CPU: Intel Core i7-4790K 4.00GHz
- Memória: 32GB (4 x 8GB 1866MHz)
- GPU: Nvidia GeForce RTX 3060 (driver 515.65.01, CUDA 11.7, cuDNN 8.5.0.96)

Uma topologia funcional representando um ambiente corporativo foi construída utilizando as redes constantes na Tabela 4.3, contendo diferentes dispositivos de rede como: *switches*, roteadores, e um *firewall*, de acordo com o representado na Figura 4.12.

Tabela 4.3: Sub-redes utilizadas na topologia.

#	Rede	Máscara de sub-rede
0	192.168.100.0	255.255.255.128
1	192.168.100.128	255.255.255.128
2	192.168.200.0	255.255.255.0
3	192.168.201.0	255.255.255.0
4	192.168.202.0	255.255.255.0
5	192.168.203.0	255.255.255.0
6	192.168.204.0	255.255.255.0
7	192.168.205.0	255.255.255.0

As seguintes VLANs departamentais detalhadas na Tabela 4.4 foram utilizadas na topologia: uma Zona Desmilitarizada (DMZ), para hospedar serviços acessíveis através da internet como: e-mail, servidor web, e DNS; uma sub-rede compreendendo serviços do SOC; uma sub-rede para o *datacenter*, contendo um *cluster* de orquestração de contêineres Kubernetes; uma sub-rede para o departamento de TI; e uma sub-rede para as estações de trabalho corporativas (*Enterprise Network*).

Tabela 4.4: Vlans utilizadas na topologia.

VLAN	VLAN Id
DMZ	100
SOC	201
Datacenter	202
Enterprise 1	203
Enterprise 2	204
IT Department	205

Na topologia representada na Figura 4.12, a DMZ está inserida na rede corporativa segregada através do *firewall*, sendo este, seu *gateway* para comunicação com as demais redes internas. O *firewall* pfSense [76] na versão 2.5.1 foi configurado de maneira a prover o isolamento entre o acesso externo oriundo da Internet, a rede de DMZ, e as demais redes internas através da filtragem e inspeção profunda de pacotes. O roteador de borda (*Border-Router*), possui uma interface com conectividade para a Internet e outra interface na sub-rede #0, fazendo a tradução de endereços (NAT - *Network Address Translation*) na comunicação com redes externas à topologia. Por outro lado, o roteador interno (*Router*), age como *gateway* das demais sub-redes internas, possibilitando o encaminhamento de pacotes através de roteamento estático. Já os *switches* utilizados na topologia, habilitam a comutação de quadros na camada de enlace na comunicação dos diferentes *hosts* existentes em cada sub-rede, além de implementar a segmentação através das VLANs.

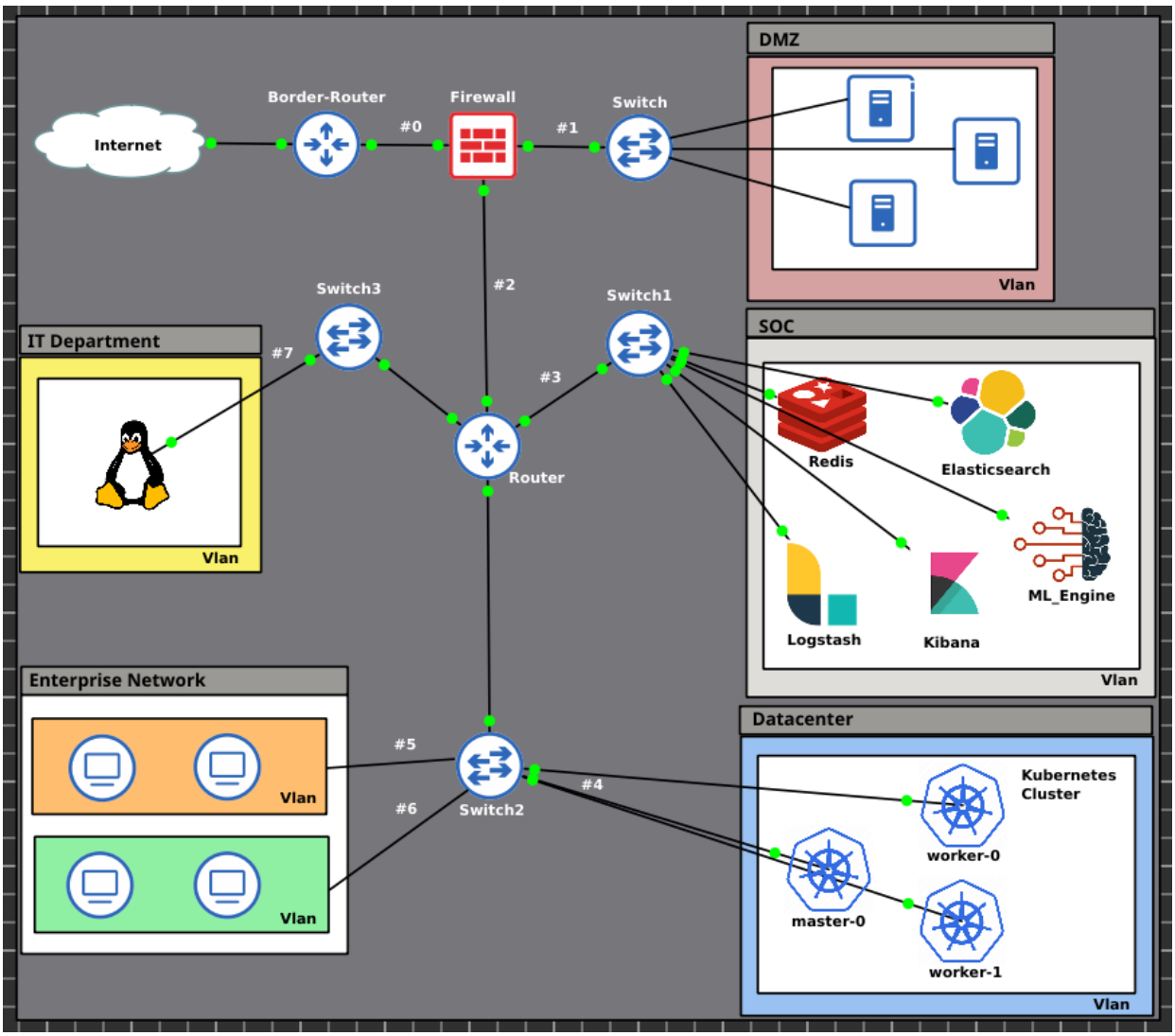

Figura 4.12: Topologia implementada no *software* de emulação de redes GNS3 [13].

Na VLAN de *Datacenter*, um *cluster* de orquestração Kubernetes composto por três nós: *master-0*, *worker-0* e *worker-1*, foi instalado na versão 1.18 e serve como plataforma para execução de aplicações conteinerizadas. Os nós *worker-0* e *worker-1* são responsáveis por executar os contêineres de aplicações, e é onde as ferramentas Sysdig e Filebeat são instaladas para captura e encaminhamento das *system calls*.

Na VLAN do SOC, encontram-se os demais componentes da arquitetura descrita nas seções anteriores deste capítulo. As ferramentas Redis, Logstash, Elasticsearch, Kibana, e o módulo de *machine learning* representados em cada camada tem ampla conectividade entre si nesta sub-rede e se integram formando o núcleo do Sistema de Detecção de Intrusão. Essas ferramentas poderão ser acessadas a partir da VLAN (*IT Department*, onde estarão os equipamentos da equipe do SOC e demais profissionais do departamento de TI.

Através da topologia de rede construída no *software* GNS3, foi possível demonstrar a viabilidade de implementação da arquitetura proposta e validar seu funcionamento em uma infraestrutura de rede simulada semelhante à encontrada em ambientes corporativos. Informações adicionais acerca da configuração dos principais componentes da topologia estão disponíveis em um repositório público no github [77].

5 EXPERIMENTAÇÃO E RESULTADOS

Neste capítulo, serão fornecidos detalhes da execução de um experimento realizado com o *framework* implementado no ambiente simulado descrito na Figura 4.12, demonstrando sua capacidade de funcionamento e integração entre as camadas da arquitetura proposta. No experimento 5.1, foi utilizando um *dataset* público como fonte de dados de uma aplicação conteinerizada e os resultados de detecção obtidos foram discutidos.

5.1 AVALIAÇÃO DO FRAMEWORK COM UM DATASET PÚBLICO

O trabalho publicado por [14] fez uso de ferramentas de introspecção e uma abordagem de *machine learning* não supervisionada para prover um monitoramento não intrusivo de uma aplicação conteinerizada. Como artefato resultante dessa pesquisa, um *dataset* foi disponibilizado contendo *system calls* de comportamentos benignos e anômalos de uma aplicação conteinerizada. Para geração desse *dataset*, a ferramenta Sysdig foi utilizada e os seguintes atributos foram coletados para cada *system call*:

- *syscall* – nome da *system call* invocada
- *time* – carimbo de tempo no momento da invocação da *system call*
- *process* – nome do processo que invocou a *system call*
- *fields* – quantidade de argumentos da *system call* invocada
- *length* – número de caracteres dos argumentos
- *value* – valor codificado dos argumentos

Nesse *dataset*, os dados originais de cada *system call* passaram por um procedimento de pré-processamento onde os valores de todos os atributos foram convertidos para valores numéricos [14].

Conforme explanado por [14], para geração dos dados de comportamento benigno, um servidor de banco de dados MySQL conteinerizado foi utilizado como aplicação de exemplo e pares aleatórios de usuários e senhas foram gerados, inseridos e atualizados em tabelas de maneira randômica no banco de dados através de um cliente MySQL. Sete ataques foram selecionados e executados para geração do comportamento anômalo da aplicação, quais são: *Brute Force Login*, *Simple Remote Shell*, *Meterpreter*, *Malicious Python Script*, *Docker Escape*, *SQL Misbehavior* e *SQL Injection*. Uma descrição de cada ataque executado, além de mais detalhes do experimento realizado podem ser encontrados na mesma publicação.

Para avaliação do *framework* proposto neste trabalho, o *dataset* disponibilizado por [14] foi copiado para um nó do *cluster* Kubernetes da topologia da Figura 4.12, representando uma captura realizada pelo Sysdig de uma aplicação em execução no *cluster*. Como o *dataset* público foi disponibilizado no Github em várias partes em formato CSV (Valores Separados por Vírgula), foi necessário concatenar as diversas partes resultando em um arquivo único para cada comportamento (normal ou de ataque), conforme a Figura 5.1.

```
root@worker-1:/var/log/sysdig/public-dataset# tree
.
├── benign
│   └── normal-full.csv
└── malicious
    ├── delete-full.csv
    ├── escape-full.csv
    ├── hydra-full.csv
    ├── injection-full.csv
    ├── meterpreter-full.csv
    ├── nc-full.csv
    └── python-full.csv

2 directories, 8 files
```

Figura 5.1: Arquivos do *dataset* público copiados para o nó do *cluster* Kubernetes.

A partir de então, os arquivos representando o *dataset* puderam ser lidos pela ferramenta Filebeat e o *pipeline* de processamento descrito na arquitetura 4.4 iniciado a partir da camada 1. Para indexação no Elasticsearch, cada arquivo CSV construído foi lido separadamente pelo Filebeat e um índice individual foi criado pelo Logstash no Elasticsearch conforme a Figura 5.2.

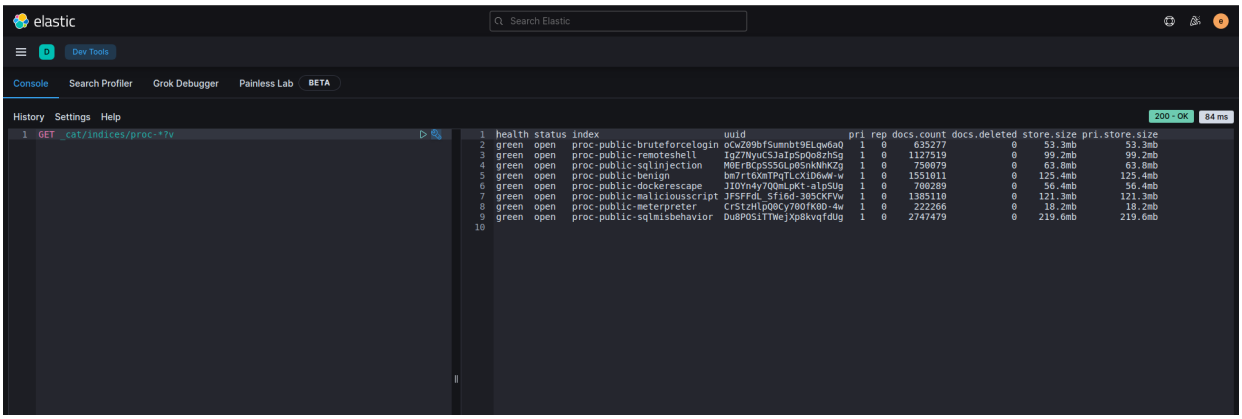

Figura 5.2: Índices criados no Elasticsearch contendo os dados do *dataset* público.

A exemplo do *dashboard* ilustrado pela Figura 4.10, um *dashborad* foi criado para melhor visualizar as características do *dataset* de comportamento benigno. A Figura 5.3 apresenta esse *dashboard*, que permite a interação do usuário através de filtragens, agregações e construção de outros gráficos para inspeção e entendimento dos dados. Compondo o *dashboard* da Figura 5.3 relaciona-se os gráficos contendo: o número total de *system calls* do *dataset*, a distribuição de *system calls* por processo, a distribuição do número de argumentos, um histograma com a distribuição das *system calls* por *timestamp*, um gráfico em linha com o histograma do número de argumentos por *timestamp*, um mapa com as principais *system calls* por processo, e um mapa de proporção com as *system calls* mais invocadas.

Com os dados já armazenados no Elasticsearch, o treinamento do módulo de *machine learning* foi realizado fazendo a leitura do *dataset* benigno representado pelo índice *proc-public-benign*, com 1.551.011 documentos (*system calls*) indexados. O processamento realizado pelo módulo de *machine learning* em modo de treinamento se deu conforme o Pseudocódigo 1 apresentado na Seção 4.1.4 e o código no formato *Jupyter Notebook* executado está disponível no Apêndice I.6.

Uma das tarefas executadas pelo código de treinamento, envolve a criação de uma base contendo o cálculo do *hash* das janelas de *system calls* do *dataset* benigno. Essa base contém a lista dos *hashes* únicos computados de todas as janelas de *system calls* coletadas durante o comportamento normal da aplicação.

Figura 5.3: *Dashboard* para visualização do *dataset* público com comportamento benigno.

De maneira análoga, realizando-se o cálculo dos *hashes* únicos das janelas de *system calls* do *dataset* com o comportamento malicioso, é possível gerar rótulos para o *dataset* de ataques. Assim, as janelas de *system calls* cujo cálculo do *hash* for igual a algum dos *hashes* constantes na base de *hashes* benignos, também serão consideradas benignas. Deste forma, é possível realizar a medição de desempenho do modelo de *machine learning*, comparando-se a predição realizada pelo modelo com o rótulo correspondente à determinada janela de *system call*. Esse procedimento foi necessário pois o *dataset* malicioso disponibilizado por [14], inclui comportamentos normais da aplicação mesclados a comportamentos anômalos, mas não provê a informação de rótulo indicando a classe (benigna ou anômala) das *system calls*. Sabe-se que existem diferentes técnicas para realizar a classificação e rotulagem de um *dataset*. Contudo, a abordagem utilizada se beneficia da facilidade de implementação e apresenta uma alternativa às técnicas computacionalmente mais onerosas, possibilitando uma análise preliminar dos resultados.

Conforme pode ser visto no bloco de código abaixo, a estrutura de dados *df_benign_data_hash* contém os *hashes* das janelas de *system calls* do *dataset* benigno e possui 1.551.006 registros, sendo 395.138 o número de *hashes* únicos, que irão compor o banco de *hashes* benignos. O tamanho das janelas de *system calls* utilizado neste experimento foi igual a seis. Isso explica o motivo de o número total de janelas ser equivalente a seis a menos do total de *system calls* do *dataset* benigno.

```
df_benign_data_hash.describe()

count                1551006
unique                395138
top       2c8fa761cb9a614b
freq                   68757
dtype: object
```

```
df_benign_data_hash.value_counts()

2c8fa761cb9a614b    68757
b5eeb6c4d471038a    66268
4c7213b763fd9aec    10055
e8f64dc5dc3af126     7538
9d043911e3521014     7220
                    ...
fa468b0a6ef2fc13        1
50cfd35f515614ae        1
f5bd5b6bd6d40d93        1
8fe902b46bd50527        1
0462e678e9c87bd0        1
Length: 395138, dtype: int64
```

O modelo utilizado no experimento foi obtido através do Hyperopt dentre uma gama de modelos possíveis. O Hyperopt é uma biblioteca em Python utilizada para otimização de hiper-parâmetros de algoritmos de *machine learning* [78]. Dessa forma, através do uso dessa biblioteca é possível testar a combinação de diversos hiper-parâmetros que compõem um modelo, buscando pela combinação que atinja o melhor desempenho. Outras combinações de parâmetros como tamanho da janela de *system calls*, funções de ativação, funções de normalização de dados, etc, foram testadas empiricamente para se chegar no modelo final utilizado no experimento.

Assim como no modelo de *machine learning* empregado por [14], um Rede Neural do tipo DAE com unidades LSTM foi a estrutura base utilizada para se chegar no modelo final deste experimento. A partir dessa estrutura, um camada de Rede Neural Convolucional (Conv1D) foi adicionada ao modelo sequencial antes das camadas na estrutura de *Autoencoder*. Essa camada Conv1D tem a capacidade de extrair as *features* mais representativas de dados complexos, eliminando as *features* menos significativas para classificação [79]. Para o *dataset* utilizado neste experimento essa camada proporcionou um ganho de desempenho na predição dos dados pelo modelo. O bloco de código a seguir apresenta a definição do modelo que obteve os melhores resultados na busca realizada com o Hyperopt.

```
model = Sequential()

# Conv1D
model.add(keras.layers.Conv1D(filters=n_features, kernel_size=window_size,
↪  strides=1, padding="same", activation="relu", input_shape=(None, window_size,
↪  n_features)))

# Encoder
model.add(CuDNNLSTM(160, kernel_initializer='he_normal', return_sequences=True))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation('tanh'))

model.add(CuDNNLSTM(64, kernel_initializer='he_normal', return_sequences=True))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation('tanh'))
```

```
model.add(CuDNNLSTM(24, kernel_initializer='he_normal', return_sequences=False))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation('tanh'))

model.add(RepeatVector(window_size))

# Decoder
model.add(CuDNNLSTM(24, kernel_initializer='he_normal', return_sequences=True))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation('tanh'))

model.add(CuDNNLSTM(64, kernel_initializer='he_normal', return_sequences=True))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation('tanh'))

model.add(CuDNNLSTM(160, kernel_initializer='he_normal', return_sequences=True))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation('tanh'))

model.add(TimeDistributed(Dense(n_features)))

model.compile(loss='mae', optimizer='nadam', metrics=['accuracy'])
```

A Tabela 5.1 detalha os valores dos limiares de erro aceitáveis (*Loss thresholds)* obtidos utilizando o modelo definido acima para cada intervalo de confiança durante os testes realizados com o *dataset* benigno. Após o treinamento do modelo e a obtenção dos limiares de erro, o módulo de *machine learning* foi operado em modo de detecção, fazendo a leitura de cada índice de ataque do Elasticsearch e executando as demais etapas do processamento, conforme o Pseudocódigo 2 descrito na Seção 4.1.4. O código no formato *Jupyter Notebook* executado para a detecção de anomalias nos ataques realizados está disponível no Apêndice I.7.

Tabela 5.1: Limiares de erro aceitável por intervalo de confiança.

Intervalo de confiança	***Loss threshold***
1	0.2456
0.995	0.0251
0.99	0.0164
0.98	0.0110
0.97	0.0091

Como resultado da detecção de anomalias realizada para cada ataque, um índice com as informações de detecção foi criado no Elasticsearch com os resultados obtidos. A Figura 5.4 ilustra um *dashboard* construído no Kibana para visualização dos dados indexados contendo anomalias detectadas durante o ataque de *Brute Force*, possibilitando a análise dos resultados obtidos de maneira interativa pela interface gráfica. Compondo o *dashboard* da Figura 5.4 relaciona-se os gráficos contendo: o total de janelas de *system calls* analisadas, um gráfico de pizza com o percentual de janelas anômalas e normais, o intervalo de confiança e *loss threshold* utilizado, um gráfico contendo o erro obtido em cada janela de *system call* por *timestamp* em constraste com o *loss threshold* definido, um histograma de distribuição das janelas

anômalas e normais por *timestamp*, e uma tabela com os detalhes de cada janela de *system call* analisada.

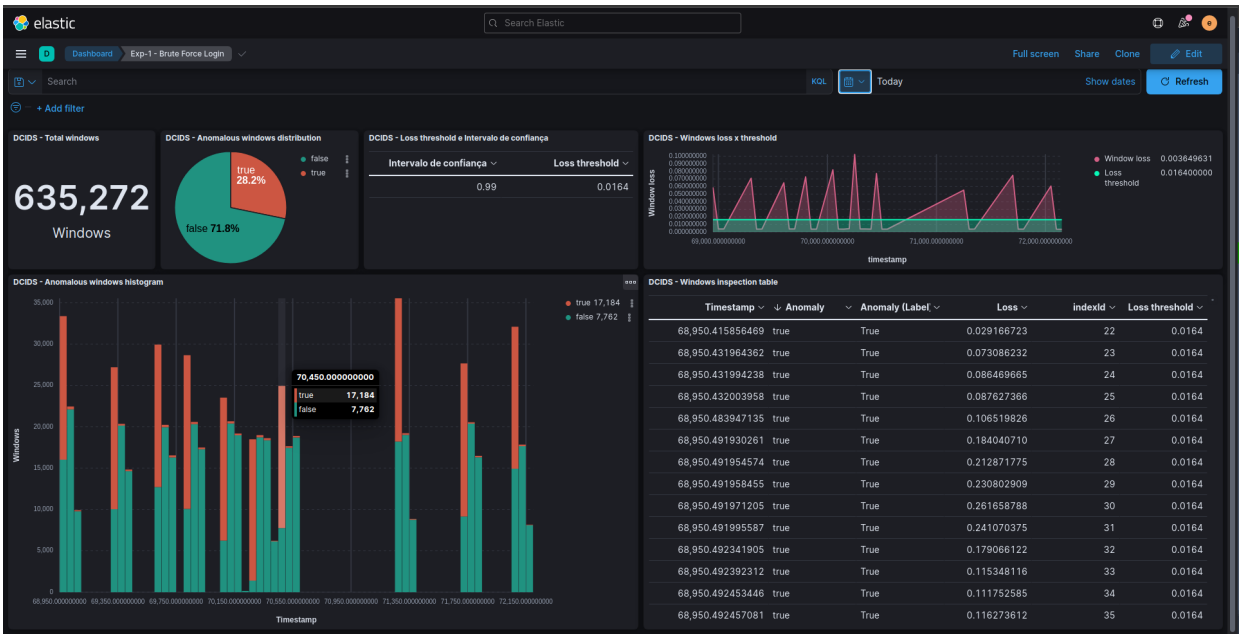

Figura 5.4: *Dashboard* para visualização de anomalias detectadas durante o ataque de *Brute Force*.

Para análise do desempenho na detecção dos ataques utilizando o *dataset* proposto, as seguintes métricas de avaliação foram adotadas por [14] em seu trabalho:

- NPV – conforme a Equação 3.4
- ACC (*Accuracy*) – conforme a Equação 3.1
- COV (*Coverage*) – conforme a Equação 3.5

A métrica ACC fornece o percentual de janelas de *system calls* classificadas corretamente (acurácia geral), enquanto que a métrica NPV representa a acurácia apenas da predição negativa [14]. Já a métrica COV provê uma visão geral do percentual de valores negativos corretamente classificados [14]. No total, 42 modelos compostos pela combinação de diferentes tamanhos de janelas de *system calls*, conjuntos de *features*, métodos de normalização de dados, e intervalos de confiança foram testados no experimento realizado por [14]. Contudo, a discussão dos resultados foi feita considerando apenas o intervalo de confiança de 99% e o tamanho da janela de *system calls* igual a 50.

A Figura 5.5 traz os resultados obtidos por [14] para seis dos sete ataques contidos no *dataset* e apresenta resultados para seis modelos utilizando diferentes funções de normalização de dados e conjuntos de *features* do *dataset*. Ainda que as métricas apresentadas na Tabela 5.5 sejam úteis para analisar o efeito das diferentes combinações de parâmetros no desempenho do modelo, na prática, o mesmo modelo compilado com seus parâmetros deverá ser utilizado para detecção de anomalias na aplicação para quaisquer dos ataques. Assim, espera-se que o modelo utilizado em um cenário real ofereça simultaneamente um desempenho aceitável para diferentes ataques.

Para fins de comparação com os resultados obtidos neste experimento, a Tabela 5.2 bem como o gráfico da Figura 5.6 foram gerados reunindo os resultados obtidos por [14] para cada ataque utilizando apenas o

Model	Brute Force Login			Remote Shell		
	NPV.	*ACC.*	*COV.*	*NPV.*	*ACC.*	*COV.*
MinMax-base	16.77%	86.55%	10.54%	**64.88%**	83.95%	34.33%
SFN-base	14.52%	85.42%	13.85%	63.86%	**84.54%**	37.93%
MinMax-prop	0.05%	84.47%	0.09%	62.5%	84.27%	31.34%
SFN-prop	67.19%	**94.12%**	**99.57%**	49.97%	84.35%	**99.98%**
MinMax-opt	1.95%	80.96%	2.08%	57.30%	84.39%	46.47%
SFN-opt	**67.85%**	92.09%	81.95%	49.37%	82.49%	99.36%

Model	Meterpreter			Malicious Script		
	NPV.	*ACC.*	*COV.*	*NPV.*	*ACC.*	*COV.*
MinMax-base	1.19%	**95.88%**	7.14%	0%	75.19%	0%
SFN-base	3.21%	91.27%	31.83%	0.11%	74.57%	0.12%
MinMax-prop	**14.35%**	96.2%	16.81%	36.73%	75.45%	4.25%
SFN-prop	13.73%	91.95%	**42.04%**	**77.22%**	**84.73%**	**68.36%**
MinMax-opt	12.91%	94.57%	15.81%	56.72%	77.14%	25.44%
SFN-opt	10.01%	95.39%	18.62%	67.62%	79.95%	48.41%

Model	Docker Escape			SQL Injection		
	NPV.	*ACC.*	*COV.*	*NPV.*	*ACC.*	*COV.*
MinMax-base	5.18%	41.28%	0.39%	6.66%	0.44%	0%
SFN-base	7.48%	42.02%	0.5%	5.55%	0.34%	0%
MinMax-prop	**86.13%**	71.45%	75.32%	8.14%	0.87%	0%
SFN-prop	6.06%	33.39%	0.6%	5.24%	1.16%	0%
MinMax-opt	77.6%	52.58%	24.31%	7.7%	1.01%	0%
SFN-opt	81.87%	**75.91%**	**75.67%**	7.00%	0.93%	0%

Figura 5.5: Métricas de desempenho na detecção dos ataques obtidas por [14].

modelo denominado *SFN-opt* da Tabela 5.5 e os resultados obtidos utilizando o *framework* proposto neste trabalho. O modelo *SFN-opt* da Figura 5.6 faz uso de uma função customizada para normalização dos dados e inclui como *features* do *dataset* o número de argumentos (*fields*) e o número de caracteres dos argumentos (*lenght*) das *system calls*, assim como neste experimento.

Tabela 5.2: Tabela comparativa das métricas de desempenho do modelo *SFN-opt* da Figura 5.5 e dos resultados obtidos neste trabalho.

Ataque	NPV		ACC		COV	
	SFN-opt	**Este trabalho**	**SFN-opt**	**Este trabalho**	**SFN-opt**	**Este trabalho**
Brute Force Login	67.85%	**78.34%**	92.09%	84.18%	81.95%	**99.53%**
Remote Shell	49.37%	**77.79%**	82.49%	**88.94%**	99.36%	99.31%
Meterpreter	10.01%	**85.55%**	95.39%	91.44%	18.62%	**99.70%**
Malicous Script	67.62%	**69.89%**	79.95%	**86.74%**	48.41%	**99.32%**
Docker Escape	81.87%	78.10%	75.91%	**95.48%**	75.67%	**99.50%**
SQL Injection	7.00%	**76.56%**	0.93%	**76.60%**	0%	**99.00%**
SQL Misbehavior	N/A	**60.12%**	N/A	**64.64%**	N/A	**99.77%**

Na Tabela 5.2, os valores obtidos por este trabalho que superaram o desempenho do estudo realizado por [14] utilizando o modelo *SFN-opt* foram grifados em negrito, o que corresponde a 80,95% de todas as métricas coletadas.

Para sumarizar os resultados obtidos neste experimento, a Tabela 5.3 contém as métricas obtidas para todos os ataques com o intervalo de confiança definido em 99%, o que equivale a um *loss threshold* igual a 0.0164. Além dos valores de NPV, ACC e COV, outras métricas como *Precision*, TPR, F1-Score e ROC-AUC demostradas nas Equações 3.2, 3.3, 3.7 e Figura 3.12 respectivamente, foram calculadas para melhor comparação dos resultados com possíveis trabalhos futuros.

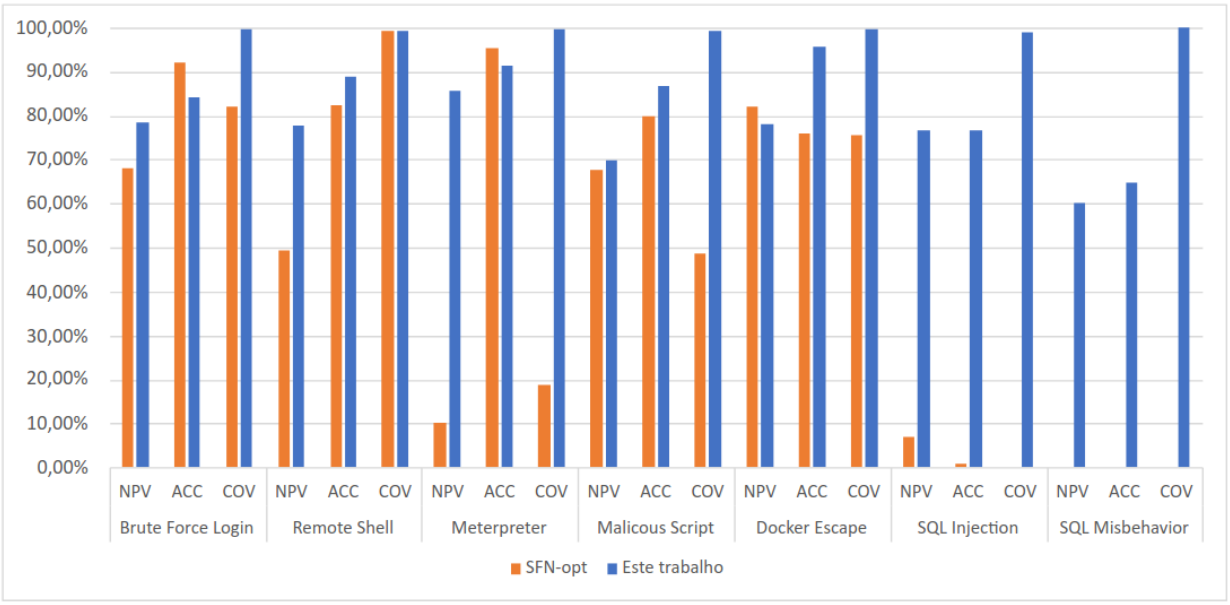

Figura 5.6: Gráfico comparativo das métricas de desempenho do modelo *SFN-opt* da Figura 5.5 e dos resultados obtidos neste trabalho.

Tabela 5.3: Métricas de desempenho na detecção dos ataques obtidas neste trabalho.

Ataque	NPV	ACC	COV	PRE	TPR	FPR	F1	ROC	Confiança	Threshold
Brute Force Login	78.34%	84.18%	99.53%	99.06%	64.24%	0.47%	77.93%	81.88%	99%	0.0164
Remote Shell	77.79%	88.94%	99.31%	99.49%	82.55%	0.69%	90.23%	90.33%	99%	0.0164
Meterpreter	85.55%	91.44%	99.70%	99.64%	83.21%	0.30%	90.69%	91.46%	99%	0.0164
Malicous Script	69.89%	86.74%	99.32%	99.63%	81.21%	0.68%	89.48%	90.26%	99%	0.0164
Docker Escape	78.10%	95.48%	99.50%	99.90%	94.72%	0.50%	97.24%	97.11%	99%	0.0164
SQL Injection	76.56%	76.60%	99.00%	77.76%	10.34%	1.00%	18.26%	54.67%	99%	0.0164
SQL Misbehavior	60.12%	64.64%	99.77%	98.96%	24.63%	0.23%	39.44%	62.20%	99%	0.0164

De acordo com [14], o experimento realizado em seu estudo falhou na detecção dos ataques de SQL *Injection* e SQL *Misbehavior*. Por outro lado, o modelo treinado no experimento realizado neste trabalho foi capaz de obter taxas significativamente melhores para esses ataques, como no caso da métrica de Cobertura (COV) com valores maiores ou iguais a 99%. Isso indica a capacidade do modelo de classificação negativa (TNR) de uma janela de *system call*. Conforme a Tabela 5.3, cabe ainda ressaltar a alta Precisão (PRE) obtida neste experimento com valores superiores a 98% para todos os ataques, com exceção do ataque de SQL *Injection*. Ademais, nota-se uma baixa FPR para todos os ataques, com valores menores ou iguais a 1%. Por outro lado, métricas como Acurácia (ACC) e TPR têm espaço para serem aprimoradas. Esse resultado provavelmente poderá ser alcançado com mais amostras de comportamentos malignos e benignos da aplicação complementando o *dataset* utilizado para treinamento do modelo. O aprimoramento do *dataset* utilizado neste experimento está entre os objetivos futuros dos autores, uma vez que a versão disponibilizada serve como uma linha de base [14].

Os autores do estudo apresentado em [14] afirmam que o trabalho realizado não tem por objetivo o desenvolvimento de uma nova técnica de *machine learning* para introspecção não supervisionada, mas sim fazer uso das técnicas do estado da arte para investigar a sua viabilidade de implementação. Do

mesmo modo, este trabalho não apresenta técnicas revolucionárias ou que já não existam atualmente. Mas visa primariamente a utilização de ferramentas e metodologias do estado da arte para demonstração da aplicabilidade de funcionamento do *framework* proposto neste trabalho. Por fim, este primeiro experimento atestou por meio da utilização de um *dataset* público que é possível fazer uso do *framework* proposto para prover uma camada adicional de segurança para detecção de anomalias em aplicações conteinerizadas.

6 CONCLUSÃO E TRABALHOS FUTUROS

6.1 CONCLUSÃO

Ambientes computacionais empregam diferentes tecnologias que em última instância, visam o atingimento dos objetivos de negócio através do aprimoramento dos processos e entrega de valor ao serviço prestado. A competitividade entre as empresas em qualquer ramo de mercado é um fator que impulsiona a adoção de novas tecnologias, que por serem mais recentes, trazem riscos de segurança ainda desconhecidos ou que por vezes são negligenciados. O emprego de plataformas de orquestração de contêineres foi observado como sendo uma das vertentes em órgãos da Administração Pública Federal e a preocupação com o aprimoramento da segurança nesses ambientes foi o principal motivador para a condução deste trabalho.

O trabalho desenvolvido apresentou um *framework* contendo uma arquitetura de referência para implementação de um HIDS voltado para plataformas de contêineres. Como principais contribuições desse *framework* destaca-se o seguinte:

- Realização da coleta remota de *system calls* em diferentes nós de um *cluster* de orquestração de contêineres, possibilitando o aprendizado de anomalias de maneira colaborativa;
- Redução do *overhead* de processamento causado pela detecção de intrusão nos nós de um *cluster* de orquestração de contêineres através de uma arquitetura de HIDS com componentes distribuídos;
- Arquitetura escalável implementada com ferramentas gratuitas em um ambiente corporativo emulado;
- Capacidade de construção de *datasets* de *system calls* próprios e possibilidade de compartilhamento com a comunidade;
- Geração de alertas de detecção de anomalias em aplicações para apoio ao SOC através da análise de *system calls*;
- Análise dos dados através de interface web contendo *datasets* e anomalias indexadas;
- Possibilidade de implementação de diferentes algoritmos de *machine learning* e abordagens para detecção de anomalias em *system calls* (frequência, sequência, argumentos e outros dados) visando uma maior eficácia na detecção;
- Capacidade de integração do *framework* com outras ferramentas, aprimorando a segurança colaborativa.

Através da experimentação realizada com um *dataset* de *system calls* público foi possível validar a viabilidade de implementação e funcionamento do *framework* proposto com ferramentas gratuitas em uma arquitetura funcional composta por cinco camadas. Técnicas e abordagens utilizadas em estudos do estado da arte no campo de pesquisa foram estudadas e aplicadas para desenvolvimento do *framework*. Mais do que a proposição teórica de um *framework*, o foco desde o princípio deste trabalho foi o funcionamento real de uma arquitetura e integração entre suas ferramentas, por esta razão, a topologia emulada no *software* GNS3 contendo um ambiente funcional.

Os resultados de detecção de ataques demostraram que a abordagem não-supervisionada de *machine learning* obteve desempenho promissor, sendo uma alternativa às técnicas supervisionadas. No que tange aos algoritmos de *machine learning* e detecção de anomalias, muitas otimizações podem ser realizadas em um universo de parâmetros e ajustes finos possíveis. Assim, o tempo se torna um fator limitante quando se trata da otimização e treinamento de modelos.

A possibilidade de visualização de *datasets* através de uma interface gráfica analítica é um diferencial do *framework*, facilitando o entendimento e análise dos dados através de filtragens, agregações, pesquisas e outras funcionalidades que podem contribuir com as atividades de especialistas e aprimoramento dos métodos de detecção. Além disso, através da análise dos alertas de anomalias detectadas, a equipe do SOC estará provida de informação adicional sobre a segurança de ambientes conteinerizados para correlação com dados de outras ferramentas e apoio na tomada de decisão e atuação.

A princípio, tinha-se o entendimento de que a lacuna de pesquisa relacionada à detecção de intrusão em tempo real poderia ser endereçada sem conhecimento profundo dos critérios e complexidade inerentes a essa problemática. Tal aspiração pode ser constatada na publicação resultante deste trabalho [13], mas que até o término desta pesquisa não pôde ser solucionada. Essa limitação, assim como outras questões encontradas no decorrer da pesquisa foram abordadas como escopo para trabalhos futuros. Ademais, ao final desta pesquisa, a percepção obtida aponta que um dos principais desafios relacionados à detecção de intrusão em plataformas de contêineres está ligado ao grande volume de dados envolvidos, se apresentando com um problema de Big Data.

Assim como encontrado em diversos trabalhos da literatura correlata, o campo de pesquisa envolvendo a detecção de intrusão em ambientes conteinerizados é vasto e carece de muitos avanços e experimentos que gerem insumos para novas pesquisas. A contribuição de trabalhos previamente realizados foram as bases para o desenvolvimento do *framework* proposto e arcabouço essencial para se alcançar os resultados obtidos neste trabalho. Nesse sentido, espera-se que o trabalho desenvolvido possa ser utilizado como referência adicional para implementação de Sistemas de Detecção de Intrusão voltados para ambientes de contêineres. Ademais, espera-se que trabalhos futuros tragam aprimoramentos ao *framework* proposto através da implementação de novos algoritmos, modificações na arquitetura e testes em ambientes produtivos.

6.2 TRABALHOS FUTUROS

Ainda que contribuições fruto deste trabalho tenham sido apresentadas, devido à complexidade e extensão do assunto abordado ao longo do desenvolvimento desta pesquisa, limitações e otimizações possíveis puderam ser observadas. Diante do exposto, ressalta-se que como um Sistema de Detecção de Intrusão, nenhuma ação automatizada é executada na ocorrência de um alerta. Deste modo, como aprimoramento ao *framework* ações de resposta automáticas podem ser elaboradas para interrupção da propagação e continuidade de um ataque no ambiente conteinerizado.

Uma eventual anomalia em uma sequência de *system calls* invocadas por um contêiner não provê informações de alto nível que levem imediatamente à origem do acesso causador de tal anomalia na aplicação.

Em outras palavras, não é possível informar o usuário responsável pelo comportamento anômalo na aplicação, o IP de origem ou dados que permitam a identificação do agente provocador da anomalia sem o cruzamento de dados oriundos de outras fontes de informações como logs de requisições ou de tráfego de rede para essa associação. Essa capacidade de análise e correlação dos dados de detecção com outras fontes de informação para rastreamento do acesso em alto nível poderá ser implementada para uma tomada de ação mais eficaz contra uma intrusão.

Um modelo de *machine learning* treinado para detecção de anomalias normalmente é específico ao comportamento de uma única aplicação. A dificuldade para se obter um bom desempenho de detecção para um contexto reduzido já não é uma tarefa trivial. Contudo, abordagens que envolvam a análise de comportamento e detecção de anomalias em múltiplas aplicações conteinerizadas que interagem entre si é uma área a ser pesquisada tendo em vista ser esta uma característica dos ambientes de contêiner existentes atualmente. Ainda, para a otimização do processo de treinamento e aprimoramento da detecção pelos modelos de *machine learning* ferramentas de processamento distribuído como o Spark, podem ser estudadas em conjunto com outras técnicas de aprendizado federado.

Conforme a classificação atual utilizada para categorizar os IDS existentes um IDS híbrido combina características de um IDS baseado em assinatura bem como detecção de anomalias. Assim, a identificação de sequências de *system calls* anômalas poderá fornecer insumos para a construção de assinaturas de ataques e colaborar para estudos visando a construção de um IDS híbrido. Nesse tipo de IDS, uma etapa de detecção através de assinaturas ocorre antes da análise de anomalias possibilitando a detecção de ataques conhecidos com menor custo computacional. Outra tendência que pode ser estudada é a construção de um IDS colaborativo, onde as capacidades de um HIDS e NIDS podem ser combinadas para o aprimoramento da detecção de intrusões.

A arquitetura proposta em cinca camadas e suas ferramentas tem por objetivo servir como uma referência para implementação de um HIDS voltado para plataformas de orquestração de contêineres, e sua configuração não é limitada à arquitetura proposta. Espera-se que a partir dessa versão inicial outras ferramentas e processos possam ser inseridos no *pipeline* de processamento visando o aprimoramento do IDS em cada cenário aplicado. Melhorias podem ser realizadas no agente utilizado para coleta de *system calls* e nos demais componentes de cada camada para aumento de desempenho, alta disponibilidade e automação de processos.

Devido à limitações de tempo, a experimentação realizada neste trabalho teve como escopo a avaliação e teste do *framework* proposto fazendo uso de um *dataset* de *system calls* público. Como trabalho futuro, pretende-se realizar outros experimentos com o *framework* e a topologia implementada para a geração e disponibilização de um *dataset* para um *software* disponível no Portal do Software Público Brasileiro, a saber, o Sistema Eletrônico de Informações (SEI). O referido sistema têm relevância no âmbito da Administração Pública Federal pela sua ampla utilização em inúmeros órgãos e um mecanismo de proteção contra ataques poderia ser útil a diversos órgãos que o utilizam. Cabe ressaltar que esse mesmo sistema foi alvo de ataques recentes em muitos órgãos por uma vulnerabilidade em um dos seus módulos.

Por fim, outros experimentos também poderão ser realizados na topologia construída para aplicações em execução no *cluster* Kubernetes configurado. Novos *datasets* poderão ser gerados e testes de penetração e ataques executados para avaliação de desempenho do IDS.

REFERÊNCIAS BIBLIOGRÁFICAS

1 SCARFONE, K.; MELL, P. *Guide to Intrusion Detection and Prevention Systems (IDPS)*. [S.l.]: Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, 2007.

2 EBERT, C.; GALLARDO, G.; HERNANTES, J.; SERRANO, N. DevOps. *IEEE Software*, IEEE Computer Society, v. 33, n. 3, p. 94–100, may 2016. ISSN 19374194.

3 HOFFMAN, P. *3 Cybersecurity of the Grid | The Resilience of the Electric Power Delivery System in Response to Terrorism and Natural Disasters: Summary of a Workshop | The National Academies Press*. 2013. Disponível em: <https://nap.nationalacademies.org/read/18535/chapter/4#11>. Acesso em: 30/12/2022.

4 HAT, R. *State of Kubernetes Security Report*. 2021.

5 CORPORATION, T. M. *Matrix - Enterprise | MITRE ATT&CK®*. 2022. Disponível em: <https://attack.mitre.org/matrices/enterprise/containers/#>. Acesso em: 29/12/2022.

6 A framework for designing a security operations centre (SOC). In: . [S.l.]: IEEE Computer Society, 2015. v. 2015-March, p. 2253–2262. ISBN 9781479973675. ISSN 15301605.

7 SARAYDARYAN, J.
Détection d'anomalies comportementales appliquée à la vision globale, 10 2008.

8 ADELEKE, O. Intrusion detection: Issues, problems and solutions. In: . [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2020. p. 397–402. ISBN 9781728172835.

9 GERON, A. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 2nd. ed. [S.l.]: O'Reilly Media, Inc., 2019. ISBN 1492032646.

10 KHRAISAT, A.; GONDAL, I.; VAMPLEW, P.; KAMRUZZAMAN, J. Survey of intrusion detection systems: techniques, datasets and challenges. *Cybersecurity*, Springer Science and Business Media B.V., v. 2, 12 2019. ISSN 25233246.

11 BERMAN, D. S.; BUCZAK, A. L.; CHAVIS, J. S.; CORBETT, C. L. A survey of deep learning methods for cyber security. *Information (Switzerland)*, MDPI AG, v. 10, 2019. ISSN 20782489.

12 SAYED, A.; ABED, O.; REED, J. H.; YANG, Y.; RAKHA, H. A.; AZAB, M. M.
Securing Cloud Containers through Intrusion Detection and Remediation, 2017.

13 ROCHA, S. L.; NZE, G. D. A.; MENDONçA, F. L. Lopes de. Intrusion detection in container orchestration clusters : A framework proposal based on real-time system call analysis with machine learning for anomaly detection. In: *2022 17th Iberian Conference on Information Systems and Technologies (CISTI)*. [S.l.: s.n.], 2022. p. 1–4.

14 CUI, P.; UMPHRESS, D. Towards unsupervised introspection of containerized application. In: . [S.l.]: Association for Computing Machinery, 2020. p. 42–51. ISBN 9781450389037.

15 MELL, P.; GRANCE, T. *The NIST Definition of Cloud Computing*. [S.l.]: Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, 2011.

16 SHU, R.; GU, X.; ENCK, W. A study of security vulnerabilities on docker hub. In: *CODASPY 2017 - Proceedings of the 7th ACM Conference on Data and Application Security and Privacy*. [S.l.]: Association for Computing Machinery, Inc, 2017. p. 269–280.

17 FLORA, J.; ANTUNES, N. Studying the applicability of intrusion detection to multi-tenant container environments. *Proceedings - 2019 15th European Dependable Computing Conference, EDCC 2019*, p. 133–136, 2019.

18 FLORA, J.; GONCALVES, P.; ANTUNES, N. Using attack injection to evaluate intrusion detection effectiveness in container-based systems. *Proceedings of IEEE Pacific Rim International Symposium on Dependable Computing, PRDC*, IEEE Computer Society, v. 2020-Decem, p. 60–69, 12 2020. ISSN 15410110.

19 KASHKOUSH, M.; CLANCY, C.; ABED, A.; AZAB, M. Resilient intrusion detection system for cloud containers. *International Journal of Communication Networks and Distributed Systems*, v. 24, p. 1, 01 2020.

20 LIU, M.; XUE, Z.; XU, X.; ZHONG, C.; CHEN, J. Host-based intrusion detection system with system calls: Review and future trends. *ACM Comput. Surv.*, Association for Computing Machinery, nov. 2018. ISSN 0360-0300.

21 ZHANG, X.; NIYAZ, Q.; JAHAN, F.; SUN, W. Early detection of host-based intrusions in linux environment. In: *2020 IEEE International Conference on Electro Information Technology (EIT)*. [S.l.: s.n.], 2020. p. 475–479.

22 ROHLING, M. M.; GRIMMER, M.; KREUBEL, D.; HOFFMANN, J.; FRANCZYK, B. Standardized container virtualization approach for collecting host intrusion detection data. In: . [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2019. p. 459–463. ISBN 9788395541605.

23 BRIDGES, R. A.; GLASS-VANDERLAN, T. R.; IANNACONE, M. D.; VINCENT, M. S.; CHEN, Q. A survey of intrusion detection systems leveraging host data. *ACM Computing Surveys*, Association for Computing Machinery, v. 52, 10 2019. ISSN 15577341.

24 TAJ, R.
A Machine Learning Framework for Host Based Intrusion Detection using System Call Abstraction, 2020.

25 BHARATHY, A. M. V.; UMAPATHI, N.; PRABAHARAN, S. An elaborate comprehensive survey on recent developments in behaviour based intrusion detection systems. 2019.

26 HICKMAN, A.; VANDEVEN, S. *Container Intrusions: Assessing the Efficacy of Intrusion Detection and Analysis Methods for Linux Container Environments*. [S.l.], 2018.

27 ABED, A. S.; CLANCY, C.; LEVY, D. S. Intrusion detection system for applications using linux containers. Springer International Publishing, v. 9331, 2015.

28 ABED, A. S.; CLANCY, T. C.; LEVY, D. S. Applying bag of system calls for anomalous behavior detection of applications in linux containers. *IEEE*, 2015.

29 KANG, D.-K.; FULLER, D.; HONAVAR, V. *Learning Classifiers for Misuse and Anomaly Detection Using a Bag of System Calls Represent ation*. 2005.

30 ATMOSPHERE. *Partners | ATMOSPHERE*. 2022. Disponível em: <https://atmosphere-eubrazil.eu/partner>. Acesso em: 30/03/2022.

31 SUBBA, B.; BISWAS, S.; KARMAKAR, S. Host based intrusion detection system using frequency analysis of n-gram terms. In: *TENCON 2017 - 2017 IEEE Region 10 Conference*. [S.l.: s.n.], 2017. p. 2006–2011.

32 SRINIVASAN, S.; KUMAR, A.; MAHAJAN, M.; SITARAM, D.; GUPTA, S. Probabilistic real-time intrusion detection system for docker containers. In: . [S.l.]: Springer Verlag, 2019. v. 969, p. 336–347. ISBN 9789811358258. ISSN 18650929.

33 BYRNES, J.; HOANG, T.; MEHTA, N. N.; CHENG, Y. A modern implementation of system call sequence based host-based intrusion detection systems. In: . [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2020. p. 218–225. ISBN 9781728185439.

34 FORREST, S.; HOFMEYR, S. A.; SOMAYAJI, A.; LONGSTAFF, T. A. *A Sense of Self for Unix Processes*. 1996.

35 HAIDER, W.; HU, J.; MOUSTAFA, N. Designing anomaly detection system for cloud servers by frequency domain features of system call identifiers and machine learning. In: . [S.l.]: Springer Verlag, 2018. v. 235, p. 137–149. ISBN 9783319907741. ISSN 18678211.

36 LIN, Y.; TUNDE-ONADELE, O.; GU, X. Cdl: Classified distributed learning for detecting security attacks in containerized applications. In: . [S.l.]: ICST, 2020. p. 179–188. ISBN 9781450388580. ISSN 21531633.

37 TIEN, C.; HUANG, T.; TIEN, C.; HUANG, T.; KUO, S. Kubanomaly: Anomaly detection for the docker orchestration platform with neural network approaches. *Engineering Reports*, Wiley, v. 1, 12 2019. ISSN 2577-8196.

38 GOLDBERG, R. P. Architecture of virtual machines. In: *AFIPS National Computer Conference*. [S.l.]: AFIPS Press/ACM, 1973. (AFIPS Conference Proceedings, v. 42), p. 309–318.

39 VMWARE, I. *O que é tecnologia de virtualização e máquina virtual? | VMware*. 2022. Disponível em: <https://www.vmware.com/br/solutions/virtualization.html>. Acesso em: 23/02/2022.

40 MORABITO, R.; KJäLLMAN, J.; KOMU, M. Hypervisors vs. lightweight virtualization: A performance comparison. In: . [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2015. p. 386–393. ISBN 9781479982189.

41 GOUDARZI, H.; PEDRAM, M. Energy-efficient virtual machine replication and placement in a cloud computing system. *Proceedings - 2012 IEEE 5th International Conference on Cloud Computing, CLOUD 2012*, p. 750–757, 2012.

42 MAVRIDIS, I.; KARATZA, H. Performance and overhead study of containers running on top of virtual machines. In: . [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2017. v. 2, p. 32–38. ISBN 9781538630341.

43 ZHANG, Q.; LIU, L.; PU, C.; DOU, Q.; WU, L.; ZHOU, W. A comparative study of containers and virtual machines in big data environment. 7 2018.

44 WANG, B.; SONG, Y.; CUI, X.; CAO, J. Performance comparison between hypervisor and container-based virtualizations for cloud users. 2017.

45 SOUPPAYA, M.; MORELLO, J.; SCARFONE, K. *NIST-SP-800-190-Application container security guide*. 2017.

46 MATTETTI, M.; SHULMAN-PELEG, A.; ALLOUCHE, Y.; CORRADI, A.; DOLEV, S.; FOSCHINI, L. Securing the infrastructure and the workloads of linux containers. p. 559–567, 2015.

47 AUTHORS, T. K. *Kubernetes | Producion-Grade Container Orchestration*. 2022. Disponível em: <https://www.kubernetes.io/>. Acesso em: 23/02/2022.

48 GROSSMAN, R. L. *The Case for Cloud Computing*. 2009.

49 MIYACHI, C. What is "cloud"? it is time to update the nist definition? *IEEE Cloud Computing*, Published by the IEEE Computer Society, v. 5, p. 6–11, 2018. ISSN 23256095.

50 BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software*, IEEE Computer Society, v. 33, n. 3, p. 42–52, may 2016. ISSN 07407459.

51 AFTERGOOD, S. Cybersecurity: The cold war online. *Nature*, v. 547, p. 30–31, 07 2017.

52 SECURITY, I. *X-Force Threat Intelligence Index 2022 Full Report*. 2022.

53 SOPHOSLABS; RESPONSE, S. M. T.; RESPONSE, S. R.; SOPHOSAI. *Sophos 2022 Threat Report - Interrelated threats target an interdependent world*. 2021.

54 PESCATORE, J. *SANS 2021 Top New Attacks and Threat Report*. 2021.

55 ENISA. Enisa threat landscape 2022. 2022. Disponível em: <www.enisa.europa.eu.> Acesso em: 29/12/2022.

56 NATHANS, D. *Designing and Building Security Operations Center*. [S.l.]: Elsevier Science, 2014. ISBN 9780128010969.

57 BIDOU, R. Security Operation Center Concepts & Implementation. 2005.

58 VIELBERTH, M.; BOHM, F.; FICHTINGER, I.; PERNUL, G. Security operations center: A systematic study and open challenges. *IEEE Access*, Institute of Electrical and Electronics Engineers Inc., 2020. ISSN 21693536.

59 QAZI, E.-u.-H.; IMRAN, M.; HAIDER, N.; SHOAIB, M.; RAZZAK, I. An intelligent and efficient network intrusion detection system using deep learning. *Computers & Electrical Engineering*, v. 99, p. 107764, apr 2022. ISSN 00457906.

60 ZHENG, X.; IEEE ITSS; Institute of Electrical and Electronics Engineers. A user-centric machine learning framework for cyber security operations center. 2017.

61 LAKSHMINARAYANA, D. H.; PHILIPS, J.; TABRIZI, N. A survey of intrusion detection techniques. In: *Proceedings - 18th IEEE International Conference on Machine Learning and Applications, ICMLA 2019*. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2019. p. 1122–1129. ISBN 9781728145495.

62 AGRAWAL, S.; AGRAWAL, J. Survey on anomaly detection using data mining techniques. In: *Procedia Computer Science*. [S.l.]: Elsevier B.V., 2015. v. 60, n. 1, p. 708–713. ISSN 18770509.

63 DUA, S.; DU, X. *Data Mining and Machine Learning in Cybersecurity*. 1st. ed. USA: Auerbach Publications, 2011. ISBN 1439839425.

64 BUCZAK, A. L.; GUVEN, E. A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection. *IEEE Communications Surveys and Tutorials*, Institute of Electrical and Electronics Engineers Inc., v. 18, n. 2, p. 1153–1176, apr 2016. ISSN 1553877X.

65 XIN, Y.; KONG, L.; LIU, Z.; CHEN, Y.; LI, Y.; ZHU, H.; GAO, M.; HOU, H.; WANG, C. Machine Learning and Deep Learning Methods for Cybersecurity. *IEEE Access*, Institute of Electrical and Electronics Engineers Inc., v. 6, p. 35365–35381, may 2018. ISSN 21693536.

66 HOCHREITER, S.; SCHMIDHUBER, J. U. Long short-term memory. 1997.

67 GUMUSBAS, D.; YLDRM, T.; GENOVESE, A.; SCOTTI, F. A comprehensive survey of databases and deep learning methods for cybersecurity and intrusion detection systems. *IEEE Systems Journal*, Institute of Electrical and Electronics Engineers (IEEE), p. 1–15, 5 2020. ISSN 1932-8184.

68 SYSDIG, I. *Security Tools for Containers, Kubernetes & Cloud*. 2022. Disponível em: <https://sysdig.com>. Acesso em: 10/08/2022.

69 B.V., E. *Elasticsearch: The Official Distributed Search & Analytics Engine | Elastic*. 2022. Disponível em: <https://www.elastic.co/elasticsearch/>. Acesso em: 03/09/2022.

70 ABADI, M.; AGARWAL, A.; BARHAM, P.; BREVDO, E.; CHEN, Z.; CITRO, C.; CORRADO, G. S.; DAVIS, A.; DEAN, J.; DEVIN, M.; GHEMAWAT, S.; GOODFELLOW, I.; HARP, A.; IRVING, G.; ISARD, M.; JIA, Y.; JOZEFOWICZ, R.; KAISER, L.; KUDLUR, M.; LEVENBERG, J.; MANé, D.; MONGA, R.; MOORE, S.; MURRAY, D.; OLAH, C.; SCHUSTER, M.; SHLENS, J.; STEINER, B.; SUTSKEVER, I.; TALWAR, K.; TUCKER, P.; VANHOUCKE, V.; VASUDEVAN, V.; VIéGAS, F.; VINYALS, O.; WARDEN, P.; WATTENBERG, M.; WICKE, M.; YU, Y.; ZHENG, X. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015.

71 TEAM, K. *Keras: the Python deep learning API*. 2022. Disponível em: <https://keras.io/>. Acesso em: 03/09/2022.

72 CHETLUR, S.; WOOLLEY, C.; VANDERMERSCH, P.; COHEN, J.; TRAN, J.; CATANZARO, B.; SHELHAMER, E. cudnn: Efficient primitives for deep learning. 10 2014.

73 TAN, K.; MAXION, R. "why 6?"defining the operational limits of stide, an anomaly-based intrusion detector. p. 188–201, 2002.

74 LASZKA, A.; ABBAS, W.; SASTRY, S.; VOROBEYCHIK, Y.; KOUTSOUKOS, X. Optimal thresholds for intrusion detection systems. 04 2016.

75 WORLDWIDE, L. S. *The software that empowers network professionals*. 2022. Disponível em: <https://gns3.com>. Acesso em: 10/08/2022.

76 FENCING, L. E. S. *pfSense - World's Most Trusted Open Source Firewal*. 2022. Disponível em: <https://www.pfsense.org/>. Acesso em: 03/09/2022.

77 ROCHA, S. L. *s4vio/rcids-public: Public repository for RCIDS project*. 2022. Disponível em: <https://github.com/s4vio/rcids-public/>. Acesso em: 03/09/2022.

78 BERGSTRA, J.; YAMINS, D.; COX, D. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. p. 13–19, 01 2013.

79 RAUT, M.; DHAVALE, S.; SINGH, A.; MEHRA, A. Insider threat detection using deep learning: A review. *Proceedings of the 3rd International Conference on Intelligent Sustainable Systems, ICISS 2020*, Institute of Electrical and Electronics Engineers Inc., p. 856–863, dec 2020.

APÊNDICES

I.1 CONFIGURAÇÃO DO FILEBEAT

```
# ============================== Filebeat inputs ===============================

filebeat.inputs:

# Each - is an input. Most options can be set at the input level, so
# you can use different inputs for various configurations.
# Below are the input specific configurations.

- type: log

  # Change to true to enable this input configuration.
  enabled: true

  # Paths that should be crawled and fetched. Glob based paths.
  paths:
    #- /var/log/*.log
        - /var/log/sysdig/*.log
    #- c:\programdata\elasticsearch\logs\*

  # Exclude lines. A list of regular expressions to match. It drops the lines that
  ↪  are
  # matching any regular expression from the list.
  #exclude_lines: ['^DBG']

  # Include lines. A list of regular expressions to match. It exports the lines
  ↪  that are
  # matching any regular expression from the list.
  #include_lines: ['^ERR', '^WARN']

  # Exclude files. A list of regular expressions to match. Filebeat drops the files
  ↪  that
  # are matching any regular expression from the list. By default, no files are
  ↪  dropped.
  #exclude_files: ['.gz$']

  # Optional additional fields. These fields can be freely picked
  # to add additional information to the crawled log files for filtering
  #fields:
  #  level: debug
  #  review: 1

  ### Multiline options

  # Multiline can be used for log messages spanning multiple lines. This is common
  # for Java Stack Traces or C-Line Continuation

  # The regexp Pattern that has to be matched. The example pattern matches all
  ↪  lines starting with [
  #multiline.pattern: ^\[
```

```
  # Defines if the pattern set under pattern should be negated or not. Default is
  ↪  false.
  #multiline.negate: false

  # Match can be set to "after" or "before". It is used to define if lines should
  ↪  be append to a pattern
  # that was (not) matched before or after or as long as a pattern is not matched
  ↪  based on negate.
  # Note: After is the equivalent to previous and before is the equivalent to to
  ↪  next in Logstash
  #multiline.match: after

# filestream is an experimental input. It is going to replace log input in the
↪  future.
- type: filestream

  # Change to true to enable this input configuration.
  enabled: false

  # Paths that should be crawled and fetched. Glob based paths.
  paths:
    - /var/log/*.log
    #- c:\programdata\elasticsearch\logs\*

  # Exclude lines. A list of regular expressions to match. It drops the lines that
  ↪  are
  # matching any regular expression from the list.
  #exclude_lines: ['^DBG']

  # Include lines. A list of regular expressions to match. It exports the lines
  ↪  that are
  # matching any regular expression from the list.
  #include_lines: ['^ERR', '^WARN']

  # Exclude files. A list of regular expressions to match. Filebeat drops the files
  ↪  that
  # are matching any regular expression from the list. By default, no files are
  ↪  dropped.
  #prospector.scanner.exclude_files: ['.gz$']

  # Optional additional fields. These fields can be freely picked
  # to add additional information to the crawled log files for filtering
  #fields:
  #  level: debug
  #  review: 1

# ============================== Filebeat modules ==============================

filebeat.config.modules:
  # Glob pattern for configuration loading
  path: ${path.config}/modules.d/*.yml
```

```
  # Set to true to enable config reloading
  reload.enabled: false

  # Period on which files under path should be checked for changes
  #reload.period: 10s

# ================================== Outputs ===================================

# Configure what output to use when sending the data collected by the beat.

# ------------------------------ Redis Output ------------------------------
output.redis:
  hosts: ["192.168.201.4"]
  key: "k8s"
  password: ## editado ##

# ================================= Processors =================================
processors:
  - add_host_metadata:
      when.not.contains.tags: forwarded
  - add_cloud_metadata: ~
  - add_docker_metadata: ~
  - add_kubernetes_metadata: ~
```

I.2 CONFIGURAÇÃO DO LOGSTASH

```
input {
    redis {
      host => "192.168.201.4"
      type => "list"
      key => "app_1"
      password => ## editado ##
    }
}


filter {
  grok {
    match =>  {"message" => "%{NUMBER:event_number} %{TIMESTAMP_ISO8601:timestamp}
    ↪  (?<event_direction>(<|>)) %{NOTSPACE:systemcall_type}
    ↪  %{NOTSPACE:systemcall} %{NUMBER:latency} %{NUMBER:return_value} [
    ↪  ]?%{GREEDYDATA:args}?"}
  }

  if [args] {

    mutate {
      gsub => ["args", "\s?(\S+=)", ";;;\1"]
      gsub => ["args", ".$", ""]
    }

    ruby {
      code => '
        args_l = event.get("args")
        length = args_l.length
        event.set("args_length", length)
        args_t = event.get("args").split(";;;")
        n = args_t.length
        #puts "#{args_t}"
        #puts "#{n}"
        event.set("argc", n - 1)
        x = 0
        y = 1
        while y < n do
          #puts "#{args_t[y]}"
          event.set("[argv][#{x}]", args_t[y])
          y += 1
          x += 1
        end
      '
    }
  } else {
      mutate {
        add_field => ["argc", "0"]
        add_field => ["args_length", "0"]
```

```
      }
    }

  date {
   match => ["timestamp" , "ISO8601"]
   timezone => "UTC"
   target => "@timestamp"
  }
}

  mutate {
    remove_field => ["@version", "host", "path", "args"]
    gsub => [
      "event_direction", ">", "1",
      "event_direction", "<", "0"
    ]
  }
}


output {
  if "_grokparsefailure" not in [tags] {
    elasticsearch {
        hosts  => [ "192.168.201.2:9200" ]
        user => 'elastic'
        password => ## editado ##
        index => "app1-%{+YYYY.MM.dd_HH}"
    }
  }
}
```

I.3 CONFIGURAÇÃO DO ELASTICSEARCH

```
---
# https://www.elastic.co/guide/en/elasticsearch/reference/current/important-setting
↪  s.html}

cluster.name: gns3-cluster
node.name: node-01

network.host: 0.0.0.0

path:
  data: /usr/share/elasticsearch/data/
  logs: /usr/share/elasticsearch/logs/

discovery.type: single-node

bootstrap.memory_lock: true

# https://www.elastic.co/guide/en/elasticsearch/reference/current/setup-xpack.html
# https://www.elastic.co/guide/en/elasticsearch/reference/current/security-minimal-
↪  setup.html

xpack.license.self_generated.type: basic
xpack.security.enabled: true
xpack.monitoring.collection.enabled: true
```

I.4 TEMPLATE DE UM ÍNDICE NO ELASTICSEARCH

```
{
  "template": {
    "settings": {
      "index": {
        "lifecycle": {
          "name": "rcids",
          "rollover_alias": "app1-1h"
        },
        "number_of_replicas": "0",
        "sort": {
          "field": "timestamp",
          "order": "asc"
        }
      }
    },
    "mappings": {
      "dynamic": "true",
      "dynamic_date_formats": [
        "strict_date_optional_time",
        "yyyy/MM/dd HH:mm:ss Z||yyyy/MM/dd Z"
      ],
      "dynamic_templates": [
        {
          "argv_as_keyword": {
            "match": "argv.*",
            "match_mapping_type": "string",
            "mapping": {
              "type": "keyword"
            }
          }
        }
      ],
      "date_detection": true,
      "numeric_detection": false,
      "properties": {
        "argc": {
          "type": "integer"
        },
        "args_length": {
          "type": "long",
          "ignore_malformed": false,
          "coerce": true
        },
        "event_direction": {
          "type": "integer"
        },
        "event_number": {
          "type": "long",
          "ignore_malformed": false,
```

```
          "coerce": true
        },
        "latency": {
          "type": "long",
          "ignore_malformed": false,
          "coerce": true
        },
        "message": {
          "type": "text"
        },
        "return_value": {
          "type": "long"
        },
        "systemcall": {
          "type": "integer",
          "ignore_malformed": false,
          "coerce": true
        },
        "systemcall_type": {
          "type": "integer",
          "ignore_malformed": false,
          "coerce": true
        },
        "timestamp": {
          "type": "date_nanos"
        }
      }
    },
    "aliases": {}
  }
}
```

I.5 ARQUIVO DE FUNÇÕES PYTHON

```
from operator import truth
import numpy as np
import pandas as pd

from elasticsearch_dsl import Search
from sklearn.metrics import *


# Creating function for read data from Elasticsearch
def read_from_elastic(index, es):
    s = Search().using(es).index(index)
    s_fields = s.source(["timestamp", "systemcall_type", "systemcall", "latency",
    ↪  "return_value", "args_length", "argc"])
    df_from_elastic = pd.DataFrame([hit.to_dict() for hit in s_fields.scan()])
    # Transforming field "timestamp" to pandas dtype: datetime64[ns, UTC]
    df_from_elastic['timestamp'] = pd.to_datetime(df_from_elastic['timestamp'],
    ↪  utc=True, infer_datetime_format=True)
    # Sorting dataframe by timestamp field
    df_from_elastic.sort_values(by='timestamp', inplace=True, ignore_index=True)
    # Reordering dataframe columns
    df_from_elastic = df_from_elastic[['systemcall_type', 'systemcall', 'latency',
    ↪  'return_value', 'args_length', 'argc']]
    return df_from_elastic


# Defining function to create array with sliding windows
def sliding_window(X, window_size):
    X1 = []
    for i in range(len(X) - window_size+1):     #Range(9)   --> print de 0 a 8
        t = X.iloc[i:(i + window_size)].values  #iloc(0:9)  --> seleciona de 0 a 8
        X1.append(t)
    return np.array(X1)


# Defining function to create array with sliding windows with steps = window_size
def sliding_window_fast(X, window_size):
    X1 = []
    i = 0
    j = int(len(X) / window_size)                   # Qtd de windows
    r = (len(X) % window_size)                      # Resto
    if (r == 0):
        for k in range(j):                          # Range(9)   --> print de 0 a 8
            t = X.iloc[i:(i + window_size)].values  # iloc(0:9)  --> seleciona de 0
            ↪  a 8
            X1.append(t)
            i = i + window_size                     # Steps do tamanho do
            ↪  window_size
    else:                                           # Se qtd de windows não for
    ↪  inteira (resto != 0)
```

```
        for k in range(int((j))):
            t = X.iloc[i:(i + window_size)].values
            X1.append(t)
            i = i + window_size
        t = X.iloc[(i - (window_size - r)):((i - (window_size - r)) +
        ↪  window_size)].values    # Repete valores na última janela até completar
        ↪  window_size
        X1.append(t)
    return np.array(X1)



# Defining function to create dataframe with windows losses
def window_loss(w, threshold):
    wt = pd.DataFrame(columns = ['window_number', 'loss', 'anomaly', 'real'])
    window_number = []
    anomalies = []
    for i in range(len(w)):
        window_number.append(i)
        anomalies.append(w['Loss_mae'][i] > threshold)
    wt['window_number'] = window_number
    wt['loss'] = w['Loss_mae']
    wt['anomaly'] = anomalies
    wt['real'] = w['real']
    return (wt)


# Defining function for tunable threshold
# Originally proposed by: Cui, P., Umphress, D. (2020). Towards unsupervised
↪  introspection of containerized application. ACM International Conference
↪  Proceeding Series, 42-51. https://doi.org/10.1145/3442520.3442530
def tunable_threshold(df_train_loss, confidence_levels):
    T = pd.DataFrame(columns=confidence_levels)
    Tmax = np.round(df_train_loss['Loss_mae'].max(), 4)
    for i in confidence_levels:
        Tc = Tmax
        while  (np.round(df_train_loss[df_train_loss['Loss_mae'] <
        ↪  Tc]['Loss_mae'].count() / df_train_loss['Loss_mae'].count(), 3) > i):
            Tc = Tc * 0.99
        T.at[0,i] = np.round(Tc, 4)
    return T

def metrics(df_results):
    # Confusion Matrix
    cm = confusion_matrix(df_results['real'], df_results['anomaly'])

   # Primary Results
    tn = cm[0,0]
    fp = cm[0,1]
    fn = cm[1,0]
    tp = cm[1,1]
```

```
    # Accuracy
    accuracy = (tp + tn) / (tp + tn + fp + fn)

    # Precision = tp / (tp + fp)
    precision = precision_score(df_results['real'], df_results['anomaly'])

    # TPR ou Recall =  tp / (tp + fn)
    tpr = recall_score(df_results['real'], df_results['anomaly'])

    # NPV = tn / (tn + fn)
    npv = tn / (tn + fn)

    # TNR ou Especificidade = tn / (tn + fp)
    tnr = tn / (tn + fp)

    # FPR ou FAR = fp / (tn + fp)
    fpr = fp / (tn + fp)

    # F1 Score = 2 * (precision * tpr) / (precision + tpr)
    f1 = f1_score(df_results['real'], df_results['anomaly'])

    # AUC (ROC Curve)
    fpr_roc, tpr_roc, thresholds_roc = roc_curve(df_results['real'],
    ↪  df_results['anomaly'])
    roc_auc = auc(fpr_roc, tpr_roc)

    return cm, accuracy, precision, tpr, npv, tnr, fpr, f1, roc_auc
```

I.6 CÓDIGO EM JUPYTER NOTEBOOK USADO PARA TREINAMENTO DO MODELO NO EXPERIMENTO

```
# In[ ]:


import sys
assert sys.version_info >= (3, 5)

import numpy as np
np.set_printoptions(suppress=True) #prevent numpy exponential

import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import os

#from pathlib import Path
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, Normalizer, MinMaxScaler,
↪  OneHotEncoder, OrdinalEncoder
from sklearn.model_selection import train_test_split

import tensorflow as tf
import tensorflow_datasets as tfds
from tensorflow import keras
assert tf.__version__ >= "2.0"

#from keras import optimizers, Sequential, metrics
from elasticsearch import Elasticsearch
from keras.models import Sequential, save_model, load_model
from keras.layers import LSTM, Dense, RepeatVector, TimeDistributed, Conv1D
from tensorflow.compat.v1.keras.layers import CuDNNLSTM
from keras.callbacks import CSVLogger, ModelCheckpoint, EarlyStopping,
↪  LearningRateScheduler
from ipynb.fs.full.rcids_functions import *

pd.set_option('display.max_colwidth', None)
pd.set_option('display.max_rows', None)
np.set_printoptions(suppress=True) #prevent numpy exponential
pd.set_option('display.float_format', lambda x: '%.4f' % x) #prevent scientific
↪  notation in pandas


# ### Reading from Elasticsearch

# In[ ]:


# Test conectivity with Elasticsearch
es = Elasticsearch(host="192.168.201.2", http_auth=("elastic","7s9w1sbwjcivorj6"))
```

```
es.info(pretty=True)


# In[ ]:


# Defining elasticsearch indice to read from
index = "proc-public-benign"

# Counting number of documents in index
n_docs = es.count(index=index)
print("Número de documentos no índice", index, "-->", n_docs['count'])


# In[ ]:


# Creating dataset for trainning
df_benign_data = read_from_elastic(index, es)


# In[ ]:


# Excluding timestamp column
df_benign_data.drop(['timestamp'], axis=1, inplace=True)


# ### Parameters

# In[ ]:


# Defining window_size, n_feature and normalization function
window_size = 6
n_features = df_benign_data.shape[1]
norm_function = "mm"                  # std (StandardScaler), norm (Normalizer), mm
↪  (MinMaxScaler)


# ### Benign HashDB

# In[ ]:


hashdb_name = "df_proc_benign_hashdb_wz" + str(window_size) + "_ft" +
↪  str(n_features) + "_" + str(norm_function) + ".pkl"


# In[ ]:
```

```
if os.path.isfile("pkl/" + hashdb_name):
    # Loading existent df from disk
    df_benign_hashdb = pd.read_pickle("pkl/" + hashdb_name)
else:
    # Normalizing data
    mm = MinMaxScaler()
    mm_benign = mm.fit(df_benign_data)
    benign_data = mm_benign.transform(df_benign_data)

    # Creating 3D array for train data
    # For an LSTM Autoencoder the shape of input has to be of the format: n_samples
    ↪  x window_size x n_features
    benign_data_wz = pd.DataFrame(benign_data)
    benign_data_wz = sliding_window(benign_data_wz, window_size)

    # Coverting to 2d pandas df
    benign_data_wz_2d = benign_data_wz.reshape(benign_data_wz.shape[0],
    ↪  benign_data_wz.shape[1] * benign_data_wz.shape[2])
    df_benign_data_wz_2d = pd.DataFrame(benign_data_wz_2d)
    # Calculating rows (windows) hash
    df_benign_data_hash = df_benign_data_wz_2d.apply(lambda x:
    ↪  hash(tuple(x)).to_bytes(8, "big", signed=True).hex(), axis=1)
    # Removing duplicates
    df_benign_hashdb = pd.DataFrame(df_benign_data_hash.unique())
    # Saving df to disk
    df_benign_hashdb.to_pickle("pkl/" + hashdb_name)

print(hashdb_name + " size: ")
df_benign_hashdb.shape[0]


# ## Splitting Train / Test

# In[ ]:


df_train_data, df_test_data = train_test_split(df_benign_data, test_size=0.2,
↪  shuffle=False)


# ## Pre-processing the data

# ### Training data

# In[ ]:


# Normalizing data
mm = MinMaxScaler()
mm_train = mm.fit(df_train_data)
train_data = mm.transform(df_train_data)
```

```
print("Train data numpy.ndarray shape:", train_data.shape)


# In[ ]:


# Creating 3D array for train data
# For an LSTM Autoencoder the shape of input has to be of the format: n_samples x
↪  window_size x n_features
train_data_wz = pd.DataFrame(train_data)
train_data_wz = sliding_window(train_data_wz, window_size)


# ## Creating Tensorflow datasets

# ### Trainning dataset

# In[ ]:


# Train dataset
ds_train_full = tf.data.Dataset.from_tensor_slices(train_data_wz)


# In[ ]:


ds_train = ds_train_full.take(0.95 * ds_train_full.cardinality().numpy())
ds_validation = ds_train_full.take(0.05 * ds_train_full.cardinality().numpy())


# In[ ]:


ds_train = ds_train.map(lambda x: (x, x))
ds_train_batch = ds_train.batch(1024).cache().prefetch(tf.data.AUTOTUNE)


# In[ ]:


ds_validation = ds_validation.map(lambda x: (x, x))
ds_validation_batch = ds_validation.batch(1024).cache().prefetch(tf.data.AUTOTUNE)


# ## Defining and training the model

# In[ ]:


model_name = 'BEST-tfds_lstm_160_64_24_conv1d_relu_5_bn_tahn_wz' + str(window_size)
↪  + '_ft' + str(n_features) + '_' + str(norm_function)
```

```
model_name


# In[ ]:


model = Sequential()

# Conv1D
model.add(keras.layers.Conv1D(filters=n_features, kernel_size=window_size,
↪  strides=1, padding="same", activation="relu", input_shape=(window_size,
↪  n_features)))

# Encoder
model.add(CuDNNLSTM(160, kernel_initializer='he_normal', return_sequences=True))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation('tanh'))

model.add(CuDNNLSTM(64, kernel_initializer='he_normal', return_sequences=True))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation('tanh'))

model.add(CuDNNLSTM(24, kernel_initializer='he_normal', return_sequences=False))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation('tanh'))

model.add(RepeatVector(window_size))

# Decoder
model.add(CuDNNLSTM(24, kernel_initializer='he_normal', return_sequences=True))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation('tanh'))

model.add(CuDNNLSTM(64, kernel_initializer='he_normal', return_sequences=True))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation('tanh'))

model.add(CuDNNLSTM(160, kernel_initializer='he_normal', return_sequences=True))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation('tanh'))

model.add(TimeDistributed(Dense(n_features)))

model.compile(loss='mae', optimizer='nadam', metrics=['accuracy'])


# In[ ]:


# Trainning parameters
train_log = CSVLogger('log-' + str(model_name) + '.log', separator=',', append=True)
```

```
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=15,
↪  mode='min', min_delta=0.001, verbose=1)
learning_rate = keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=10)
mc = tf.keras.callbacks.ModelCheckpoint(filepath='model-' + str(model_name) +
↪  '.h5', monitor='val_loss', mode='min', save_best_only=True)


# In[ ]:


# Train model
model.fit(ds_train_batch, epochs=200, shuffle=False, callbacks=[train_log,
↪  early_stopping, learning_rate, mc], validation_data=ds_validation_batch)


# In[ ]:


# Saving/Loading the model
filepath = 'model-' + str(model_name) + '.h5'
#save_model(model, filepath)
model = load_model(filepath, compile=True)


# In[ ]:


# Reading model training history
df_history = pd.read_csv('log-' + str(model_name) + '.log', sep=',',
↪  engine='python')


# ## Loss distribution for training data

# In[ ]:


# Plotting the loss distribution
plot = sns.displot(data=df_history['val_loss'], kind='kde', color='blue', height=5,
↪  aspect=2)
plot.set_axis_labels("Validation Loss", "Density")
plot.set(title='Training Validation Loss Distribution')


# ## Defining the Loss Threshold

# ### Test data

# In[ ]:


# Normalizing data
```

```
mm = MinMaxScaler()
mm_test = mm.fit(df_train_data) # Fit deve ser feito com dados de treinanento
test_data = mm_test.transform(df_test_data) # Apenas transform nos dados de teste

print("Test data numpy.ndarray shape:", test_data.shape)


# In[ ]:


# Creating 3D array for train data
# For an LSTM Autoencoder the shape of input has to be of the format: n_samples x
↪  window_size x n_featuress
test_data_wz = pd.DataFrame(test_data)
test_data_wz = sliding_window(test_data_wz, window_size)


# ### Tensorflow test dataset

# In[ ]:


# Test dataset
ds_test = tf.data.Dataset.from_tensor_slices(test_data_wz)


# In[ ]:


ds_test = ds_test.map(lambda x: (x, x))
ds_test_batch = ds_test.batch(1024).cache().prefetch(tf.data.AUTOTUNE)


# ### Predicting test data using the model

# In[ ]:


# Predicting values using the trained model
pred = model.predict(ds_test_batch)


# In[ ]:


# Reshaping array with predictions to 2D dataframe (column 2 x column 3)
#X_pred.shape #--> (samples - window_size, window_size, n_features)
pred = pred.reshape(pred.shape[0], pred.shape[1] * pred.shape[2])
df_pred = pd.DataFrame(pred)


# In[ ]:
```

```
# Reshaping array with real data to 2D dataframe (column 2 x column 3)
#X_test.shape # --> (samples - window_size, window_size, n_features)
test = test_data_wz.reshape(test_data_wz.shape[0], test_data_wz.shape[1] *
↪  test_data_wz.shape[2])
df_test = pd.DataFrame(test)


# ### Calculating the loss

# In[ ]:


# Calculating test loss with MAE (Mean Absolute Error)
df_test_loss = pd.DataFrame(index=df_pred.index)
df_test_loss['Loss_mae'] = tf.metrics.MAE(df_test, df_pred)


# ### Defining the loss threshold

# In[ ]:


df_test_loss['Loss_mae'].describe()


# In[ ]:


# Defining threshold based on the training loss
#loss_threshold = np.round(df_train_loss.quantile([.75]).values[0][0], 4)
df_test_loss_mean = df_test_loss['Loss_mae'].values.mean()
df_test_loss_std = df_test_loss['Loss_mae'].values.std()
loss_threshold_mean_std = np.round(df_test_loss_mean + df_test_loss_std, 4)
loss_threshold_max = np.round(df_test_loss.values.max(), 4)
loss_threshold_percentile = np.round(np.percentile(df_test_loss['Loss_mae'].values,
↪  99), 4)
print("Threshold baseado na maior loss durante os testes --> ", loss_threshold_max)
print("Threshold calculado através da média + desvio padrão --> ",
loss_threshold_mean_std)
print("Threshold baseado no percentil de 99 do loss durante os testes --> ",
↪  loss_threshold_percentile)

thresholds = loss_threshold_mean_std, loss_threshold_max, loss_threshold_percentile


# In[ ]:


# Defining confidence levels for threshold adjustment
confidence_levels =  [1, 0.995, 0.99, 0.98, 0.97, 0.96, 0.95]
```

```
# Calling function and creating dataframe with thresholds
thresholds = tunable_threshold(df_test_loss, confidence_levels)

# Printing threshold per confidence level
print("--- Threshold por intervalos de confiança ---")
for i in confidence_levels:
    print("Intervalo de confiança [", i, "] --> ", thresholds.iloc[0][i])


# In[ ]:


filepath = 'model-' + str(model_name) + '.h5'

df_name = 'df_proc_thresholds.pkl'

if os.path.isfile('pkl/' + df_name):
    # Loading existent df from disk
    df_thresholds = pd.read_pickle('pkl/' + df_name)

    for i in confidence_levels:
        # Adding last execution results in to dataframe
        df_thresholds.loc[df_thresholds.shape[0]] = [filepath, i,
        ↪  thresholds.iloc[0][i]]

    # Saving df to disk
    df_thresholds.to_pickle('pkl/' + df_name)

else:
    # Defining dataframe columns
    df_thresholds = pd.DataFrame(columns=["Model", "Confidence_Level", "Threshold"])

    for i in confidence_levels:
    # Adding last execution results in to dataframe
        df_thresholds.loc[df_thresholds.shape[0]] = [filepath, i,
        ↪  thresholds.iloc[0][i]]

    # Saving df to disk
    df_thresholds.to_pickle('pkl/' + df_name)


# In[ ]:


df_thresholds.sort_values(['Model']).groupby(['Model']).value_counts()


# In[ ]:
```

```
loss_threshold = float(input("Escolha um dos thresholds:
↪  \n{}".format(thresholds.to_string(header=None, index=False))))
print("Threshold escolhido --> ", loss_threshold)
```

I.7 CÓDIGO EM JUPYTER NOTEBOOK USADO PARA DETECÇÃO DE ANOMALIAS NO EXPERIMENTO

```
# In[ ]:


import sys
assert sys.version_info >= (3, 5)

import numpy as np
np.set_printoptions(suppress=True) #prevent numpy exponential

import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import os

#from pathlib import Path
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, Normalizer, MinMaxScaler,
↪  OneHotEncoder, OrdinalEncoder
from sklearn.model_selection import train_test_split

import tensorflow as tf
import tensorflow_datasets as tfds
from tensorflow import keras
assert tf.__version__ >= "2.0"

#from keras import optimizers, Sequential, metrics
from elasticsearch import Elasticsearch
from espandas import Espandas
from keras.models import Sequential, load_model
from keras.layers import LSTM, Dense, RepeatVector, TimeDistributed, Conv1D
from tensorflow.compat.v1.keras.layers import CuDNNLSTM
from keras.callbacks import CSVLogger, ModelCheckpoint, EarlyStopping
from ipynb.fs.full.rcids_functions import *

pd.set_option('display.max_colwidth', None)
pd.set_option('display.max_rows', None)
np.set_printoptions(suppress=True) #prevent numpy exponential
pd.set_option('display.float_format', lambda x: '%.4f' % x) #prevent scientific
↪  notation in pandas


# ### Reading from Elasticsearch

# In[ ]:


# Test conectivity with Elasticsearch
es = Elasticsearch(host="192.168.201.2", http_auth=("elastic","7s9w1sbwjcivorj6"))
```

```
#es.info(pretty=True)


# In[ ]:


# Defining elasticsearch indice to read from

attack_name = "bruteforcelogin"
#attack_name = "dockerescape"
#attack_name = "maliciousscript"
#attack_name = "meterpreter"
#attack_name = "remoteshell"
#attack_name = "sqlinjection"
#attack_name = "sqlmisbehavior"

index = "proc-public-" + str(attack_name)

# Counting number of documents in index
#n_docs = es.count(index=index)
#print("Número de documentos no índice", index, "-->", n_docs['count'])


# In[ ]:


# Creating dataset for attack
df_attack = read_from_elastic(index, es)
df_attack.shape


# In[ ]:


# Preserving timestamp column on a new dataframe
df_attack_timestamp = pd.DataFrame()
df_attack_timestamp['timestamp'] = df_attack['timestamp']

# Excluding timestamp column
df_attack.drop(['timestamp'], axis=1, inplace=True)


# In[ ]:


# Defining window_size and n_feature
window_size = 6
n_features = df_attack.shape[1]


# ## Pre-processing the data
```

```
# In[ ]:


# Normalizing data

# Loading existent df from disk
df_benign_data = pd.read_pickle("pkl/df_proc_benign_data.pkl")

mm = MinMaxScaler()
mm_attack = mm.fit(df_benign_data)
attack = mm_attack.transform(df_attack)

print(attack_name, "numpy.ndarray shape:", attack.shape)


# In[ ]:


# Creating 3D array for train data
# For an LSTM Autoencoder the shape of input has to be of the format: n_samples x
↪  window_size x n_features
attack_wz = pd.DataFrame(attack)
attack_wz = sliding_window(attack_wz, window_size)


# ## Creating Tensorflow datasets

# ### Attack dataset

# In[ ]:


# Attack dataset
ds_attack = tf.data.Dataset.from_tensor_slices(attack_wz)
ds_attack = ds_attack.map(lambda x: (x, x))
ds_attack_batch = ds_attack.batch(1024).cache().prefetch(tf.data.AUTOTUNE)


# ## Loading the trainned model

# In[ ]:


model_name = 'tfds_lstm_160_64_24_conv1d_relu_5_bn_tahn_wz6_ft5_mm'


# In[ ]:


# Saving/Loading the model
filepath = 'model-' + str(model_name) + '.h5'
model = load_model(filepath, compile=True)
```

```
# ## Predicting test data using the model

# In[ ]:


# Predicting values using the trained model
pred = model.predict(ds_attack_batch)


# In[ ]:


# Reshaping array with predictions to 2D dataframe (column 2 x column 3)
#X_pred.shape #--> (samples - window_size, window_size, n_features)
pred = pred.reshape(pred.shape[0], pred.shape[1] * pred.shape[2])
df_pred = pd.DataFrame(pred)


# In[ ]:


# Reshaping array with real data to 2D dataframe (column 2 x column 3)
#X_test.shape # --> (samples - window_size, window_size, n_features)
attack_2d = attack_wz.reshape(attack_wz.shape[0], attack_wz.shape[1] *
↪  attack_wz.shape[2])
df_attack_2d = pd.DataFrame(attack_2d)


# ### Calculating the loss

# In[ ]:


# Calculating test loss with MAE (Mean Absolute Error)
df_test_loss = pd.DataFrame(index=df_pred.index)
df_test_loss['Loss_mae'] = tf.metrics.MAE(df_attack_2d, df_pred)


# In[ ]:


# Plotting the loss distribution
plot = sns.displot(data=df_test_loss['Loss_mae'], kind='kde', color='blue',
↪  height=5, aspect=2)
plot.set_axis_labels("Loss", "Density")
plot.set(title='Loss Distribution')


# ## Attack data x Loss Threshold
```

```
# In[ ]:


# Confidence level e threshold escolhidos a partir dos valores obtidos com dados de
↪  teste
confidence_level = 0.99
loss_threshold = 0.0169


# ### Verifying attack loss against defined threshold

# #### Labeling Malicious Dataframe

# In[ ]:


# Calculating rows (windows) hash
attack_hash = df_attack_2d.apply(lambda x: hash(tuple(x)).to_bytes(8, "big",
↪  signed=True).hex(), axis=1)
df_attack_hash = pd.DataFrame(attack_hash)


# In[ ]:


# Loading benign_hashdb
df_bening_hashdb = pd.read_pickle('pkl/df_proc_benign_hashdb_wz6_ft5_mm.pkl')


# In[ ]:


# Adding real label to df_test_loss
df_test_loss['real'] = ~df_attack_hash[0].isin(df_bening_hashdb[0])


# In[ ]:


# Creating dataframe with test data results
df_test_results = window_loss(df_test_loss, loss_threshold)


# ### Ploting loss of the test data prediction

# In[ ]:


# Plotting the test data x loss threshold
df_test_results_plot = df_test_results[['window_number', 'loss']]
```

```
df_test_results_plot.plot(kind='line', marker= 'H', x='window_number', y='loss',
↪  ylabel='Loss', xlabel='Window number', figsize=(20,
↪  7)).axhline(y=loss_threshold, linewidth= 1, color='r')


# In[ ]:


df_test_results['anomaly'].value_counts(), df_test_results['anomaly'].value_counts(
↪  normalize=True).mul(100).round(2).astype(str) +
↪  '%'


# ## Metrics

# In[ ]:


# Storing metrics results
cm, accuracy, precision, tpr, npv, tnr, fpr, f1, roc_auc = metrics(df_test_results)


# In[ ]:


print("--- MÉTRICAS ---")
print("Acurácia:", np.round(accuracy * 100, 2), "%")
print("Precisão:", np.round(precision * 100, 2), "%")
print("TPR ou Recall:", np.round(tpr * 100, 2), "%")
print("NPV:", np.round(npv * 100, 2), "%")
print("TNR ou Especificidade:", np.round(tnr * 100, 2), "%")
print("FPR ou FAR:", np.round(fpr * 100, 2), "%")
print("F1 Score:", np.round(f1 * 100, 2), "%")
print("ROC AUC:", np.round(roc_auc * 100, 2), "%")


# In[ ]:


cmd = ConfusionMatrixDisplay(cm, display_labels=['Normal','Anomaly'])
cmd.plot(cmap="Blues", values_format='', )


# In[ ]:


fpr_roc, tpr_roc, _ = roc_curve(df_test_results['real'], df_test_results['anomaly'])


# In[ ]:
```

```
plt.title('Receiver Operating Characteristic')
plt.plot(fpr_roc, tpr_roc, label = 'AUC = %0.2f' % roc_auc)
plt.legend(loc = 'lower right')
#plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()


# ### Saving metrics

# In[ ]:


if os.path.isfile('pkl/df_proc_metrics.pkl'):
    # Loading existent df from disk
    df_metrics = pd.read_pickle('pkl/df_proc_metrics.pkl')
else:
    # Defining dataframe columns
    df_metrics =
    ↪  pd.DataFrame(columns=["Model","Attack","Confidence_Level","Threshold","Accu
    ↪  racy","Precision","TPR","NPV","TNR","FPR","F1-Score","ROC-AUC","CM"])

# Adding last execution results in to dataframe
df_metrics.loc[df_metrics.shape[0]] = [filepath, attack_name, confidence_level,
↪  loss_threshold, accuracy, precision, tpr, npv, tnr, fpr, f1, roc_auc, cm]

# Saving df to disk
df_metrics.to_pickle('pkl/df_proc_metrics.pkl')


# ### Writing results in Elasticsearch

# In[ ]:


# Test conectivity with Elasticsearch
es = Elasticsearch(host="192.168.201.2", http_auth=("elastic","7s9w1sbwjcivorj6"))


# In[ ]:


# Creating index with each window loss
index = "proc-public-" + str(attack_name)
index = "scan-" + str(index)


# In[ ]:
```

```
# Copying df_test_results to a new dataframe before sending to Elastic
df_result_es = df_test_results.copy()

# The dataframe to insert in elasticsearch must have a column with name 'indexId'
↪  (https://github.com/dashaub/espandas#usage)
df_result_es['indexId'] = df_result_es.index.astype(str)

# Removing window_number column to reduce size of dataframe
df_result_es.drop(['window_number'], axis=1, inplace=True)

# Coverting 'anomaly' colum to string lower case due to Elastic requirements for
↪  boolean mapping type
df_result_es['anomaly'] = df_result_es['anomaly'].astype('string').str.lower()

# Adding chosen loss threshold to the dataframe
df_result_es['threshold'] = loss_threshold

# Adding chosen confidence_interval to the dataframe
df_result_es['confidence_level'] = confidence_level

# Adding the timestamp of the first system call in window to the dataframe
df_result_es.loc[:, 'timestamp'] = df_attack_timestamp.loc[:, 'timestamp']


# In[ ]:


# Configuring Elastic credentials
esp = Espandas(host="192.168.201.2", http_auth=("elastic","7s9w1sbwjcivorj6"))

# Writing index in Elastic
esp.es_write(df_result_es, index=index, doc_type=None)
```