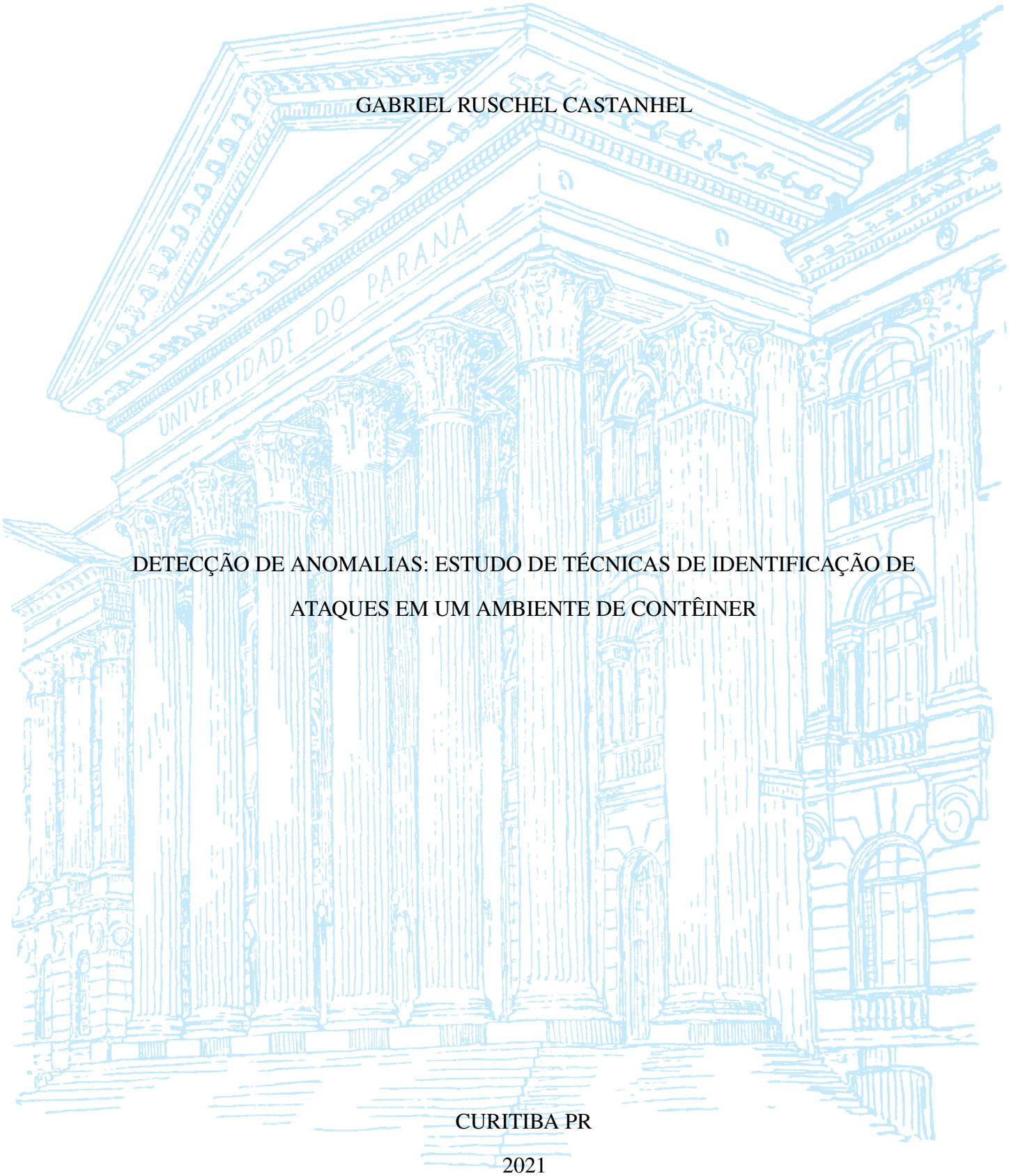
UNIVERSIDADE FEDERAL DO PARANÁ

GABRIEL RUSCHEL CASTANHEL



DETECÇÃO DE ANOMALIAS: ESTUDO DE TÉCNICAS DE IDENTIFICAÇÃO DE ATAQUES EM UM AMBIENTE DE CONTÊINER

CURITIBA PR

2021

GABRIEL RUSCHEL CASTANHEL

DETECÇÃO DE ANOMALIAS: ESTUDO DE TÉCNICAS DE IDENTIFICAÇÃO DE ATAQUES EM UM AMBIENTE DE CONTÊINER

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Carlos Alberto Maziero.

Coorientador: Tiago Heinrich.

CURITIBA PR

2021

Ficha catalográfica

Substituir o arquivo `0-iniciais/catalografica.pdf` pela ficha catalográfica fornecida pela Biblioteca da UFPR (PDF em formato A4).

Instruções para obter a ficha catalográfica e fazer o depósito legal da tese/dissertação (contribuição de André Hochuli, abril 2019):

1. Verificar se está usando a versão mais recente do modelo do PPGInf e atualizar, se for necessário (`https://gitlab.c3sl.ufpr.br/maziero/tese`).
2. conferir o Checklist de formato do Sistema de Bibliotecas da UFPR, em `https://portal.ufpr.br/teses_servicos.html`.
3. Enviar email para "`referencia.bct@gmail.com`" com o arquivo PDF da dissertação/tese, solicitando a respectiva ficha catalográfica.
4. Ao receber a ficha, inseri-la em seu documento (substituir o arquivo `0-iniciais/catalografica.pdf` do diretório do modelo).
5. Emitir a Certidão Negativa (CND) de débito junto a biblioteca (`https://www.portal.ufpr.br/cnd.html`).
6. Avisar a secretaria do PPGInf que você está pronto para o depósito. Eles irão mudar sua titulação no SIGA, o que irá liberar uma opção no SIGA pra você fazer o depósito legal.
7. Acesse o SIGA (`http://www.prppg.ufpr.br/siga`) e preencha com cuidado os dados solicitados para o depósito da tese.
8. Aguarde a confirmação da Biblioteca.
9. Após a aprovação do pedido, informe a secretaria do PPGInf que a dissertação/tese foi depositada pela biblioteca. Será então liberado no SIGA um link para a confirmação dos dados para a emissão do diploma.

Ficha de aprovação

Substituir o arquivo 0-iniciais/aprovacao.pdf pela ficha de aprovação fornecida pela secretaria do programa, em formato PDF A4.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais Francisco e Adriana, pela minha primeira formação. Por me ensinarem, me apoiarem e por me incentivarem a trilhar o meu próprio caminho, estando ao meu lado sempre, mesmo à 600 km de distância. À minha irmã Isadora, por todo o companheirismo, todas as brigas e todas as cobranças, por acreditar sempre no melhor de mim, e por me ensinar muito mais do que eu posso listar.

Sou muito grato ao meu coorientador Tiago Heinrich, pela disponibilidade e pela enorme ajuda e orientação no desenvolvimento do trabalho, pelo apoio e incentivo em produzir e apresentar os três artigos derivados deste trabalho, que contribuíram muito na minha formação. Ao Prof. Dr. Carlos Alberto Maziero por ter me orientado ao longo de todo o percurso deste trabalho de graduação, e também por todo o conhecimento e ensinamentos compartilhados ao longo da minha graduação.

Sou grato ao Departamento de Informática (Dinf) da universidade, por toda a infraestrutura, recursos e oportunidades que tive o privilégio de usufruir ao longo desta jornada de graduação. À Universidade Federal do Paraná, por uma educação superior pública, gratuita e de qualidade.

Finalmente, também sou muito grato a todos os amigos que me acompanharam ao longo desta jornada. Aos amigos de antes, que mesmo com os obstáculos da distância mantiveram a mesma parceria e amizade. E também aos amigos que fiz durante esta caminhada na universidade, pelos trabalhos e estudos em conjunto que me ajudaram imensamente, e por toda a amizade, apoio, frustrações, festas e risadas compartilhadas, por fim, meu muito obrigado.

RESUMO

O crescimento do uso de tecnologias de virtualização na última década contribuiu com o desenvolvimento de diferentes técnicas, entre elas a virtualização baseada em contêineres. Com o crescente uso e implementação deste tipo de ambiente isolado, surgem também questionamentos relacionados à segurança de tais sistemas. Considerando a proximidade entre *host* e contêiner, abordagens utilizando detecção de intrusão por anomalias são uma alternativa para monitorar e detectar comportamentos inesperados. Este trabalho propõe o uso de *system calls* como dado base para identificar ameaças em um ambiente de contêiner, utilizando algoritmos de *Machine Learning* (ML) para distinguir entre comportamento normal e anômalo. Uma estratégia que realiza uma filtragem de *system calls* avaliadas como inofensivas é avaliada.

Palavras-chave: Detecção de Intrusão, Segurança de Computadores, e Contêineres.

ABSTRACT

The usage growth of virtualization technologies in the last decade contributed to the development of different techniques, such as container-based virtualization. With the increasing use and implementation of such isolated environments, security concerns about such systems also arise. Considering the host's proximity to a container, anomaly intrusion detection approaches come up as an alternative to monitor and detect unexpected behaviors. This work proposes to use system calls to identify threats within a container environment, using Machine Learning algorithms to distinguish between normal and anomalous behavior. A strategy that performs filtering of system calls classified as harmless is evaluated.

Keywords: Intrusion Detection, Computer Security, and Containers.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

BoSC	Bag of System Calls
HIDS	Host-based Intrusion Detection System
IDS	Intrusion Detection System
ML	Machine Learning
MLP	Multilayer Perceptron
KNN	K-Nearest Neighbors
NIDS	Network Intrusion Detection System
RF	Random Forest
RCE	Remote Code Execution
SO	Sistema Operacional
STIDE	Sequence Time-Delay Embedding
SVM	Support Vector Machine
XSS	Cross-site Scripting

SUMÁRIO

1 INTRODUÇÃO

A partir da difusão da Internet, devido à facilidade e eficiência de oferecer serviços e comunicações entre os mais diversos usuários e organizações ao redor do planeta, o conceito de computação em nuvem (ou *cloud computing*) se destacou como uma tendência. Ele proporcionou uma revolução na forma como são implementados e disponibilizados serviços na rede, e exerce grande influência na definição do conceito de Internet que é conhecido atualmente (Deshpande et al., 2018).

A computação em nuvem refere-se à capacidade de usufruir de um conjunto de recursos de computação pertencentes e mantidos por terceiros através da Internet. Não é uma tecnologia nova por se dizer, mas sim um novo método para disponibilizar recursos computacionais baseado em tecnologias já existentes como a virtualização de servidores. A "nuvem" é composta por *hardware*, armazenamento, redes, interfaces, serviços, que fornecem os meios pelos os quais usuários podem usufruir de infraestruturas, aplicações, poder computacional e serviços sob demanda, estes que funcionam independente da localização (Arora et al., 2013).

A tecnologia para virtualização de sistemas foi uma das técnicas que, apesar de já existente, recebeu grande atenção e desenvolvimento nos últimos anos. Também foi um dos expoentes para a popularização deste novo conceito de computação em nuvem, por possibilitar uma alocação flexível de recursos físicos para aplicações virtualizadas onde é possível controlar e adaptar dinamicamente os recursos utilizados, ajustando-se à demanda de cada aplicação. Esta técnica também possibilita manter múltiplas instâncias de diferentes sistemas e aplicações, com diferentes requisitos, compartilhando um mesmo servidor, o que proporcionou uma otimização de uso de recursos físicos e redução de custos operacionais (Sharma et al., 2016).

Das técnicas de virtualização existentes, dois tipos podem ser destacados: um deles é a virtualização em nível de *hardware*, que envolve a utilização de um *hypervisor*. O *hypervisor* é o componente que faz a virtualização dos recursos de *hardware* para múltiplas máquinas virtuais, onde cada máquina virtual executa seu próprio sistema operacional e aplicações. O outro tipo é a virtualização em nível de sistema operacional, que por sua vez realiza a virtualização de recursos do sistema operacional. São encapsulados processos padrão do sistema e dependências em um contêiner, sendo este administrado pela ferramenta responsável pela virtualização e compartilha o *kernel* do sistema operacional com os demais contêineres no mesmo computador. Exemplos de ferramentas para virtualização de *hardware* são: KVM, VMWare e XEN. Já exemplos para virtualização de sistema operacional, podem ser listados Docker e LXC, baseados em contêineres Linux (Sharma et al., 2016).

O desenvolvimento de técnicas de virtualização proporcionaram melhorias, inclusive na segurança de aplicações disponíveis na web, por proporcionar camadas extra de abstração e distanciando ainda mais a interação direta entre o usuário e o servidor em si. Entretanto, é possível encontrar sistemas que fazem uso de virtualizações e que são alvo de diferentes tipos de ataque, onde é comum ocorrer o vazamento de bases de dados sensíveis, a indisponibilidade temporária do serviço ou corrupção de dados.

Existem algumas ameaças que são únicas a ambientes virtualizados. Ataques entre virtualizações por exemplo, que ocorrem quando um sistema virtualizado pode interferir em outro que executa sob o mesmo *host*, o que ocorre devido a um fraco isolamento dos sistemas. Outro exemplo seria quando um atacante pode executar código que o permita "escapar" do ambiente virtualizado e interagir diretamente com o sistema subjacente.

Na tentativa de encontrar soluções para tais problemas, estudos e pesquisas voltados à segurança de sistemas virtualizados ganharam enfoque nos últimos anos. Se tratando de virtualizações, o desafio é encontrar uma maneira onde seja possível identificar ataques acontecendo dentro do sistema sem interferir em sua execução normal. Para que assim seja possível identificar o tipo de ameaça, evitar que o sistema seja comprometido e também evitar que o ataque seja possivelmente direcionado a outro sistema virtualizado, a ferramenta responsável pela administração dos sistemas virtuais presentes ou o sistema subjacente.

Pelo fato de as virtualizações trabalharem com múltiplos sistemas diferentes e independentes, se torna difícil desenvolver um *framework* de segurança que possa cobrir os aspectos de segurança necessários para todas as aplicações e sistemas possíveis nos ambientes virtualizados. Uma das soluções mais recorrentes se baseia no desenvolvimento de um IDS (*Intrusion Detection System*), que elabora um meio de identificar e prevenir atividades maliciosas realizando o monitoramento de recursos do sistema. Um IDS pode detectar se uma ação maliciosa está em andamento com base em um conhecimento prévio de comportamento de ataques direcionados ao sistema. Para esta abordagem, um IDS pode ser classificado como baseado em assinatura, onde realiza detecções com base em assinaturas de ataques já conhecidos; ou baseado em anomalias, onde é traçado um perfil de comportamento normal do sistema e busca identificar ataques que diferem deste perfil conhecido (Mishra et al., 2017).

Para este trabalho de conclusão de curso, foi realizado um estudo com enfoque em detecção de intrusão por detecção de anomalias em um ambiente de virtualização em nível de sistema operacional proporcionada pela ferramenta *Docker* em um sistema operacional Linux. Esta ferramenta foi escolhida por ser a mais recorrente e popular atualmente neste contexto. O estudo se baseou em executar uma aplicação web em um contêiner *Docker* com a finalidade de monitorar o comportamento da aplicação em situações de uso normal e também em situações de ataques ou operações mal intencionadas, para então, testar e comparar a eficiência de diferentes técnicas de classificação para a detecção de intrusão neste contexto.

1.1 OBJETIVOS

Objetivo geral: Realizar um estudo voltado à detecção de intrusão por anomalias em um ambiente de contêiner *Docker*.

Objetivos específicos: Comparar e analisar a eficiência de diferentes técnicas de classificação de aprendizado de máquina para a detecção de anomalias neste contexto; construir uma base representativa do comportamento (com foco no comportamento anormal) da aplicação web em um ambiente de contêiner; estudar o impacto da diferença de tratamento de dados coletados nos métodos utilizados

1.2 METODOLOGIA

Este trabalho de conclusão de curso descreve uma pesquisa aplicada seguindo o método de estudo de caso. Em primeiro lugar foi realizado um breve estudo sobre a possibilidade da captura do fluxo de *system calls*, que representaria o comportamento de uma aplicação web em um contêiner *docker*, e também alguns métodos básicos de detecção de anomalia. Foi então disponibilizada uma aplicação web para ser analisada, com o *Wordpress* como aplicação escolhida, também foram feitos alguns ajustes na aplicação a fim de diminuir o número de processos e *threads* utilizadas, procurando preservar ao máximo a sequência de chamadas geradas. Feito isso, foram construídos dois *datasets*, onde para ambos foi realizada a coleta de dados gerados pela aplicação primeiro sob comportamentos normais e inofensivos, e depois sob influência de ataques

e ações mal intencionadas. Com as bases de dados definidas, foram testados classificadores de ML (*machine learning*) para a detecção de anomalia em ambos os *datasets*, comparando algumas métricas de cada método. Também foram realizados os testes após uma filtragem de chamadas classificadas como inofensivas a partir de um nível de ameaça atribuído a cada *system call* encontrada, utilizando os mesmos métodos e métricas.

1.3 ORGANIZAÇÃO TEXTUAL

Este trabalho está dividido em 6 capítulos. O capítulo 2 descreve a fundamentação teórica, apresentando conceitos de detecção de anomalia, IDSs, *system calls*, virtualização baseada em contêiner e alguns métodos de detecção. O capítulo 3 apresenta a revisão bibliográfica da literatura envolvendo detecção de anomalia utilizando *system calls*. O capítulo 4 apresenta a proposta, descrevendo a estratégia, o processo de coleta de dados e filtragem. O capítulo 5 apresenta a avaliação dos resultados encontrados e por fim, o capítulo 6 apresenta a conclusão e considerações finais deste trabalho e também perspectivas para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo introduz os conceitos básicos necessários para a compreensão dos experimentos e resultados encontrados no decorrer do trabalho. A seção 2.1 introduz os conceitos relacionados aos sistemas de detecção de intrusão, a seção 2.2 descreve os conceitos envolvidos em detecção de anomalia, a seção 2.3 fala sobre *system calls*, a subseção 2.3.1 apresenta conceitos sobre um dos primeiros métodos de detecção de anomalias com *system calls*, a seção 2.4 apresenta detalhes sobre a virtualização baseada em contêiner e por fim, a seção 2.5 descreve conceitos de ML.

2.1 SISTEMA DE DETECÇÃO DE INTRUSÃO

Decorrente da necessidade de estabelecer a segurança de sistemas computacionais modernos, atualmente é comum encontrar estudos e pesquisas dedicados à identificação e prevenção de ameaças que acontecem em tempo real ou que podem vir a acontecer eventualmente (Deshpande et al., 2018).

Inicialmente o processo de análise para identificar atividades anômalas ou que representam algum tipo de perigo se baseava na auditoria de *logs* e relatórios gerados por diferentes *hosts* (Anderson, 1980). Por sua vez, esta prática é considerada uma base para a detecção de anomalias: o monitoramento de recursos do sistema em busca de comportamentos que diferem do que é assumido como "normal" para uma aplicação ou sistema operacional, que pode indicar uma possível falha ou ameaça externa.

O estudo do primeiro modelo de sistema de detecção de intrusão foi apresentado por (Denning, 1987), e tece como base a hipótese de que violações de segurança podem ser detectadas a partir da monitoração de relatórios de auditoria de um sistema, em busca de padrões anormais de uso. O modelo inclui perfis para representar o comportamento em termos de métricas e modelos estatísticos, e regras para extrair conhecimento sobre esse comportamento a partir de registros de auditoria e para detectar comportamento anormal. O modelo é independente de qualquer sistema específico, ambiente de aplicativo, vulnerabilidade de sistema, ou tipo de intrusão, fornecendo assim um *framework* para um sistema de detecção de intrusão de propósito geral.

De acordo com (Bace e Mell, 2001), a detecção de intrusão é classificada como "o processo de monitorar os eventos ocorrendo em um sistema computacional ou rede e analisá-los em busca de sinais de intrusão, definidos como tentativas de comprometer a confidencialidade, integridade e disponibilidade dos dados ou neutralizar mecanismos de defesa de um computador ou de uma rede". Os IDSs são subdivididos em duas classes em relação a sua área de atuação: arquitetados para atuar em *hosts* específicos, conhecidos como *Host-based Intrusion Detection Systems* (HIDS) ou para atuar em uma rede composta por diversos *hosts*, em um *Network Intrusion Detection System* (NIDS). Existem diferentes técnicas que IDSs modernos podem empregar para reunir e analisar dados, entretanto, uma grande parte conta com uma arquitetura comum: um módulo de detecção, que obtém e reúne dados de comportamento do objeto alvo, um mecanismo de análise que processa a informação coletada com o objetivo de detectar atividades intrusivas, e um componente de resposta que relata intrusões (Brown et al., 2002).

É possível encontrar na literatura certas métricas utilizadas para avaliar a eficiência de um IDS, como por exemplo (Debar et al., 1999):

- **Accuracy (acurácia):** Representa a taxa de detecções corretamente previstas como intrusões. Baixos valores ocorrem quando o sistema identifica como anomalia um comportamento legítimo no ambiente;
- **Performance (desempenho):** O desempenho de um IDS é a taxa na qual os dados são processados. Se o desempenho é baixo, então detecções em tempo real não são possíveis;
- **Completeness (completude):** Define a abrangência de detecção. A incompletude ocorre quando um sistema falha em detectar um ataque que ocorreu. É uma métrica difícil de avaliar, pois é complexo obter uma base de conhecimento que possa reconhecer todos os ataques possíveis;
- **Fault tolerance (tolerância a falhas):** Define que o próprio sistema de detecção de intrusão deve ser resistente a ataques e invasões que comprometam o seu funcionamento;
- **Timeliness (pontualidade):** Um IDS deve realizar e propagar suas análises o mais cedo possível, para que assim o profissional de segurança possa tomar atitudes para frear ataques o quanto antes, o que vai além do desempenho, pois considera também o tempo de propagação e reação para um alerta gerado.

Diversas técnicas podem ser utilizadas para a detecção de intrusão, com duas grandes categorias sendo mais utilizadas: (i) baseadas em assinatura, onde o sistema realiza a identificação de ameaças de ataques com uma base de padrões de ameaça, que reúne diversas assinaturas de uma variedade numerosa de ataques que já são conhecidos; (ii) e baseadas em anomalias, onde é definida uma representação do que consiste o comportamento normal do sistema alvo, e é realizado o monitoramento de certos recursos do sistema em busca de padrões que desviam do comportamento normal definido, e consequentemente podem indicar possíveis ameaças (Debar et al., 1999; Yassin et al., 2013)

Também existem sistemas que fazem uso dessas duas categorias na mesma implementação, chamados de modelos híbridos, onde é realizada a detecção por assinaturas e também por anomalias.

Um exemplo de sistemas que utilizam técnicas baseadas em assinaturas são os antivírus, que se tornaram um sucesso comercial nas últimas décadas. Atualmente, essa categoria de *software* oferece diversas soluções diferentes no âmbito de segurança de sistemas. Entretanto, uma parcela considerável das soluções realizam algumas tarefas em comum, como a análise de arquivos e serviços vindos de redes remotas, e varreduras periódicas de arquivos e serviços do sistema, a fim de identificar assinaturas conhecidas como ameaças, e portanto dependem de uma grande e complexa base de assinaturas de ataques, que precisa ser constantemente atualizada.

Na detecção de anomalias, uma definição do comportamento normal ou esperado é aprendida pelo sistema a partir dos dados utilizados para o treinamento, e um desvio deste perfil que atinge um certo nível limite é sinalizado como uma possível ameaça (Bridges et al., 2019). A vantagem de sistemas baseados em assinatura é o fato de que eles podem identificar ataques com uma acurácia aceitável e costumam produzir uma menor quantidade de alarmes falsos comparado com os sistemas baseados em anomalias. Também são mais fáceis de implementar e configurar, e por conta disso, grande parte dos sistemas comerciais e soluções instaladas usam detecção baseada em assinatura (Kruegel e Toth, 2003).

Entretanto, a detecção baseada em assinatura possui algumas desvantagens, como por exemplo: requer conhecimentos específicos sobre comportamentos intrusivos, onde dados coletados antes do ataque podem estar desatualizados, ao mesmo tempo que é mais difícil detectar

ataques novos ou desconhecidos. Também possui o problema de gerar alertas independente do resultado, o que pode tornar difícil a administração dos mesmos. Pode não ser eficiente para detectar ataques internos que envolvem abusos de privilégios, e a base de conhecimento de ataques é dependente do sistema operacional, aplicação e outros fatores de ambiente (Kumar e Sangwan, 2012).

Referente à detecção de anomalias, é possível definir duas principais vantagens: A primeira é a capacidade destes sistemas de detectarem ataques desconhecidos, assim como os ataques "*zero day*", que são ataques recém descobertos, que ainda não possuem defesas conhecidas e divulgadas. Isso acontece por conta do seu modelo de funcionamento, onde é possível detectar comportamentos muito distantes do comportamento normal esperado do sistema, e possui uma importância relevante no contexto de segurança de sistemas modernos. Uma segunda vantagem seria que os perfis de comportamento normal são customizados para cada sistema, aplicação ou rede, o que dificulta para um atacante saber com certeza que tipo de atividades pode realizar sem ser detectado. Entretanto, o modelo de detecção de anomalias também possui desvantagens, como por exemplo é comum obter uma alta taxa de falsos positivos, o que dificulta o tratamento de alertas gerados e a dificuldade de determinar quais eventos geraram tais alertas. É possível também listar a complexidade inerente a este modelo, e a dificuldade de manter perfis de comportamento que sejam representativos ao ambiente como desafios na implementação de sistemas que utilizam a detecção de anomalia (Patcha e Park, 2007).

Já ao tratar de segurança de redes, um NIDS tem como objetivo detectar possíveis intrusões, como por exemplo atividades maliciosas, ataques a *hosts* e proliferação de vírus, e realizar alertas para tais eventos. O sistema realiza um monitoramento e analisa os pacotes de rede que percorrem uma determinada rede, em busca de atividades suspeitas. Um NIDS de grande porte pode ser configurado nos enlaces de uma rede de *backbone*, para monitorar todo o tráfego, ou também pode ser configurado em sistemas menores, para monitorar o tráfego direcionado a um servidor específico, *switch*, *gateway* ou roteador (Kumar, 2007).

Um HIDS tem a habilidade de detectar ataques internos, o que indica que a procura por atividades maliciosas é realizada em sistemas baseados em *hosts*, e pode ter como alvo o sistema do *host* como um todo, ou aplicações específicas. Consequentemente, ao contrário de um NIDS que utiliza principalmente apenas dados de rede como fonte de informação, um HIDS tem acesso a um leque maior de fontes de informação que podem ser utilizadas, entre elas: Dados referentes a utilização de recursos, como memória, disco, uso de rede e processos executando no sistema; dados de auditoria como logs gerados tanto pelo sistema operacional quanto por aplicações, que contém informações de processos envolvidos e *timestamps*, e também dados gerais de auditoria do sistema como mapeamento de processos, *system calls*, usuários, grupos, etc (Deshpande et al., 2018).

Sistemas de detecção de intrusão podem variar de acordo com o tratamento dos dados utilizados, onde é possível encontrar diferentes abordagens que podem ser adotadas. Na detecção de anomalia por exemplo, é possível listar métodos baseados em variáveis indicadoras, onde os dados categóricos são representados como valores numéricos; métodos baseados em frequência, que fazem uso da frequência dos dados de respectivas categorias para a análise; e métodos baseados em distância, que leva em consideração a distância entre observações obtidas dos dados a serem analisados (Taha e Hadi, 2019). É possível encontrar sistemas que tenham diferentes aplicações operacionais, como sistemas de detecção distribuídos, sistemas voltados a aplicações específicas, virtualizações, etc.

2.2 DETECÇÃO DE ANOMALIA

A detecção de anomalia refere-se à tarefa de encontrar padrões presentes em conjuntos de dados que não condizem com o comportamento normal esperado, estes são identificados como anomalias ou disparidades, e é uma técnica vastamente aplicada em diversos domínios como a segurança de sistemas computacionais, finanças, estatísticas e análise de fraude. A importância da detecção de anomalias é devido ao fato de que anomalias em conjuntos de dados representam informações significativas, e críticas (Chandola et al., 2009), como por exemplo um tráfego incomum de dados em uma rede de computadores pode significar que um computador foi comprometido e está fornecendo informações a terceiros não autorizados.

Anomalias são padrões que diferem de um conjunto de dados que representa um comportamento considerado normal, e que podem ocorrer em um conjunto de dados de diversas formas dependendo do domínio da análise. Com isso, é possível afirmar que a principal etapa para a implementação de um sistema guiado por detecção de anomalia é uma definição consistente e abrangente do que é, de fato, comportamento normal no ambiente a ser estudado, o que acaba sendo também um desafio na utilização dessa metodologia.

Referente à segurança de sistemas computacionais, a detecção de anomalias apresenta diversos desafios para a obtenção de resultados relevantes que refletem a realidade dos acontecimentos analisados, entre eles: (*i*) ter um conjunto de dados relevante que represente, em uma aplicação por exemplo, o seu comportamento normal, o que pode se tornar uma tarefa difícil dependendo do tamanho, complexidade e da quantidade de usuários (remotos ou locais) que interagem com a mesma; (*ii*) quando estas anomalias referem-se a ações maliciosas, é comum que atacantes adaptem tais ações para que sejam similares a operações normais e autorizadas, o que pode dificultar o trabalho da detecção de anomalias; (*iii*) em muitos domínios, o conceito de comportamento normal evolui constantemente, o que faz com que uma representação atual de "normal" não seja o suficiente no futuro; (*iv*) diferentes domínios possuem diferentes padrões e limiares que podem representar uma possível anomalia; (*v*) a precária disponibilidade de bases de dados representativas e propriamente classificadas para treinamento e testes é um problema recorrente; (*vi*) é comum que os dados obtidos tenham "ruído", que pode facilmente passar por anomalia e que não representa perigo real para o ambiente, o que prejudica a eficiência do método. (Chandola et al., 2009)

Técnicas de detecção de anomalia resolvem uma formulação específica de um problema, esta formulação que é influenciada por diversos fatores como a natureza dos dados a serem analisados, a disponibilidade de dados classificados, tipos de anomalias a serem detectadas, etc. Tais fatores são determinados pelo domínio da aplicação em que estas anomalias devem ser detectadas.

Um aspecto importante da detecção de anomalia é a natureza da anomalia, que pode ser classificada em três categorias (Ahmed et al., 2016):

Anomalias pontuais: Quando uma instância individual pode ser considerada anômala em relação ao restante dos dados, então essa instância é denominada uma anomalia pontual.

Anomalias contextuais: Quando uma instância é anômala apenas em um contexto específico e nenhum outro caso, então é considerada uma anomalia contextual.

Anomalias coletivas: Quando uma coleção de dados relacionados é anômala em comparação ao *dataset* como um todo, então é considerada como uma anomalia coletiva. Os dados individuais nesta coleção podem não ser anomalias por si mesmos, porém sua ocorrência em grupo é considerada anômala.

Em relação aos rótulos (ou *labels*), estes são associados à instâncias de dados e determinam se determinada instância é considerada como normal ou anômala, o que atua

auxiliando no processo de classificação para a detecção de anomalia. Por ser um trabalho manual, o rotulamento de dados requer um esforço considerável (Chandola et al., 2009).

Existem três abordagens fundamentais para o problema de detecção de anomalia (Hodge e Austin, 2004):

Detecção de anomalia supervisionada: Nesta classe, são necessários dados pré-rotulados, ou seja, que os dados utilizados para treinar o classificador possuam instâncias classificadas como normais, e também instâncias identificadas como anormais, onde os dados não antes vistos são comparados com o modelo para determinar a qual classe pertencem.

Detecção de anomalia semi supervisionada: Técnicas que operam de modo semi supervisionado assumem que os dados para treinamento do classificador são rotulados apenas para a classe normal; como não necessitam de classificações para a classe anômala, são métodos mais utilizados que a detecção de anomalia supervisionada. A abordagem mais típica é construir um modelo de dados para a classe considerada como comportamento normal, e utilizar esse modelo para identificar anomalias.

Detecção de anomalia não supervisionada: Nas técnicas de detecção não supervisionada, anomalias são determinadas sem nenhum conhecimento prévio sobre os dados, e assumem que instâncias normais são encontradas em quantidades maiores que anomalias no conjunto de dados. Caso essa suposição não seja verdadeira, tais técnicas sofrem com uma alta taxa de alarmes falsos.

2.3 SYSTEM CALLS

Todo sistema operacional executando em qualquer tipo de *host* depende de um *kernel* (ou núcleo). O *kernel* é o programa responsável por implementar as operações mais básicas de um sistema operacional: ele controla as interfaces de *hardware*, lida com a alocação de memória e recursos de *hardware*, permite que múltiplos programas executem em paralelo, gerencia o sistema de arquivos, entre outras funcionalidades (Mitchell et al., 2001).

A camada do *kernel* executa em um modo especial de operação do processador denominado modo privilegiado, enquanto os demais programas e aplicações executam no modo denominado modo usuário. Enquanto o *kernel* e *drivers* devem ter pleno acesso ao *hardware* para poder configurá-lo e gerenciá-lo, os aplicativos e utilitários devem ter acesso mais restrito a ele, para não interferir nas configurações e potencialmente desestabilizar o sistema inteiro. Além disso, aplicações comuns com acesso irrestrito ao *hardware* seriam um risco à segurança pois poderiam contornar facilmente os mecanismos de controle de acesso aos recursos. Entretanto, é comum que aplicações do nível de usuário precisem requisitar recursos do *hardware* durante sua execução, nestes casos são utilizadas rotinas do *kernel* através do mecanismo de interrupções, operação que é conhecida como chamada de sistema (*system call* ou *syscall*). Os sistemas operacionais definem chamadas de sistemas para todas as operações envolvendo o acesso a recursos de baixo nível (memória, arquivos, periféricos, etc.) ou abstrações lógicas (criação e encerramento de tarefas, operadores de sincronização, etc.) (Maziero, 2020).

Uma *system call* é o recurso disponível para a interação entre uma aplicação (ou processo) e o *kernel* do sistema operacional. Quando um processo precisa realizar operações com o *kernel*, as requisições são feitas através das *system calls* disponíveis, pela principal razão de que processos comuns executando em modo usuário não tem permissão para realizar tais atividades do modo *kernel*. Isso ocorre constantemente para uma variedade de tarefas como: interagir com recursos do *hardware*, gerenciamento de memória, processamento de entrada/saída, uso da interface de rede, manipulação do sistema de arquivos, entre outras. As *system calls* representam uma interface essencial entre sistema operacional e processos, onde cada chamada representa uma

operação básica ou capacidade. Sistemas Linux por exemplo oferecem por volta de 530 *system calls*, a listagem de todas as chamadas da respectiva versão do *kernel* podem ser encontradas no arquivo */usr/include/asm/unistd.h* (Mitchell et al., 2001). O conjunto de *system calls* encontradas em um sistema está diretamente relacionado com o sistema operacional e arquitetura utilizada.

Para fins de observação, uma estratégia comum se baseia em tratar o programa (ou aplicação) a ser observado como uma caixa preta, cujo funcionamento e definições internas são desconhecidas e desconsideradas, emitindo algum tipo de saída. É preferível que esta saída seja uma característica dinâmica do programa; embora o código do programa e suas definições seja a sua representação mais fiel, é preciso que este código seja executado para que reproduza alguma saída ou resultado no sistema. Se for considerado o processo do programa em execução como uma caixa preta, não é necessário conhecimento especializado do funcionamento interno do mesmo, é possível inferir tais características observando seu comportamento. Com base nessas considerações, as *system calls* são um conteúdo observável natural de processos do *host*, pois processos realizam o acesso à recursos do sistema através do mecanismo subjacente de *system calls* (Hofmeyr et al., 1998).

Devido à posição em que as *system calls* são encontradas, este recurso acaba sendo poderoso e tem direta influência no sistema. Suas funcionalidades permitem o desligamento total do sistema, alocação de recursos ou também impedir que outros usuários obtenham acesso aos mesmos. Desta forma, a arquitetura dos sistemas modernos acaba implementando camadas de segurança, que especificam alguns níveis de segurança que determinam quais tipos de programas podem interagir com determinados recursos. Este tipo de chamada contém restrições onde apenas processos executando com privilégios de superusuário podem executá-las (Mitchell et al., 2001).

Dentro do conceito de segurança de sistemas, temos conhecimento de variadas formas de ameaça à segurança de um *host*, com diversos tipos de alvos e inúmeras técnicas que podem ser utilizadas para comprometer um sistema (Garfinkel et al., 2004). Esta diversidade de possíveis ameaças torna complexo trabalhar em medidas defensivas, especialmente para aquelas que tem uma ampla aplicação. Apesar das diferentes abordagens utilizadas, estes ataques compartilham uma característica, eles geralmente exploram a interface de *system calls* para operações maliciosas. É apenas através desta interface que uma aplicação comprometida pode, por exemplo, escrever em disco ou enviar dados pela rede (Rajagopalan et al., 2006).

É possível então identificar e prevenir possíveis danos realizando o monitoramento de chamadas de sistema utilizadas por um processo, e lançar ações preventivas como por exemplo cancelar a chamada ou terminar o processo em questão. Além disso, interceptar *system calls* pode aumentar significativamente a eficácia de técnicas de detecção de intrusão *offline* que fazem uso de dados de auditoria do sistema, porque *logs* de auditoria do sistema geralmente não fornecem toda a informação necessária para a detecção de intrusão (Jain e Sekar, 2000).

2.3.1 STIDE

O primeiro método de HIDS baseado em *system calls* foi apresentado por Forrest et al. (Forrest et al., 1996) e também nos trabalhos derivados seguintes (Hofmeyr et al., 1998; Warrender et al., 1999), também conhecido como *Sequence Time-Delay Embedding* (STIDE). Este método é inspirado no sistema imunológico natural de organismos, e é considerado simples e eficiente para uso em possíveis implementações de tempo real. Nesta abordagem, os parâmetros das *system calls* são removidos a fim de reduzir a carga de trabalho e obter o melhor resultado utilizando apenas os identificadores das chamadas (Liu et al., 2018).

Segundo o estudo (Forrest et al., 1996), todo programa especifica implicitamente um conjunto de sequências de *system calls* que pode produzir, que são determinadas pelos possíveis

caminhos de execução que podem ser adotados. Para qualquer programa não trivial, os subconjuntos de chamadas serão enormes, e é provável que qualquer execução deste mesmo programa gere uma sequência de chamadas não observada anteriormente. Entretanto, subconjuntos locais de sequências curtas aparentam manter uma consistência notável, o que sugere uma definição simples de *self*, ou comportamento normal.

O funcionamento principal do método é descrito em dois passos: Primeiro são extraídos *traces* de *system calls* que representam comportamento normal, e é construída uma base de comportamento normal contendo sequências curtas de chamadas, obtidas através de uma janela deslizante aplicada no conjunto completo. Em seguida, são obtidos *traces* de execuções que podem conter atividades anômalas, e assim como no primeiro passo, são extraídas sequências curtas de chamadas utilizando uma janela deslizante (de mesmo tamanho), e então, este conjunto de sequências é comparado ao conjunto da base de comportamento normal em busca de discrepâncias.

2.4 VIRTUALIZAÇÃO BASEADA EM CONTÊINER

Embora o conceito de contêiner tenha sido popularizado apenas nos últimos anos, esta é uma técnica introduzida já na década de 80, para realizar um "isolamento" de *software* através da ferramenta *chroot* em sistemas Linux. O contêiner Linux é uma tecnologia de virtualização em nível de sistema operacional, que fornece isolamento e contenção para um ou mais processos de forma leve, onde é possível executar diversos contêineres simultaneamente, compartilhando o mesmo *kernel*. O isolamento é obtido principalmente por dois mecanismos: *namespace* e *cgroup* (Lin et al., 2018).

Do ponto de vista do usuário, cada contêiner aparenta executar exatamente como se fosse um sistema operacional completo, o que ocorre graças à *Linux Container Virtualization* (LCV), que é uma tecnologia de virtualização que permite a criação de contêineres no Linux, onde diferentes aplicações podem ser executadas. Além disso, devido a um sistema de versionamento, a LCV permite resolver as dependências de *software* de maneira flexível. É um paradigma de virtualização em nível de sistema operacional, que permite múltiplas instâncias do mesmo SO no espaço de usuário, enquanto compartilha o *kernel* do sistema subjacente, além de facilitar tarefas de *setup*, configuração, otimização e administração de recursos. Soluções populares para virtualização baseada em contêineres incluem Docker, LXC, lmctfy e OpenVZ (Celesti et al., 2016).

A virtualização de recursos consiste em usar uma camada de *software* intermediária acima de um sistema subjacente, com o objetivo de fornecer abstrações de múltiplos recursos virtuais (Xavier et al., 2013). Contêineres conseguem oferecer uma abordagem de virtualização leve e objetiva em comparação com os *hypervisors*, que permitem a abstração do *hardware* e alocação de recursos em um ambiente isolado conhecido como máquina virtual.

Os *hypervisors* realizam a virtualização de *hardware* e *drivers* de dispositivos, gerando uma alta sobrecarga. Em contraste, contêineres evitam essa sobrecarga implementando isolamento de processos em nível de sistema operacional. Uma única instância de um contêiner reúne a aplicação com todas as suas dependências, e executa como um processo isolado no espaço de usuário no sistema operacional subjacente, onde o *kernel* do SO é compartilhado entre todos os contêineres executando (Morabito, 2017). Consequentemente, por conta de tal característica, todos os sistemas convidados precisam ser baseados na mesma arquitetura de *hardware* e *kernel*. Por exemplo, essa técnica permite usar a mesma plataforma de *hardware* para executar múltiplas aplicações no mesmo servidor, onde cada aplicação estará isolada das outras, dando a impressão de ser a única executando com o *kernel* (Abeni et al., 2019). A Figura 2.1 ilustra uma

comparação de arquiteturas entre a virtualização baseada em contêiner e a virtualização baseada em *hypervisor*.

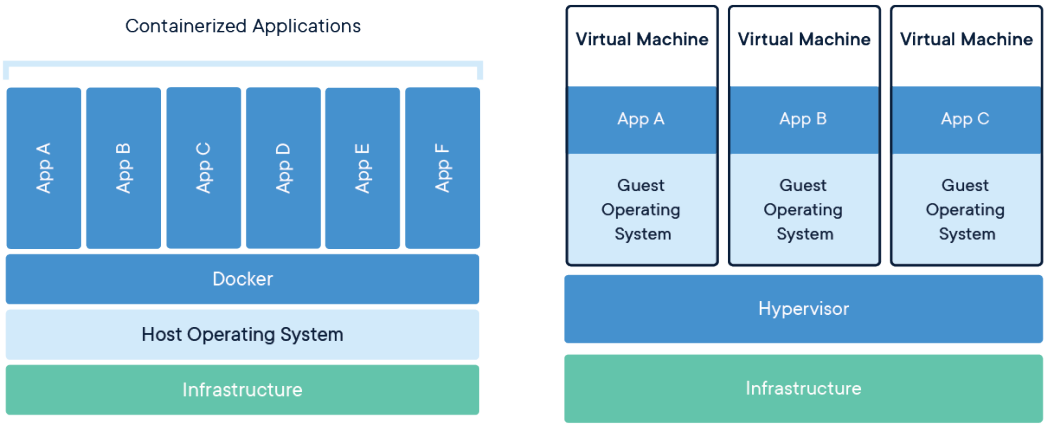


Figura 2.1: Comparação entre Contêiner e Máquina Virtual

Contêineres Linux, em particular, empregam duas características chave (Sharma et al., 2016):

Cgroups: Conhecidos como *control groups*, são mecanismos disponibilizados pelo *kernel* para controlar a alocação de recursos para grupos de processos. Cgroups existem para cada tipo majoritário de recurso: CPU, memória, interface de rede, entrada/saída bloqueante e dispositivos. A alocação de recursos para cada tipo pode ser controlada individualmente, permitindo especificar os limites para cada processo ou grupo de processos.

Namespaces: Um *namespace* fornece uma abstração para um recurso do *kernel* que faz com que o contêiner aparente ter a sua própria instância isolada deste recurso. No Linux, existem *namespaces* para isolamento: IDs de processos, IDs de usuários, interfaces de rede, *hostnames*, etc.

Contêineres contam com sistemas operacionais já em execução como seu ambiente base, o que promove dois benefícios significantes: Primeiro, a utilização de recursos é mais eficiente, visto que se um contêiner não está executando nenhuma operação, também não está consumindo recursos, e pode requisitar o sistema operacional para satisfazer algumas ou todas as suas dependências. Em segundo lugar, contêineres são leves e simples de criar e desativar, não é necessário iniciar um sistema operacional inteiro, e pode apenas encerrar os processos executando em seu espaço isolado, bastante similar a iniciar ou encerrar uma aplicação qualquer (Merkel, 2014).

Processos executando em um contêiner são isolados dos processos executando no sistema operacional subjacente, ou de outros contêineres, mesmo assim todos os processos executam no mesmo *kernel*. Embora o isolamento fornecido seja relativamente forte, não é tão forte quanto pode ser aplicado por máquinas virtuais no nível do *hypervisor*, onde estas foram mais fortalecidas e se provaram em diferentes ambientes de alta disponibilidade, enquanto tecnologias atreladas a contêineres ainda enfrentam grandes mudanças frequentemente. O próprio *daemon* responsável pelo Docker se mostra como um potencial vetor de ataque pois executa com privilégios de *root* (Merkel, 2014).

O fato de contêineres executarem compartilhando o mesmo *kernel* revela um risco de segurança crítico. Se um processo dentro de um contêiner comprometer o *kernel*, todo o isolamento providenciado pelo contêiner se torna inválido. Por conta disso, são empregados diversos mecanismos de segurança do *kernel* para restringir os processos dentro do contêiner,

onde alguns exemplos de tais mecanismos são: Capability, Seccomp e mecanismos de controle de acesso mandatórios (Mandatory Access Control) (Lin et al., 2018).

2.4.1 Docker

Entre as ferramentas utilizadas para a virtualização baseada em contêineres, o Docker é notado como uma das mais populares nos últimos anos. O Docker, que surgiu como um projeto *open-source* no início de 2013, é uma plataforma aberta para desenvolver, carregar, e executar aplicações de forma isolada em contêineres. Embora a tecnologia de contêineres já tenha sido utilizada por mais de uma década, o Docker se destacou fornecendo características que tecnologias iniciais não possuíam. A ferramenta oferece interfaces para criar e controlar contêineres de forma simples e segura, além disso, é possível desenvolver e agrupar aplicações e suas dependências em contêineres que podem executar em diferentes plataformas sem modificações, e também consegue interagir com aplicações terceiras que auxiliam o processo de administração e distribuição de contêineres Docker (Bui, 2015).

O Docker utiliza uma arquitetura de cliente-servidor, onde o cliente Docker comunica com o *daemon* Docker, que é responsável pelo trabalho de construir, executar e distribuir os contêineres. O cliente e o *daemon* interagem através de uma API REST usando sockets UNIX ou uma interface de rede. O *daemon* escuta por requisições feitas à API e gerencia os objetos Docker como contêineres, imagens, volumes e redes (Docker, 2021).

Ao utilizar o Docker, é possível interagir com diferentes objetos da plataforma. Uma imagem Docker é um modelo com instruções para criar um contêiner Docker, é possível criar uma imagem que tem como base uma outra imagem Docker com customizações adicionais. Um usuário pode criar suas próprias imagens para uso ou utilizar imagens feitas por outros usuários e publicadas em um registro público. Para criar imagens é utilizado um *Dockerfile*, um arquivo contendo etapas para criar uma imagem, onde cada etapa adiciona uma camada na imagem. Um contêiner Docker por sua vez é uma instância executável de uma imagem Docker, é possível criar, executar, parar, mover ou remover um contêiner através da API ou utilizando o cliente (Docker, 2021).

O *daemon* é responsável por tarefas como controlar o nível de isolamento de contêineres, realizar atividades de monitoramento, e executar interfaces de linha de comando nos ambientes de contêineres em execução, além de administrar as imagens de contêineres: receber e enviar imagens em um repositório remoto (Docker Hub), construir imagens a partir de arquivos Dockerfile, realizar assinaturas, entre outras. O Docker Hub é um repositório online onde desenvolvedores podem fazer *upload* de imagens Docker e também permite que usuários possam baixar imagens, onde também estão disponíveis imagens oficiais de diversas aplicações (Combe et al., 2016). Cada contêiner Docker é definido por sua respectiva imagem, que descreve o conjunto de *software* necessitada pela aplicação, começando com o SO (Debian ou Alpine, por exemplo), com suas dependências e configurações necessárias.

A segurança relacionada ao ambiente isolado proporcionado pelo Docker depende de alguns componentes como: isolamento de processos, isolamento do sistema de arquivos, isolamento de dispositivos, isolamento de IPC (comunicação entre processos), isolamento de rede e limitação de recursos. O Docker consegue o isolamento de processos envolvendo os processos executando no contêiner em *namespaces* e limitando suas permissões e visibilidades para processos executando em outros contêiners e no *host* subjacente. Também utiliza *mount namespaces* para isolar a hierarquia dos sistemas de arquivos associados com diferentes contêineres. Recursos de *namespaces* também são utilizados para garantir o isolamento de IPC, e de rede, enquanto alguns mecanismos disponibilizados pelos *cgroups* são utilizados para garantir o isolamento de dispositivos e a limitação de recursos (Bui, 2015).

2.5 MACHINE LEARNING

Machine Learning (ML) é um ramo da área de algoritmos computacionais projetado para emular a inteligência humana através do aprendizado. No processo de aprendizado é utilizada a informação passada ao método, que comumente tem a forma de dados eletrônicos coletados e disponibilizados para análise. Estes dados podem estar na forma de conjuntos de treinamento de dados digitalizados, ou outros tipos de informações coletados através de interações com o ambiente. Técnicas baseadas em ML têm sido aplicadas em diversas áreas como mecanismos de busca, diagnósticos médicos, reconhecimento de texto e escrita, previsões, finanças, entre outras (Mohri et al., 2018; El Naqa e Murphy, 2015).

Um elemento importante utilizado neste contexto é o *dataset*, que é definido por um conjunto de dados que serão utilizados no aprendizado e teste de algoritmos de ML, onde os dados devem seguir um formato e estrutura que seja compatível com o algoritmo que será utilizado. Um *dataset* pode ser composto por (ou subdividido em) diferentes conjuntos de dados, onde é necessário conjuntos para o treinamento e teste do modelo, sendo que estes dados devem refletir o ambiente alvo.

De maneira geral, o processo de aprendizado de máquina se baseia em encontrar e descrever padrões estruturais em um *dataset* fornecido. A entrada deste processo é na forma de um conjunto de instâncias, onde uma instância se refere a um exemplo independente e individual contido no *dataset*. Cada instância é caracterizada por seus atributos, que medem diferentes aspectos da instância. A saída produzida é a representação do conhecimento adquirido, onde a maneira que os resultados específicos do processo de aprendizado são representados depende da abordagem utilizada. Existem diferentes tipos de aprendizados que podem ser aplicados, entre eles: Classificação, Clusterização, associação e predição numérica. No caso da classificação, envolve o aprendizado de um conjunto de exemplos pré categorizados, onde a partir destes é construído um conjunto de regras de classificação para classificar exemplos não vistos (Nguyen e Armitage, 2008).

Técnicas tradicionais de detecção e prevenção de intrusões possuem limitações para proteger totalmente sistemas e redes de ataques sofisticados em constante evolução. Além disso, sistemas baseados nestas técnicas tradicionais sofrem com falsos positivos e falsos negativos e com a falta de adaptação contínua às mudanças em comportamentos maliciosos. Entretanto, na última década, diversas técnicas de aprendizado de máquina tem sido aplicadas no problema de detecção de intrusão com o objetivo de melhorar a taxa de detecção e adaptabilidade, tais técnicas são frequentemente usadas para manter as bases de conhecimento atualizadas e compreensivas (Zamani e Movahedi, 2013). Portanto, técnicas de aprendizado de máquina são consideradas como métodos eficientes para melhorar taxas de detecção, reduzir taxas de alarmes falsos e, ao mesmo tempo, diminuir custos computacionais e de comunicação (Sultana et al., 2019).

2.5.1 Algoritmos

A escolha de qual algoritmo de aprendizado utilizar para a tarefa a ser realizada é um passo crítico. A classificação supervisionada por exemplo, é uma tarefa frequentemente encontrada em abordagens que fazem uso de ML, consequentemente, diversas técnicas foram desenvolvidas baseadas em conceitos de inteligência artificial, técnicas baseadas em *perceptrons* e estatística (Kotsiantis et al., 2007). Algumas subáreas e algoritmos de classificação supervisionados de ML podem ser encontrados a seguir.

Uma árvore de decisão (*decision tree*) é uma estrutura de árvore que possui folhas, que representam classificações e ramificações, que representam os conjuntos de características que levam a uma classificação. Um exemplar é classificado a partir do teste de seus valores de

atributos com as folhas da árvore de decisão (Buczak e Guven, 2015). O algoritmo *Random Forest* (RF) é um conjunto de árvores de decisão, e considera a saída de cada árvore antes de retornar uma resposta final (Apruzzese et al., 2018).

Existem também os algoritmos de redes neurais, como o *Multilayer Perceptron*(MLP), também referenciados como redes neurais artificiais. que consistem de uma rede neural composta por um conjunto de neurônios organizados em duas ou mais camadas. Durante a classificação, os sinais recebidos nos neurônios de entrada são propagados através da rede para determinar os valores de ativação nos neurônios de saída (Kotsiantis et al., 2007).

Sob o domínio de métodos estatísticos, se tratando de técnicas baseadas em instâncias, um exemplo de algoritmo bastante difundido é o *K-Nearest Neighbors* (KNN), que categoriza objetos baseado em sua proximidade em relação a suas características nos conjuntos de treinamento. Também vale citar o *Support Vector Machine*, que pertence a categoria de métodos de classificação discriminativos, comumente reconhecidos por serem mais precisos. O método de SVM é baseado no princípio de minimização de risco estrutural, onde a ideia é encontrar uma hipótese que garanta o menor erro (Khan et al., 2010).

É possível fazer uma distinção de métodos de classificação em relação ao número de classes em que os dados podem ser categorizados. No caso de métodos multi classe, ao contrário da classificação binária onde existem apenas duas classes para identificar as instâncias, o problema se baseia em classificar instâncias em três ou mais classes, e pode ser resolvido estendendo a classificação binária para alguns algoritmos (Aly, 2005).

Como alternativa aos métodos multi classe, existem métodos de classificação de uma classe, que aplicam somente os dados do conjunto normal para treinamento dos métodos, e são recomendados para classificação em um conjunto de dados desbalanceados (Zhang et al., 2015). Um *dataset* é considerado desbalanceado quando contém muito mais amostras de uma classe em comparação a outras classes. *Datasets* são desbalanceados quando pelo menos uma classe é representada por um número muito pequeno de amostras no conjunto de treinamento, enquanto as demais compõem a maioria. Neste cenário, os classificadores podem obter uma boa precisão com a classe que representa a maioria dos dados, porém não consegue obter resultados relevantes na classe com menor representatividade, devido à influência que a classe majoritária possui nos critérios de treinamento tradicionais (Ganganwar, 2012).

3 REVISÃO BIBLIOGRÁFICA

No contexto do estudo deste trabalho é comum encontrar na literatura trabalhos que abordam máquinas virtuais, visto que é uma técnica difundida e popular, o que inclui alguns estudos que utilizam *system calls* e técnicas de *machine learning* para a detecção de anomalias. O objetivo desta seção é reunir trabalhos presentes na literatura próximos ao contexto apresentado, que é a detecção de intrusão por anomalias em um ambiente de contêiner. O objetivo é apresentar estudos que utilizem *system calls* como dado base para avaliação e, possivelmente, utilizem algoritmos de *machine learning* para a detecção, e avaliar e comparar entre estudos as metodologias utilizadas, quais resultados foram obtidos e categorias dos dados utilizados.

3.1 DETECÇÃO DE INTRUSÃO EM CONTÊINERES

Observando o lado de um atacante, existem diversas possibilidades de como explorar a virtualização para extrair informações privadas dos usuários, lançar ataques DDoS ou escalar a invasão para múltiplas instâncias (Liu et al., 2018). A literatura destaca trabalhos voltados ao monitoramento em diferentes níveis, com o intuito de identificar estas rotinas maliciosas. Um dos estudos pioneiros no campo de detecção de intrusão baseada em *system calls* foi proposto por (Forrest et al., 1996). O estudo apresentou um método inspirado em mecanismos e algoritmos usados por sistemas imunológicos naturais, e observou que sequências curtas de *system calls* aparentavam manter uma notável consistência entre os muitos conjuntos de chamadas possíveis entre os caminhos de execução de um programa, o que inspirou o uso de sequências curtas para definir o comportamento normal do sistema para a detecção de anomalias. O estudo conseguiu detectar diversas intrusões comuns envolvendo as ferramentas *sendmail* e *lpr*.

(Wang et al., 2006) apresenta *Anagram*, um detector de anomalias que modela uma combinação de *n-grams* ($n > 1$) projetado para detectar conteúdos anômalos e "suspeitos" em pacotes de rede. O trabalho fornece uma comparação entre abordagens utilizando diferentes tamanhos de *n-grams* (ou tamanhos de janelas). Os autores demonstraram que uma estratégia semi-supervisionada utilizando o *Bloom Filter* obteve 100% de detecção com 0.006% de taxa de falso positivo em alguns casos.

Ao tratar de detecção de anomalias para HIDs, com máquinas virtuais como objeto de estudo, o trabalho de (Laureano et al., 2004) apresenta uma abordagem para fortalecer a segurança de ambientes baseados em máquinas virtuais aplicando técnicas de detecção de intrusão. A ideia principal se baseia em monitorar o sistema convidado, extraindo dados como sequências de *system calls* para análise externa, e também tomar ações no sistema convidado em resposta a intrusões, com a possibilidade de também interagir com um *firewall* usado pelo *host*. O trabalho também faz uso da classificação de ameaças de *system calls* para restringir processos considerados suspeitos, e em um teste de detecção de intrusão conseguiu detectar modificações feitas por todos os 6 *rootkits* utilizados no experimento. O trabalho de (Alarifi e Wolthusen, 2012) lida com uma VM como um único processo, e utilizou as *system calls* interceptadas entre a VM e o *host*. O trabalho utiliza a técnica de *Bag of System Calls* (BoSC) em conjunto com janelas deslizantes, onde a sequência de entrada é processada em *epochs*, onde uma janela de tamanho k é deslizada por cada *epoch*, armazenando as frequências. Uma *epoch* é considerada anômala se o número de mudanças nas frequências ultrapassa um certo limiar. Para uma janela de tamanho 10, esta técnica obteve 100% de *accuracy* e taxa de detecção, com 0% de taxa de falso positivo.

No contexto de detecção de intrusão por anomalias em ambiente de contêiner, o trabalho de (Abed et al., 2015b) aplica uma técnica que combina BoSC com a técnica de STIDE. A análise do comportamento do contêiner é feita após o encerramento do mesmo, com o auxílio de uma tabela contendo todas as *system calls* distintas com o respectivo número total de ocorrências. O método faz a leitura do fluxo de *system calls* por *epochs*, e desliza uma janela de tamanho 10 através de cada *epoch* produzindo uma BoSC para cada janela, esta que é utilizada para a detecção de anomalias, que por sua vez é declarada caso o número de disparidades da base de comportamento normal ultrapasse um *threshold* definido. O classificador utilizando esta técnica atingiu uma taxa de detecção de 100% e uma taxa de falso positivo de 0.58% para *epoch* de tamanho 5,000 e *threshold* de detecção de 10% do tamanho da *epoch*. A base do experimento não é disponibilizada.

Ainda neste contexto, (Flora e Antunes, 2019) apresenta uma análise preliminar de viabilidade para a detecção de intrusão por detecção de anomalia, focando nas tecnologias Docker e LXC. O artigo propõe uma arquitetura de análise e captura de *system calls*, a aplicação dos algoritmos STIDE e BoSC, e estuda o processo de treinamento em diferentes casos. O estudo treinou ambos algoritmos com tamanhos de janela variados de 3 a 6 *system calls*, e calculou a inclinação da curva de crescimento, que significa a taxa de novas janelas adicionadas à base de comportamento normal de cada classificador após um período de tempo, onde os resultados destacam um estado de aprendizado estável para o STIDE com janelas de tamanho 3 e 4 e de 3 a 6 para o BoSC, o que significa que a melhor configuração obtida foi utilizando janelas de tamanho 3 e 4. A base de dados utilizada para a validação e experimentação foi desenvolvida para o estudo mas também não é disponibilizada.

(Srinivasan et al., 2018) apresenta um modelo para identificação de anomalias em aplicações executando dentro do Docker. A estratégia consiste em utilizar *n*-gramas para identificar a probabilidade de ocorrência de um evento. O experimento consegue a acurácia de até 97% para o *dataset* UNM, ao qual acaba não sendo representativo para eventos e aplicações atuais ou ocorrentes no ambiente virtualizado.

O trabalho (Tien et al., 2019) propõe KubAnomaly, um sistema que performa detecção de anomalias na plataforma Kubernetes em tempo de execução. O trabalho implementa um módulo de monitoramento de contêineres para Kubernetes e abordagens de redes neurais para criar modelos de classificação que fortalecem sua capacidade de encontrar comportamentos anômalos como ataques em serviços *web* e vulnerabilidades. No trabalho são utilizados três *datasets* diferentes, reunindo dados privados, dados disponíveis publicamente e dados de experimentos reais. Foram realizados testes e avaliações com diferentes algoritmos de ML e os resultados mostraram que o modelo de detecção de anomalia proposto conseguiu detectar com sucesso ataques em serviços *web* com acurácia de 96%. Foram realizados também experimentos reais com um contêiner de exemplo onde a ferramenta detectou mais de 40 eventos anômalos. Os conjuntos de dados utilizados nos experimentos são disponibilizados pelos autores.

(Tunde-Onadele et al., 2019) conduz um estudo sobre diferentes técnicas de detecção de vulnerabilidades e avalia a eficácia na detecção de vulnerabilidades de segurança de aplicações executando em contêineres. O trabalho considera técnicas de detecção estáticas e dinâmicas, onde técnicas estáticas focam principalmente no escaneamento de imagens de contêineres, que podem detectar vulnerabilidades comparando as versões dos pacotes utilizados com bases CVE, enquanto técnicas dinâmicas realizam o monitoramento do comportamento do contêiner e detectam atividades anômalas em tempo de execução. O estudo reproduz 28 vulnerabilidades encontradas em imagens do *Docker Hub* e conduz um estudo comparativo com esquemas de detecção estáticos e dinâmicos. A ferramenta de detecção de vulnerabilidades de imagens Docker *CoreOS Clair* é utilizada, resultando na detecção de apenas 3 das 28 vulnerabilidades. O estudo

então utiliza um conjunto de esquemas de detecção dinâmica que envolvem o uso de algoritmos de detecção de anomalia não supervisionados. Esses métodos utilizam *system calls* para monitorar o comportamento dos contêineres, implementando um sistema de extração de características das chamadas para a aplicação em algoritmos como o *K-Nearest Neighbors*, *K-Means clustering*, uma combinação do KNN com *Principal component analysis*, e *Self-Organizing Maps*. Os resultados mostraram que foi possível detectar 22 das 28 vulnerabilidades utilizando a detecção de anomalias baseada em *Self-Organizing Maps* com uma taxa média de 1.7% falso positivo. Além disso, esquemas dinâmicos alcançaram mais de 20 segundos de vantagem na detecção para um grupo de ataques, e também que pode ser vantajoso combinar esquemas estáticos e dinâmicos, que puderam melhorar a cobertura de detecção para 24 vulnerabilidades.

Trabalho	**Estratégia utilizada**	**Objeto alvo**	**Dataset**
(Forrest et al., 1996)	STIDE (janela deslizante)	Processos linux	-
(Wang et al., 2006)	*n-grams* com auxílio do *Bloom Filter*	Redes	-
(Laureano et al., 2004)	STIDE (janela deslizante), ACLs	Máquina virtual	-
(Alarifi e Wolthusen, 2012)	BoSC com janela deslizante em *epochs*	Máquina virtual	-
(Abed et al., 2015b)	BoSC com janela deslizante em *epochs*	Contêiner	-
(Flora e Antunes, 2019)	Análise de viabilidade para STIDE e BoSC	Contêiner (Docker e LXC)	-
(Srinivasan et al., 2018)	*n-grams* para análise probabilística	Contêiner Docker	*dataset* UNM
(Tien et al., 2019)	ML utilizando *system calls* e recursos adicionais	Contêiner (mais especificamente para a plataforma Kubernetes)	Dados públicos e privados, conjunto de dados disponibilizado pelos autores
(Tunde-Onadele et al., 2019)	Detecção estática (vulnerabilidades de imagens Docker) e detecção dinâmica (algoritmos de *machine learning* utilizando *system calls*	Contêiner Docker	-

Tabela 3.1: Trabalhos relacionados

Na tabela 3.1 há um resumo dos trabalhos relacionados citados nessa seção. Na coluna Estratégia é descrito brevemente quais foram as técnicas utilizadas pelo estudo para alcançar os resultados, a coluna Objeto alvo indica qual é o objeto alvo do estudo, um contêiner ou máquina virtual por exemplo, e a coluna Dataset indica qual foi o conjunto de dados utilizado pelo estudo.

A partir da análise dos trabalhos relacionados é possível perceber que o estudo na linha de detecção de intrusão em contêineres tem se popularizado cada vez mais nos últimos anos, onde as *system calls* se mostraram um recurso importante nesse contexto. Nos trabalhos foi possível aproveitar técnicas mais antigas como análise de sequência e de frequência em conjunto com métodos populares de *machine learning*, conseguindo alcançar resultados relevantes. Apesar do desafio do isolamento, os trabalhos mostram que as *system calls* são um dado representativo

na detecção de intrusão em *hosts*, e apesar das diferentes abordagens para interceptação e tratamento das chamadas, em nenhum trabalho analisado foi proposto algum tipo de filtragem das chamadas utilizadas nos experimentos. Por enquanto é difícil encontrar na literatura *datasets* que disponibilizam dados de *system calls* voltados à detecção de intrusão em ambiente de contêiner, o que faz com que os trabalhos nessa área tenham que formar os próprios conjuntos de dados para os estudos, entretanto, a maioria não disponibiliza os dados coletados.

4 PROPOSTA

Considerando a literatura atual disponível, existem algumas limitações referente a pesquisas e estudos voltadas para detecção e análise de intrusão em ambientes conteinerizados. Entretanto, é possível encontrar fontes relacionadas a métodos tradicionais que são utilizados para a detecção de intrusão, estas que podem ser utilizadas para a comparação com novas técnicas utilizadas em contêineres. Tais adaptações estão se mostrando uma tendência promissora para pesquisas no contexto de segurança de contêineres (Hickman, 2018). Um exemplo de pesquisa desenvolvida neste contexto é o trabalho de (Tien et al., 2019), que implementou abordagens de redes neurais para desenvolver um detector de anomalias em um ambiente de contêineres e compara sua eficiência com outros algoritmos de *Machine Learning*.

A tecnologia de contêineres, embora sendo atualmente uma técnica popular e eficiente para instalar, executar e disponibilizar uma aplicação em um ambiente isolado, não consegue garantir um isolamento completo do contêiner em relação ao *host*. Isso faz com que contêineres se tornem uma possível porta de entrada de ataques direcionados ao sistema do *host*, e consequentemente para outras aplicações e contêineres executando no mesmo sistema (Sultan et al., 2019). Tais ataques podem ser originados de uma aplicação vulnerável executando em um contêiner, ou também podem vir de uma imagem "maliciosa" que pode vir a ser utilizada (Combe et al., 2016; Kwon e Lee, 2020).

Nesse contexto é possível listar três possibilidades para monitorar o comportamento de uma aplicação isolada em um contêiner, a primeira seria monitorar a aplicação dentro do ambiente isolado. Realizar o monitoramento de vulnerabilidades e comportamentos em nível de aplicação é uma rotina com uma carga de trabalho consideravelmente alta, o que pode exercer certa influência no funcionamento da mesma, além disso, pode comprometer o próprio sistema de detecção de intrusão (Abed et al., 2015a). A segunda opção seria realizar esse monitoramento de fora do contêiner, no *host* em que o mesmo está sendo executado, o que fornece uma certa proteção ao IDS, mas em contrapartida, pode ser difícil ter acesso a certos recursos, e também pode ser necessário separar o que é resultante da aplicação e o que é resultante da ferramenta de virtualização. Outra alternativa seria definir um contêiner privilegiado para realizar o monitoramento do alvo presente em um contêiner vizinho, que necessitaria das permissões necessárias para o escopo da aplicação monitorada.

Pensando nesses fatores, uma possível solução para este problema seria executar essas rotinas de fora do ambiente isolado, no sistema subjacente, e monitorar os processos do sistema que representam a execução do contêiner a ser analisado, utilizando como dado base algum recurso que seja compartilhado com o sistema do *host*, como neste caso, as *system calls* geradas pelos processos responsáveis pelo contêiner.

Como citado anteriormente, por ser uma virtualização em nível de sistema operacional, o *host* e o contêiner compartilham o mesmo *kernel* do sistema operacional, ou seja, ao extrair as *system calls* geradas pelo processo do contêiner, este conjunto também representa as instruções executadas pela aplicação de dentro do ambiente isolado. Isso faz com que as *system calls* sejam uma boa opção para a detecção de anomalias, pois representam todo o comportamento do contêiner, e consequentemente, tornam possível uma maneira de observar e analisar o que é executado pela aplicação isolada a partir do *host* subjacente, sem a necessidade de interferir com modificações neste ambiente isolado. Também é necessário elencar que um dos problemas nesta abordagem seria a dificuldade de filtrar as *system calls* emitidas pela ferramenta responsável pela virtualização das emitidas pela aplicação do contêiner.

No contexto de análise de *system calls* para detecção de anomalias, (Forrest et al., 1996) introduz um método de detecção baseado em janelas deslizantes de sequências de *system calls* que apresentou simplicidade e eficácia para a detecção de atividades maliciosas em tempo real, e é considerado como um dos trabalhos base no estudo de detecção de intrusão baseado em *system calls*. Outras abordagens utilizando tabelas de frequência (Abed et al., 2015a; Alarifi e Wolthusen, 2012), algoritmos de *machine learning* (Liao e Vemuri, 2002; Yuxin et al., 2011) e Cadeias de Markov (Wang et al., 2004) também apresentaram bons resultados referentes a taxa de detecção.

Visando comparar diferentes técnicas possíveis para a detecção de anomalias baseado em *system calls*, a abordagem proposta foca em uma aplicação executando em ambiente de contêiner, realizando uma observação a partir do *host* subjacente. Por fim, é feita uma avaliação da detecção de intrusão no ambiente.

4.1 ESTRATÉGIA

Além da análise e utilização de todos os dados coletados, devido ao grande volume de dados para este trabalho foi considerada uma estratégia para a redução de "ruído" do conjunto de *system calls* obtidas. Esta estratégia consistiu de classificar *system calls* de acordo com um nível de ameaça e descartar as chamadas que foram classificadas como inofensivas no processo de tratamento dos dados coletados para a análise.

Se tratando do alvo de estudo, o *Wordpress* foi a aplicação escolhida para observação em um ambiente de contêiner. A escolha é em decorrência à sua popularidade em meio a aplicações web e também por conta de recorrentes problemas de segurança e vulnerabilidades encontrados na plataforma, onde a possibilidade do uso de *plugins* e temas personalizados disponibilizados pela comunidade que geralmente possuem falhas de segurança, permite que atacantes explorem vulnerabilidades como *Cross-site Scripting* (XSS): um tipo de vulnerabilidade encontrada em aplicações *web* que permite a injeção de *scripts* que podem ser executados na aplicação do cliente; *SQL Injection*: onde é possível interferir com *queries* utilizadas pela aplicação no banco de dados para realizar alterações maliciosas no banco de dados; e *Remote Code Execution* (RCE), que permite a execução de código arbitrário remota a partir de uma falha de segurança em uma aplicação.

Para o processo de avaliação, cinco classificadores multi classe e dois classificadores de uma classe foram utilizados. Os métodos de ML foram: *Naive Bayes, K-Nearest Neighbors (KNN), Random Forest, Multilayer Perceptron (MLP), Ada Boost, One-Class SVM* e o *Isolation Forest*. Também foi proposto um *dataset* que reúne um conjunto de sequências de *system calls* representando o comportamento normal e anormal da aplicação alvo, este que foi utilizado para treinar e testar os classificadores.

4.2 COLETA DE DADOS

Para o processo de coleta dos dados que foram utilizados no experimento, a captura das *system calls* geradas pelo contêiner foi feita com auxilio da ferramenta *strace*, uma ferramenta para sistemas Linux utilizada para monitorar e retornar as interações entre os processos analisados e o *kernel*, o que inclui o conjunto de chamadas utilizadas, sinais e mudanças de estado. A ferramenta é executada com um parâmetro especificando o PID do processo a ser monitorado, neste caso o processo responsável por executar o contêiner onde se encontra o *Wordpress*, e com um parâmetro para redirecionar a saída da ferramenta para um arquivo específico, onde cada coleta diferente é direcionada a um arquivo correspondente.

Tais dados são gravados em arquivos de texto, onde cada linha representa uma *system call* ou sinal obtido, e são gravados sem nenhum *timestamp*, com o nome da *system call* obtida, junto com todos os seus argumentos e valores de retorno.

Foram feitas algumas modificações nas configurações do servidor Apache responsável por disponibilizar o *Wordpress* para a web, onde foi definido um número mínimo de processos filhos e *threads*. Tais modificações foram feitas na tentativa de garantir a sequência das *system calls* utilizadas, sem alterações e com todos os parâmetros e valores de retorno. Definido o método de coleta dos dados, para a detecção de anomalia é necessário definir uma base de comportamento normal, que contém sequências de *system calls* que representam execuções normais da aplicação alvo, e também é preciso definir uma base de comportamento anormal, esta que é composta por sequências de *system calls* que representam a aplicação sob ataque, ou influência de um comportamento mal intencionado.

Após a coleta inicial, ambas as bases são “reconstruídas", onde desta vez são formadas por janelas deslizantes, onde são realizados experimentos utilizando tamanhos de janela variando entre 3, 5, 7, 9, 11, 13 e 15, deslizadas pelos arquivos que contém as *system calls*. Esta técnica é apresentada em alguns métodos baseados em sequência, como o primeiro caso de estudo utilizando janelas deslizantes de (Forrest et al., 1996).

4.2.1 Métodos de Filtragem

Em um dos casos de estudo realizados, com o objetivo de obter resultados mais relevantes e representativos, foi considerada a eliminação de “ruído” do conjunto de dados obtidos para o treinamento e avaliação dos classificadores. Inspirado na análise de *system calls* presente no trabalho de (Bernaschi et al., 2002), onde há uma classificação de nível de ameaça do conjunto de *system calls* disponíveis, foi utilizada uma estrutura de dados que contém, além de um identificador único para cada chamada, uma classificação de nível de ameaça que cada *system call* pode oferecer ao sistema. Assim como no trabalho de referência, os níveis de ameaça vão de 1 a 4, onde:

- O nível 1 representa *system calls* que permitem o controle completo do sistema;
- O nível 2 representa *system calls* que podem ser utilizadas em ataques de DoS;
- O nível 3 representa *system calls* que podem ser utilizadas para subverter o processo responsável; e
- O nível 4 apresenta *system calls* consideradas como inofensivas.

Utilizando esta classificação como base, para o método de eliminação de ruído foram desconsideradas do processo de treino e avaliação todas as *system calls* identificadas como inofensivas, ou seja, com o nível de ameaça igual a 4. As chamadas de baixa ameaça classificadas pelo artigo podem ser encontradas na Tabela 4.1, que representa parte da estrutura definida por (Bernaschi et al., 2002).

Analisando as *system calls* presentes na Tabela 4.1, é possível notar que a maioria das chamadas listadas neste nível tem como funcionalidade o retorno de algum valor ou atributo sobre determinado arquivo, como no caso das chamadas *stat* e suas variações, ou sobre recursos do sistema como *getpid, getuid, gettimeofday*, por exemplo. Essa classe de chamadas não é utilizada para manipulação em arquivos, memória ou execução de comandos ou programas, e não realizam mudanças significativas no sistema, e portanto podem ser classificadas como inofensivas em relação a segurança do sistema, pois seu uso não representa uma ameaça concreta.

Tabela 4.1: *System calls* classificadas como inofensivas. Retirada de (Bernaschi et al., 2002).

4	I	oldstat, oldfstat, access, sync, pipe, ustat, oldstat, readlink, readdir, statfs, fstatfs, stat, getpmsg, lstat, fstat, oldname, bdflush, sysfS, getdents, fdatasync
	II	getpid, getppid, getuid, getgid, geteuid, getegid, acct, getpgrp, sgetmask, getrlimit, getrusage, getgroups, getpriority, sched_getscheduler, sched_getparam, sched_get_priority_min, sched_rr_get_interval, capget, getpid, getsid, getcwdm getresgid, getresuid
	III	get_kernel_syms, create_module, query_module
	IV	times, time, gettimeofday, getitimer
	V	sysinfo, uname
	VI	idle
	VII	break, ftime, mpx, stty, prof, ulimit, gtty, lock, profil

4.3 MÉTODOS DE AVALIAÇÃO

Com o objetivo de comparar diferentes técnicas possíveis para detecção de anomalia baseado em *system calls*, a abordagem proposta foca em uma aplicação executando em um contêiner e sendo observada do *host* subjacente. O objetivo é explorar técnicas de ML para identificar ataques, e entender como os sistemas de detecção de intrusão se comportam nessa perspectiva. Com a utilização da técnica de janela deslizante, é avaliado em como o tamanho da janela impacta nos resultados. Dois grupos de testes foram elaborados, no primeiro foram consideradas todas as *system calls* capturadas, e no segundo caso as chamadas são classificadas em níveis de ameaça, e as chamadas consideradas inofensivas foram desconsideradas.

5 AVALIAÇÃO DE RESULTADOS

Nesta seção são apresentados os resultados atingidos com os métodos de avaliação propostos. Primeiro é detalhado o ambiente de testes e como ocorreu a coleta dos dados reunidos para os experimentos. Por fim, são apresentados os resultados obtidos em conjunto com uma comparação e discussão dos resultados.

5.1 CONJUNTO DE TESTES

Os experimentos foram realizados em um ambiente Linux 5.4.44-1-MANJARO utilizando a distribuição Manjaro 20.0.3, o ambiente de virtualização escolhido para a realização dos testes foi o Docker na versão 19.03.11-ce. Para realizar a avaliação é necessário um *dataset* que represente o comportamento a ser estudado, desta forma foram elaboradas duas versões de conjuntos de dados[1] para serem utilizados nos testes, onde cada versão formou um *dataset* de comportamento normal e anormal a partir dos dados, obtidos pela captura de instâncias do Wordpress, o objeto alvo dos experimentos. Posteriormente foi aplicada a técnica de janela deslizante para gerar os modelos que são utilizados para o treinamento do classificador.

Para ambos *datasets*, dois grupos de testes foram elaborados, no primeiro caso todas as *system calls* são utilizadas, sem a aplicação de nenhum tipo de filtro, já para o segundo caso de teste chamadas classificadas como baixo nível de ameaça foram desconsideradas.

5.1.1 *Dataset* Versão 1

Na formação do primeiro conjunto de dados utilizado nos testes, o processo de coleta ocorreu como descrito na seção 4.2, como objeto alvo um *Wordpress* na versão 4.9.14 executando em um contêiner com as modificações mencionadas.

A simulação de comportamento normal do *Wordpress* contou com um conjunto de *hosts* dentro da própria rede. Foi optado por utilizar variados sistemas operacionais que seriam responsáveis por interagir com o *Wordpress*, com diferentes *hosts* Windows e Linux sendo utilizados. A simulação tentou explorar variadas rotinas e interações normais para a aplicação, como a criação de novos *posts* e comentários, e manutenções corriqueiras no blog de teste implantado como alvo do experimento.

Para a obtenção dos dados de comportamento anormal, foi realizada a exploração de uma vulnerabilidade identificada pelo código CVE-2019-9978. A vulnerabilidade afeta o *plugin* do *Wordpress Social Warfare* com versão menor ou igual a 3.5.3, e permite a execução de código arbitrário no alvo em uma funcionalidade que gerencia a importação de configurações. Foi capturado o comportamento de três execuções similares da exploração da vulnerabilidade para formar o conjunto de dados anormais.

Sobre esta primeira versão do conjunto de dados utilizados nos experimentos, é importante lembrar que a captura das chamadas foram realizadas a partir de execuções únicas, e salvas em arquivos únicos. O que significa que, no momento da simulação do comportamento normal, foi executado o *strace* para realizar a captura das chamadas e todas as simulações com o *Wordpress* foram realizadas a partir daquele momento, reunindo tudo em um único arquivo, e o mesmo processo aconteceu para a simulação do comportamento anômalo. Esse processo traz algumas desvantagens na qualidade e representatividade geral dos dados capturados, pois acaba

[1] A base pode ser encontrada em `https://github.com/gabrielruschel/hids-docker`.

gerando uma certa redundância de dados visto que muitas rotinas acabam gerando sequências de chamadas similares, e também não é o processo recomendado para a simulação destes comportamentos, pois não separa diferentes execuções.

5.1.2 *Dataset* Versão 2

Para a formação da segunda versão do conjunto de dados utilizados nos experimentos, o processo de coleta ocorreu como descrito na seção 4.2, desta vez utilizando como objeto alvo o *Wordpress* na versão 4.9.2 executando em um contêiner, com as mesmas modificações citadas anteriormente. Para esta versão, a simulação dos comportamentos normais e anormais ocorreu de forma diferente. Para a simulação de comportamento normal, foram elaboradas 5 rotinas de execução envolvendo atividades normais corriqueiras de interação com o blog, e cada rotina foi executada da mesma maneira 5 vezes, onde o fluxo de chamadas de cada execução foi redirecionado para um arquivo diferente, totalizando 25 arquivos com fluxos de comportamento normal.

A simulação do comportamento anômalo ocorreu de forma similar, onde foram exploradas 5 vulnerabilidades diferentes:

- CVE-2019-9978 – A mesma vulnerabilidade explorada na formação do comportamento anômalo da primeira versão do *dataset*;
- CVE-2020-25213 – O *plugin File Manager* (wp-file-manager) anterior à versão 6.9 permite o *upload* e execução código PHP arbitrário;
- CVE-2020-12800 – O *plugin Drag and Drop Multiple File Upload – Contact Form 7* anterior à versão 1.3.3.3 permite o *upload* de arquivos sem restrição, o que permite a execução de código PHP arbitrário;
- Vulnerabilidade encontrada no *plugin AIT CSV Import/Export* com versão menor ou igual a 3.0.3, que permite o *upload* e execução de código PHP, por conta de uma falha do *plugin* em verificar e validar os arquivos enviados; e
- Vulnerabilidade presente no *plugin Simple File List* (simple-file-list) com versão anterior à 4.2.3, que falha em validar extensões de arquivos ao renomear, permitindo o *upload* e execução de arquivos PHP.

Cada exploração foi executada 5 vezes e em cada execução o fluxo de chamadas foi redirecionado para um arquivo diferente, totalizando 25 arquivos representando o conjunto de dados anormais.

5.2 RESULTADOS

O primeiro conjunto de experimentos consiste da utilização das *system calls* sem nenhum tratamento. As chamadas foram representadas por identificadores numéricos, com o auxílio de uma tabela associando cada *system call* a um *id* único.

Ambas versões dos *datasets* são formadas pela junção de suas respectivas bases de comportamento, e possuem um *label* categorizando cada janela em “normal” ou “anormal”. Para a aplicação nos métodos de avaliação cada *dataset* foi dividido pela metade, onde a primeira parte foi utilizada para treinamento e a segunda parte para efetuar os testes.

O algoritmo KNN utilizou uma distância de observação com $n = 3$. Para a avaliação do desempenho de classificação dos algoritmos, foram utilizadas três métricas: *precision*, *recall*

e *f1-score*. *Precision* representa a razão entre as detecções corretamente previstas e todas as detecções que ocorreram, onde valores altos representam baixa ocorrência de falsos-positivos. Já o *recall* representa a fração de detecções identificadas dentro de todas as detecções possíveis. O *f1-score* combina os valores de *precision* e *recall* em um único resultado, que indica a qualidade geral do modelo.

Os resultados deste experimento utilizando a primeira versão do *dataset* podem ser encontrados na Tabela 5.1. O *Random Forest*, MLP e o KNN são os algoritmos com a melhor *precision* entre os algoritmos multi classe em relação ao conjunto de casos observados, apesar do *Random Forest* apresentar a menor variação dentre os tamanhos de janelas observados. Dentre todos os algoritmos o *recall* predomina com valores baixos, que devido a valores razoavelmente altos para *precision*, demonstra que os classificadores acertam na maioria dos rótulos previstos.

A avaliação do *f1-score* aponta um grande desbalanceamento de classes presente no conjunto de dados coletados, uma vez que temos muito mais dados normais que anormais. Portanto, a coleta de mais dados anormais, geração de dados sintéticos ou até mesmo o uso de classificadores que lidam com dados desbalanceados podem ajudar a melhorar a taxa de detecção como um todo.

Pensando em resolver o problema de base desbalanceada, a primeira ideia foi tentar utilizar uma técnica de *boosting*, o *AdaBoost*, que consiste em um conjunto de classificadores treinados com foco nas instâncias que apresentam maior dificuldade na classificação (Freund e Schapire, 1997), neste caso, nos exemplares de ataques. Para isso, foi utilizado o modelo *Ada Boost* com o *Random Forest* como classificador interno.

Finalmente, como outra alternativa aos métodos já apresentados, foram utilizados dois classificadores de uma única classe, estes que somente aplicam o conjunto normal para o treinamento dos métodos, já que eles servem para detectar anomalias com base em um comportamento normal e são extremamente recomendados para classificar dados muito desbalanceados (Zhang et al., 2015). Esta nova avaliação destaca um valor relativamente adequado para a *precision* e *f1-score* principalmente para o *one-class SVM*, que apresentou melhores resultados gerais.

Com o objetivo de aperfeiçoar os resultados, um método de seleção de chamadas foi definido. Inspirado na análise de *system calls* de (Bernaschi et al., 2002), em que uma estrutura de dados foi criada contendo, além do identificador único, uma classificação de nível de ameaça que cada *system call* pode oferecer ao sistema. Os níveis vão de 1 a 4, onde o nível 1 representa *system calls* que permitem controle completo do sistema, o nível 2 representa *system calls* que podem ser utilizadas para ataques de DoS, o nível 3, *system calls* usadas para subverter o processo responsável, e por fim, *system calls* do nível 4, que são classificadas como inofensivas. As chamadas de baixa ameaça classificadas pelo artigo podem ser encontradas na Tabela 4.1, que representa parte da estrutura definida por (Bernaschi et al., 2002).

A partir das *system calls* apresentadas na Tabela 4.1, é realizada a filtragem das mesmas na formação do conjunto de janelas, onde todas as *system calls* que foram classificadas como inofensivas são descartadas, e portanto não são incluídas nas janelas tanto da base de comportamento normal quanto da base de comportamento anômalo. No caso do conjunto de dados da primeira versão do *dataset*, apesar de estar desconsiderando *system calls* de apenas um dos 4 grupos, após esta medida, o número de janelas geradas para todos os casos, em comparação ao experimento anterior, caíram pela metade. Com exceção dessa filtragem, todos os processos de avaliação são idênticos à avaliação com todas as chamadas.

A Tabela 5.2 apresenta os resultados para o conjunto de classificadores avaliados com os dados filtrados da primeira versão do *dataset*. Uma diferença não significativa é observada no *Naive-Bayes*, e uma pequena evolução no f1-score do KNN que destaca uma pequena melhora em decorrência do tratamento das chamadas.

Tabela 5.1: Desempenho dos algoritmos de ML considerando todas as chamadas - *Dataset* Versão 1

Tipo	**Classificador**	**Metrica**	**Largura da Janela**						
			3	**5**	**7**	**9**	**11**	**13**	**15**
Multi classe	Naive-Bayes	*precision*	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		recall	0.01	0.03	0.03	0.04	0.04	0.04	0.05
		f1-score	0.00	0.01	0.01	0.01	0.01	0.01	0.01
	KNN	*precision*	0.77	0.50	0.63	0.37	0.55	0.68	0.50
		recall	0.03	0.02	0.02	0.01	0.02	0.03	0.02
		f1-score	0.06	0.04	0.05	0.03	0.04	0.05	0.04
	RandomForest	*precision*	0.69	0.76	0.79	0.76	0.73	0.79	0.82
		recall	0.03	0.04	0.05	0.07	0.08	0.07	0.09
		f1-score	0.06	0.07	0.10	0.14	0.15	0.14	0.17
	MLP	*precision*	0.00	1.00	0.54	0.81	0.70	0.68	0.77
		recall	0.00	0.00	0.00	0.01	0.01	0.03	0.04
		f1-score	0.00	0.00	0.01	0.02	0.02	0.06	0.09
	Ada Boost	*precision*	0.75	0.94	0.94	0.85	0.80	0.88	0.96
		recall	0.03	0.04	0.05	0.07	0.07	0.07	0.08
		f1-score	0.06	0.08	0.11	0.13	0.14	0.14	0.14
Uma classe	One-class SVM	*precision*	0.99	0.99	0.99	0.99	0.99	0.99	0.99
		recall	0.98	0.97	0.96	0.97	0.97	0.97	0.97
		f1-score	0.99	0.98	0.98	0.98	0.98	0.98	0.98
	Isolation Forest	*precision*	0.99	0.99	0.99	0.99	0.99	0.99	0.99
		recall	0.88	0.90	0.88	0.89	0.90	0.90	0.91
		f1-score	0.93	0.94	0.93	0.94	0.94	0.94	0.95

Uma evolução pequena, mas significativa é observada para os algoritmos *Random Forest* e MLP, observada pelo crescimento no valor do *f1-score*. Tal resultado representa que a redução do número de *system calls* ajuda a fazer com que o classificador identifique melhor os comportamentos maliciosos, dado que a representação fica menos esparsa, necessitando de um menor conjunto de dados para identificar os padrões de cada classe.

Novamente foram realizados testes com o *Ada Boost* com o objetivo de melhorar o resultado de classificação geral, o que é de fato observado em seus resultados, chegando a ter *precision* próximo de 100% (nenhum falso positivo), e com um pequeno aumento no *recall* (diminuindo falso negativos). Na avaliação de uma única classe é possível destacar novamente um valor de *recall* elevado, diferente dos classificadores multi classe, resultando em um baixo falso negativo, isto é, os classificadores de uma classe conseguem detectar anomalias muito melhor que os classificadores multi classe em geral, provavelmente devido ao grande desbalanceamento de classes desta primeira versão do *dataset* (cenário que favorece esse tipo de classificador).

Referente à segunda versão do *dataset*, os conjuntos de testes ocorreram de forma similar, onde os mesmos algoritmos foram utilizados, com as mesmas métricas de avaliação em relação aos testes do primeiro *dataset*, porém com a diferença que, nos testes utilizando a segunda versão do conjunto de dados, cada algoritmo avaliado foi executado 10 vezes com amostras de treino e teste diferentes, onde os resultados de cada métrica são a média das 10 execuções. Para este conjunto de testes o *dataset* também foi dividido na metade para treino e teste, e a abordagem com a filtragem de *system calls* inofensivas também foi utilizada.

Os algoritmos multi classe foram executados com os mesmos parâmetros nos experimentos para a segunda versão do *dataset*. No caso dos algoritmos de uma classe, com os resultados obtidos com a primeira versão, foram definidos parâmetros diferentes com o objetivo

Tabela 5.2: Desempenho dos algoritmos de ML com filtragem de chamadas - *Dataset* Versão 1

Tipo	Classificador	Metrica	Largura da Janela						
			3	**5**	**7**	**9**	**11**	**13**	**15**
Multi classe	Naive-Bayes	*precision*	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		recall	0.04	0.08	0.13	0.13	0.14	0.17	0.21
		f1-score	0.01	0.01	0.01	0.01	0.01	0.01	0.01
	KNN	*precision*	0.35	0.66	0.62	0.52	0.57	0.73	0.62
		recall	0.01	0.05	0.04	0.02	0.06	0.05	0.05
		f1-score	0.03	0.09	0.09	0.05	0.11	0.10	0.10
	RandomForest	*precision*	0.74	0.90	0.98	0.84	0.72	0.89	0.83
		recall	0.07	0.16	0.15	0.18	0.17	0.19	0.18
		f1-score	0.13	0.27	0.27	0.30	0.28	0.32	0.30
	MLP	*precision*	0.0	0.0	0.71	0.76	0.86	0.80	0.71
		recall	0.0	0.0	0.03	0.06	0.06	0.07	0.11
		f1-score	0.0	0.0	0.06	0.12	0.12	0.12	0.20
	Ada Boost	*precision*	0.75	1.00	1.00	0.96	0.87	1.00	1.00
		recall	0.09	0.17	0.15	0.17	0.16	0.18	0.17
		f1-score	0.16	0.29	0.26	0.29	0.27	0.31	0.29
Uma classe	One-class SVM	*precision*	0.99	0.99	0.99	0.99	0.99	0.99	0.99
		recall	0.97	0.95	0.97	0.95	0.95	0.96	0.95
		f1-score	0.98	0.97	0.99	0.97	0.97	0.97	0.97
	Isolation Forest	*precision*	0.99	0.99	0.99	0.99	0.99	0.99	0.99
		recall	0.87	0.86	0.87	0.90	0.89	0.91	0.89
		f1-score	0.93	0.92	0.92	0.94	0.94	0.95	0.94

de obter resultados melhores com a segunda versão, visto que esta versão possui dados mais representativos e com um balanceamento melhor entre as duas classes.

Os resultados dos experimentos com a segunda versão do *dataset* considerando todas as chamadas podem ser encontrados na tabela 5.3. No geral, o experimento apresentou resultados adequados, com precisão e acurácia acima de 90% para os classificadores multi classe para todos os tamanhos de janela, onde a diferença de desempenho entre estes classificadores é mínima. Novamente observando apenas a variação no *f1-score* com o crescimento do tamanho de janela não é possível identificar uma preferência de melhor caso ou tamanho ideal, mas é possível observar que os tamanhos 3 e 5 apresentam os valores mais "instáveis" em relação aos outros tamanhos de janela, e que os algoritmos *Random Forest* e *Ada Boost* apresentam os melhores resultados no geral.

É possível perceber uma melhora considerável de resultados dos algoritmos multi classe em comparação ao mesmo experimento feito com os dados da primeira versão do *dataset*. É possível atribuir esta melhora de resultados principalmente à composição da segunda versão do *dataset*, que possui um melhor balanceamento entre dados normais e anormais, e também com melhores exemplares de comportamentos anormais, obtidos a partir de ataques mais diversos e complexos. Estes fatores combinados ao fato que os dados foram capturados de uma forma melhor estruturada e com uma melhor distinção de cada execução na segunda versão do *dataset* também explicam uma menor variação entre os resultados das janelas.

Na tabela 5.4 podemos encontrar os resultados dos testes realizando a filtragem de chamadas utilizando a segunda versão do *dataset*. Comparando com os resultados obtidos considerando todas as chamadas da segunda versão, é possível notar uma melhora pequena mas considerável nos resultados, inclusive do *f1-score*, o que demonstra uma melhor adequação de resultados de todos os classificadores em comparação aos obtidos nos experimentos anteriores, o

Tabela 5.3: Desempenho dos alogritmos de ML considerando todas as chamadas - *Dataset* Versão 2

Tipo	Classificador	Metrica	Largura da Janela						
			3	**5**	**7**	**9**	**11**	**13**	**15**
Multi classe	Naive-Bayes	*precision*	0.26	0.26	0.26	0.30	0.31	0.31	0.30
		recall	0.04	0.07	0.10	0.12	0.14	0.16	0.16
		f1-score	0.08	0.11	0.14	0.17	0.19	0.21	0.21
		accuracy	0.78	0.78	0.77	0.77	0.77	0.77	0.76
	KNN	*precision*	0.84	0.90	0.90	0.88	0.91	0.90	0.87
		recall	0.63	0.71	0.74	0.76	0.76	0.77	0.79
		f1-score	0.71	0.79	0.81	0.81	0.83	0.83	0.83
		accuracy	0.90	0.92	0.93	0.93	0.94	0.94	0.93
	Random Forest	*precision*	0.99	0.98	0.98	0.98	0.98	0.98	0.98
		recall	0.58	0.70	0.73	0.74	0.75	0.76	0.77
		f1-score	0.73	0.82	0.83	0.84	0.85	0.86	0.86
		accuracy	0.91	0.94	0.94	0.94	0.95	0.95	0.95
	MLP	*precision*	0.91	0.89	0.90	0.92	0.91	0.90	0.90
		recall	0.54	0.64	0.67	0.68	0.69	0.69	0.70
		f1-score	0.68	0.74	0.77	0.78	0.78	0.78	0.79
		accuracy	0.90	0.91	0.92	0.92	0.92	0.92	0.92
	Ada Boost	*precision*	0.99	0.98	0.98	0.98	0.98	0.98	0.98
		recall	0.58	0.70	0.73	0.74	0.75	0.76	0.77
		f1-score	0.73	0.82	0.83	0.84	0.85	0.86	0.86
		accuracy	0.91	0.94	0.94	0.94	0.95	0.95	0.95
Uma classe	One-class SVM	*precision*	0.39	0.43	0.44	0.50	0.52	0.51	0.56
		recall	0.04	0.07	0.44	0.11	0.11	0.11	0.13
		f1-score	0.08	0.12	0.09	0.18	0.19	0.19	0.22
		accuracy	0.79	0.80	0.15	0.80	0.80	0.80	0.81
	Isolation Forest	*precision*	0.44	0.40	0.38	0.37	0.34	0.34	0.35
		recall	0.35	0.30	0.29	0.24	0.22	0.20	0.19
		f1-score	0.39	0.34	0.33	0.29	0.27	0.25	0.25
		accuracy	0.79	0.77	0.77	0.77	0.76	0.77	0.77

que novamente foi atingido graças à redução do número de *system calls*, diminuindo o "ruído" de chamadas que não eram relevantes para a classificação. Novamente os classificadores que atingiram os melhores resultados foram o *Random Forest* e o *Ada Boost*.

Não é possível comparar os resultados dos algoritmos de uma classe da segunda versão com os resultados obtidos com a primeira versão do conjunto de dados, visto que ambos os algoritmos utilizaram parâmetros diferentes. Nos dois experimentos com a segunda versão do *dataset*, os algoritmos de uma classe apresentaram resultados abaixo dos algoritmos multi classe e com valores mais instáveis, visto que foram utilizados dados mais balanceados e representativos.

5.3 DISCUSSÃO

Para a discussão dos resultados é importante frisar a diferença entre as duas versões dos *datasets* utilizados nos experimentos. Na primeira versão, cada classe de comportamento (normal e anômala) foi coletada em uma execução única, onde todos os comportamentos coletados foram reunidos em um único arquivo para cada classe. A coleta para a classe anômala não foi muito expressiva, visto que foi utilizado apenas um ataque utilizando execuções diferentes. Já na segunda versão, foram utilizados mais exemplares de ataques diferentes, e também cada execução

Tabela 5.4: Desempenho dos algoritmos de ML com filtragem de chamadas - *Dataset* Versão 2

Tipo	**Classificador**	**Metrica**	**Largura da Janela**						
			3	**5**	**7**	**9**	**11**	**13**	**15**
Multi classe	Naive-Bayes	*precision*	0.33	0.34	0.35	0.34	0.34	0.34	0.35
		recall	0.03	0.06	0.09	0.11	0.13	0.16	0.18
		f1-score	0.06	0.10	0.14	0.17	0.19	0.22	0.24
		accuracy	0.72	0.72	0.71	0.71	0.70	0.69	0.69
	KNN	*precision*	0.88	0.96	0.95	0.95	0.96	0.95	0.97
		recall	0.71	0.82	0.85	0.86	0.86	0.87	0.88
		f1-score	0.78	0.88	0.90	0.90	0.91	0.91	0.92
		accuracy	0.89	0.94	0.95	0.95	0.95	0.95	0.96
	Random Forest	*precision*	0.99	0.99	0.99	0.99	0.99	0.99	0.99
		recall	0.68	0.81	0.85	0.86	0.87	0.87	0.88
		f1-score	0.80	0.89	0.91	0.92	0.92	0.93	0.93
		accuracy	0.91	0.95	0.95	0.96	0.96	0.96	0.96
	MLP	*precision*	0.92	0.94	0.95	0.96	0.95	0.96	0.96
		recall	0.64	0.77	0.81	0.82	0.82	0.83	0.83
		f1-score	0.75	0.85	0.87	0.88	0.88	0.89	0.89
		accuracy	0.89	0.93	0.94	0.94	0.94	0.94	0.94
	Ada Boost	*precision*	0.99	0.99	0.99	0.99	0.99	0.99	0.99
		recall	0.68	0.81	0.85	0.86	0.87	0.87	0.88
		f1-score	0.80	0.89	0.91	0.92	0.92	0.93	0.93
		accuracy	0.91	0.95	0.95	0.96	0.96	0.96	0.96
Uma classe	One-class SVM	*precision*	0.45	0.60	0.53	0.50	0.55	0.55	0.59
		recall	0.08	0.10	0.10	0.10	0.12	0.14	0.16
		f1-score	0.13	0.17	0.17	0.17	0.20	0.23	0.26
		accuracy	0.72	0.74	0.74	0.73	0.74	0.74	0.75
	Isolation Forest	*precision*	0.55	0.45	0.36	0.37	0.37	0.38	0.40
		recall	0.44	0.29	0.20	0.17	0.16	0.17	0.17
		f1-score	0.49	0.35	0.26	0.23	0.23	0.23	0.24
		accuracy	0.76	0.72	0.69	0.70	0.70	0.71	0.71

de coleta foi repetida 5 vezes, onde cada execução separou os dados coletados em um arquivo diferente.

A partir dos resultados relatados é possível notar que, no caso dos algoritmos multi classe utilizados nos experimentos, não foram obtidos resultados expressivos na utilização da primeira versão do *dataset*, visto que esta versão possuía uma quantidade relativamente maior de dados da classe normal do que dos dados da classe anômala, que careciam de melhores amostras. Embora tenha sido notada uma melhora após a filtragem de chamadas, ainda não foi possível conseguir resultados significativos para esta classe de algoritmos com a primeira versão do *dataset* (o melhor resultado obtido foi o *f1-score* de 0.32 do algoritmo Random Forest). No caso dos algoritmos de uma classe, foram obtidos os melhores resultados antes e após a filtragem de chamadas, atingindo um *f1-score* de 0.99 no One-class SVM.

Já se tratando dos resultados desta classe de algoritmo utilizando a segunda versão do *dataset*, é possível notar uma melhora considerável do *f1-score* logo nos resultados considerando todas as chamadas capturadas, onde foi obtido um *f1-score* de 0.86 com os algoritmos Random Forest e Ada Boost, e é possível conseguir valores ainda mais altos após as filtragens, atingindo o maior *f1-score* de 0.93 com Random Forest e Ada Boost, o que indica que reunir uma base de dados que seja expressiva tanto nos conteúdos de comportamento normal quanto anormal,

e bem estruturada nos métodos de coletas são fatores que podem contribuir positivamente no trabalho dos classificadores multi classe. Foi possível obter resultados relevantes com todos os tamanhos de janela testados, o que indica que esta faixa de tamanho pode ser o suficiente para detecções nesse contexto. É possível notar também que os valores do *f1-score* para esta classe de algoritmos na segunda versão do *dataset* são crescentes em função do tamanho da janela, onde um tamanho maior de janela nesse intervalo consegue um valor maior ou igual ao tamanho anterior, o que não é observado nos resultados da primeira versão do *dataset*.

Por fim, a partir da análise dos resultados, é possível concluir que para a detecção de anomalias utilizando classificadores de ML é importante utilizar uma base de dados que seja representativa no contexto a ser estudado, e que seja bem estruturada nos métodos de coleta utilizados. Não foi possível definir qual é o melhor tamanho de janela a ser utilizado, visto que diferentes tamanhos obtiveram resultados próximos na maioria dos experimentos, porém levando em consideração os algoritmos multi classe os melhores resultados foram obtidos com tamanhos de janela maiores que 3. Também foi possível identificar que houve uma melhora de resultados após a realização da filtragem de *system calls* avaliadas como inofensivas. Por mais que essa melhora tenha sido pequena, mostrou-se um ponto interessante para ser estudado e avaliado no futuro, com classificações diferentes e mais completas para chamadas encontradas em sistemas modernos.

6 CONCLUSÃO

Neste trabalho de conclusão de curso foi estudada a possibilidade e viabilidade da detecção de intrusão por anomalia com foco em um ambiente de contêiner, utilizando *system calls* organizadas em janelas deslizantes para definir o comportamento de um contêiner alvo, onde foi possível analisar e avaliar o desempenho de diferentes algoritmos de ML, e também comparar a diferença de resultados dos métodos quanto aos tamanhos de janelas utilizadas e também do tipo de *system calls* levadas em consideração para treinar e testar os classificadores.

Foi realizada a coleta de dados da classe normal e anômala para formar dois *datasets* diferentes utilizando duas versões diferentes de um Wordpress, o objeto alvo, executando em um contêiner. A primeira versão do *dataset* contou com dados coletados de uma forma mais simples e contou com apenas um exemplar de ataque ao Wordpress, enquanto na segunda versão, os dados foram coletados de uma maneira mais estruturada e contou com 5 exemplares de ataques diferentes ao objeto alvo. A partir dos dados coletados, são formados conjuntos de janelas deslizantes, com diferentes tamanhos a serem utilizados nos testes. Foram definidos dois conjuntos de teste a serem executados com ambos os *datasets* utilizando classificadores de ML com diferentes abordagens, no primeiro teste são utilizadas todas as *system calls* capturadas na formação das janelas deslizantes, e no segundo caso é utilizada uma filtragem de chamadas consideradas inofensivas, com o objetivo de melhorar o desempenho dos classificadores.

Foi constatado que é possível observar o comportamento de um contêiner a partir do *host* subjacente realizando o monitoramento das *system calls* emitidas pelo contêiner, que se mostraram um recurso valioso para a detecção de anomalia neste contexto. É possível perceber que esta é uma linha de pesquisa ganhando popularidade nos últimos anos, onde é possível encontrar estudos que utilizam diferentes abordagens tanto na etapa de detecção de anomalias quanto na parte de extração e tratamento de recursos do contêiner como dado base, encontrando resultados relevantes para este contexto, porém ainda é difícil encontrar trabalhos que disponibilizam os conjuntos de dados utilizados.

Na avaliação dos resultados obtidos pelos classificadores, comparando os resultados dos classificadores multi classe entre os dois *datasets* utilizados, é possível perceber uma melhora significativa nos resultados utilizando a segunda versão, o que indica que utilizar dados mais representativos e estruturados pode influenciar positivamente nos resultados dos classificadores. Ao analisar o desempenho com a primeira versão do conjunto de dados, foi possível notar que em um *dataset* com dados desbalanceados os classificadores de uma classe se destacam, obtendo resultados superiores em relação aos métodos multi classe. Para ambas as versões, foi possível identificar que houve uma melhora de resultados após a realização da filtragem de *system calls* avaliadas como inofensivas. Por mais que esta melhora tenha sido pequena, mostrou-se um ponto interessante a ser estudado e avaliado, utilizando uma nova avaliação de ameaça de chamadas, que consiga abordar um número maior de chamadas presentes em sistemas modernos.

Por fim, é possível elencar que o seguinte trabalho gerou três publicações, estas sendo:

1. (Castanhel et al., 2020a) (SBSeg 2020) – Este trabalho analisou e comparou métodos de ML para detecção de anomalia em um contêiner utilizando *system calls*, comparando os resultados em relação à diferentes tamanhos de janela e a filtragem de chamadas inofensivas, utilizando os dados da primeira versão do *dataset*;

2. (Castanhel et al., 2020b) (ERRC 2020) – Este trabalho realizou um estudo sobre o impacto do tamanho das janelas utilizadas para classificadores de ML de uma classe; e

3. (Castanhel et al., 2021) (ISCC 2021) – Este trabalho fez uso dos métodos de avaliação dos dois trabalhos anteriores utilizando algoritmos de ML multi classe com os dados da segunda versão do *dataset*.

REFERÊNCIAS

Abed, A. S., Clancy, C. e Levy, D. S. (2015a). Intrusion detection system for applications using linux containers. Em *International Workshop on Security and Trust Management*. Springer.

Abed, A. S., Clancy, T. C. e Levy, D. S. (2015b). Applying bag of system calls for anomalous behavior detection of applications in linux containers. Em *2015 IEEE Globecom Workshops (GC Wkshps)*, páginas 1–5. IEEE.

Abeni, L., Balsini, A. e Cucinotta, T. (2019). Container-based real-time scheduling in the linux kernel. *ACM SIGBED Review*, 16(3):33–38.

Ahmed, M., Mahmood, A. N. e Hu, J. (2016). A survey of network anomaly detection techniques. *Journal of Network and Computer Applications*, 60:19–31.

Alarifi, S. S. e Wolthusen, S. D. (2012). Detecting anomalies in iaas environments through virtual machine host system call analysis. Em *2012 International Conference for Internet Technology and Secured Transactions*. IEEE.

Aly, M. (2005). Survey on multiclass classification methods. *Neural Netw*, 19:1–9.

Anderson, J. P. (1980). Computer security threat monitoring and surveillance, james p. *Anderson Co., Fort Washington, PA*.

Apruzzese, G., Colajanni, M., Ferretti, L., Guido, A. e Marchetti, M. (2018). On the effectiveness of machine and deep learning for cyber security. Em *2018 10th international conference on cyber Conflict (CyCon)*, páginas 371–390. IEEE.

Arora, R., Parashar, A. e Transforming, C. C. I. (2013). Secure user data in cloud computing using encryption algorithms. *International journal of engineering research and applications*, 3(4):1922–1926.

Bace, R. e Mell, P. (2001). Nist special publication on intrusion detection systems. Relatório técnico, BOOZ-ALLEN AND HAMILTON INC MCLEAN VA.

Bernaschi, M., Gabrielli, E. e Mancini, L. V. (2002). Remus: a security-enhanced operating system. *ACM Transactions on Information and System Security (TISSEC)*.

Bridges, R. A., Glass-Vanderlan, T. R., Iannacone, M. D., Vincent, M. S. e Chen, Q. (2019). A survey of intrusion detection systems leveraging host data. *ACM Computing Surveys (CSUR)*.

Brown, D. J., Suckow, B. e Wang, T. (2002). A survey of intrusion detection systems. *Department of Computer Science, University of California*.

Buczak, A. L. e Guven, E. (2015). A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Communications surveys & tutorials*, 18(2):1153–1176.

Bui, T. (2015). Analysis of docker security. *arXiv preprint arXiv:1501.02967*.

Castanhel, G. R., Heinrich, T., Ceschin, F. e Maziero, C. A. (2020a). Detecção de anomalias: Estudo de técnicas de identificação de ataques em um ambiente de contêiner. Undergraduate Research Workshop - Brazilian Security Symposium (WTICG - SBSeg).

Castanhel, G. R., Heinrich, T., Ceschin, F. e Maziero, C. A. (2020b). Sliding window: The impact of trace size in anomaly detection system for containers through machine learning. Regional Workshop on Information Security and Computer Systems (WRSeg - ERRC).

Castanhel, G. R., Heinrich, T., Ceschin, F. e Maziero, C. A. (2021). Taking a peek: An evaluation of anomaly detection using system calls for containers. 26th IEEE Symposium on Computers and Communications (ISCC 2021).

Celesti, A., Mulfari, D., Fazio, M., Villari, M. e Puliafito, A. (2016). Exploring container virtualization in iot clouds. Em *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*, páginas 1–6. IEEE.

Chandola, V., Banerjee, A. e Kumar, V. (2009). Anomaly detection: A survey. *ACM computing surveys (CSUR)*.

Combe, T., Martin, A. e Di Pietro, R. (2016). To docker or not to docker: A security perspective. *IEEE Cloud Computing*.

Debar, H., Dacier, M. e Wespi, A. (1999). Towards a taxonomy of intrusion-detection systems. *Computer Networks*.

Denning, D. E. (1987). An intrusion-detection model. *IEEE Transactions on software engineering*, (2):222–232.

Deshpande, P., Sharma, S. C., Peddoju, S. K. e Junaid, S. (2018). Hids: A host based intrusion detection system for cloud computing environment. *International Journal of System Assurance Engineering and Management*.

Docker (2021). Docker overview. `https://docs.docker.com/get-started/overview/`.

El Naqa, I. e Murphy, M. J. (2015). What is machine learning? Em *machine learning in radiation oncology*, páginas 3–11. Springer.

Flora, J. e Antunes, N. (2019). Studying the applicability of intrusion detection to multi-tenant container environments. Em *2019 15th European Dependable Computing Conference (EDCC)*, páginas 133–136. IEEE.

Forrest, S., Hofmeyr, S. A., Somayaji, A. e Longstaff, T. A. (1996). A sense of self for unix processes. Em *IEEE Symposium on Security and Privacy*.

Freund, Y. e Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*

Ganganwar, V. (2012). An overview of classification algorithms for imbalanced datasets. *International Journal of Emerging Technology and Advanced Engineering*, 2(4):42–47.

Garfinkel, T., Pfaff, B., Rosenblum, M. et al. (2004). Ostia: A delegating architecture for secure system call interposition. Em *NDSS*.

Hickman, A. (2018). Container intrusions: Assessing the efficacy of intrusion detection and analysis methods for linux container environments.

Hodge, V. e Austin, J. (2004). A survey of outlier detection methodologies. *Artificial intelligence review*, 22(2):85–126.

Hofmeyr, S. A., Forrest, S. e Somayaji, A. (1998). Intrusion detection using sequences of system calls. *Journal of computer security*, 6(3):151–180.

Jain, K. e Sekar, R. (2000). User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. Em *NDSS*.

Khan, A., Baharudin, B., Lee, L. H. e Khan, K. (2010). A review of machine learning algorithms for text-documents classification. *Journal of advances in information technology*, 1(1):4–20.

Kotsiantis, S. B., Zaharakis, I. e Pintelas, P. (2007). Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160(1):3–24.

Kruegel, C. e Toth, T. (2003). Using decision trees to improve signature-based intrusion detection. Em Vigna, G., Kruegel, C. e Jonsson, E., editores, *Recent Advances in Intrusion Detection*, páginas 173–191, Berlin, Heidelberg. Springer Berlin Heidelberg.

Kumar, S. (2007). Survey of current network intrusion detection techniques. *Washington Univ. in St. Louis*.

Kumar, V. e Sangwan, O. P. (2012). Signature based intrusion detection system using snort. *International Journal of Computer Applications & Information Technology*, 1(3):35–41.

Kwon, S. e Lee, J. (2020). Divds: Docker image vulnerability diagnostic system. *IEEE Access*.

Laureano, M., Maziero, C. e Jamhour, E. (2004). Intrusion detection in virtual machine environments. Em *Proceedings. 30th Euromicro Conference, 2004.*, páginas 520–525. IEEE.

Liao, Y. e Vemuri, V. R. (2002). Using text categorization techniques for intrusion detection. Em *USENIX Security Symposium*.

Lin, X., Lei, L., Wang, Y., Jing, J., Sun, K. e Zhou, Q. (2018). A measurement study on linux container security: Attacks and countermeasures. Em *Proceedings of the 34th Annual Computer Security Applications Conference*, páginas 418–429.

Liu, M., Xue, Z., Xu, X., Zhong, C. e Chen, J. (2018). Host-based intrusion detection system with system calls: Review and future trends. *ACM Computing Surveys (CSUR)*.

Maziero, C. (2020). *Sistemas Operacionais: Conceitos e Mecanismos*. Editora da UFPR.

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux journal*.

Mishra, P., Pilli, E. S., Varadharajan, V. e Tupakula, U. (2017). Intrusion detection techniques in cloud environment: A survey. *Journal of Network and Computer Applications*, 77:18–47.

Mitchell, M., Oldham, J. e Samuel, A. (2001). *Advanced linux programming*. New Riders Publishing.

Mohri, M., Rostamizadeh, A. e Talwalkar, A. (2018). *Foundations of machine learning*. MIT press.

Morabito, R. (2017). Virtualization on internet of things edge devices with container technologies: a performance evaluation. *IEEE Access*, 5:8835–8850.

Nguyen, T. T. e Armitage, G. (2008). A survey of techniques for internet traffic classification using machine learning. *IEEE communications surveys & tutorials*, 10(4):56–76.

Patcha, A. e Park, J.-M. (2007). An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer networks*, 51(12):3448–3470.

Rajagopalan, M., Hiltunen, M. A., Jim, T. e Schlichting, R. D. (2006). System call monitoring using authenticated system calls. *IEEE Transactions on Dependable and Secure Computing*.

Sharma, P., Chaufournier, L., Shenoy, P. e Tay, Y. (2016). Containers and virtual machines at scale: A comparative study. Em *Proceedings of the 17th International Middleware Conference*, páginas 1–13.

Srinivasan, S., Kumar, A., Mahajan, M., Sitaram, D. e Gupta, S. (2018). Probabilistic real-time intrusion detection system for docker containers. Em *International Symposium on Security in Computing and Communication*, páginas 336–347. Springer.

Sultan, S., Ahmad, I. e Dimitriou, T. (2019). Container security: Issues, challenges, and the road ahead. *IEEE Access*.

Sultana, N., Chilamkurti, N., Peng, W. e Alhadad, R. (2019). Survey on sdn based network intrusion detection system using machine learning approaches. *Peer-to-Peer Networking and Applications*, 12(2):493–501.

Taha, A. e Hadi, A. S. (2019). Anomaly detection methods for categorical data: A review. *ACM Computing Surveys (CSUR)*, 52(2):1–35.

Tien, C.-W., Huang, T.-Y., Tien, C.-W., Huang, T.-C. e Kuo, S.-Y. (2019). Kubanomaly: Anomaly detection for the docker orchestration platform with neural network approaches. *Engineering Reports*.

Tunde-Onadele, O., He, J., Dai, T. e Gu, X. (2019). A study on container vulnerability exploit detection. Em *2019 IEEE International Conference on Cloud Engineering (IC2E)*, páginas 121–127. IEEE.

Wang, K., Parekh, J. J. e Stolfo, S. J. (2006). Anagram: A content anomaly detector resistant to mimicry attack. Em *International workshop on recent advances in intrusion detection*, páginas 226–248. Springer.

Wang, W., Guan, X.-H. e Zhang, X.-L. (2004). Modeling program behaviors by hidden markov models for intrusion detection. Em *Proceedings of 2004 International Conference on Machine Learning and Cybernetics*. IEEE.

Warrender, C., Forrest, S. e Pearlmutter, B. (1999). Detecting intrusions using system calls: Alternative data models. Em *Proceedings of the 1999 IEEE symposium on security and privacy (Cat. No. 99CB36344)*, páginas 133–145. IEEE.

Xavier, M. G., Neves, M. V., Rossi, F. D., Ferreto, T. C., Lange, T. e De Rose, C. A. (2013). Performance evaluation of container-based virtualization for high performance computing environments. Em *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, páginas 233–240. IEEE.

Yassin, W., Udzir, N. I., Muda, Z., Sulaiman, M. N. et al. (2013). Anomaly-based intrusion detection through k-means clustering and naives bayes classification. Em *Proc. 4th Int. Conf. Comput. Informatics, ICOCI*, número 49.

Yuxin, D., Xuebing, Y., Di, Z., Li, D. e Zhanchao, A. (2011). Feature representation and selection in malicious code detection methods based on static system calls. *Computers & Security*.

Zamani, M. e Movahedi, M. (2013). Machine learning techniques for intrusion detection. *arXiv preprint arXiv:1312.2177*.

Zhang, M., Xu, B. e Gong, J. (2015). An anomaly detection model based on one-class svm to detect network intrusions.