Detecção de Anomalias: Estudo de Técnicas de Identificação de Ataques em um Ambiente de Contêiner

Gabriel Ruschel Castanhel[1], Tiago Heinrich[1], Fabrício Ceschin[1], Carlos A. Maziero[1]

[1]Departamento de Informática
Universidade Federal do Paraná (UFPR)

{grc15,theinrich,fjoceschin,maziero}@inf.ufpr.br

Abstract. *Application deployment in an isolated environment with practical administration is what the container technology offers. Due to its popularity, security-related questioning about the tool and the relationship between host and container are presented. Anomaly detection offers guidance for the system's behaviour monitoring, making possible the detection of behaviours that differ from what is assumed normal for that application. This paper aims to analyse and compare different methods used for real-time anomaly detection in a containerized application approach, in order to indentify the impact of detection in an isolated environment for a non-isolated host. Machine learning methods were used as detectors to classify if a behavior from a given window is a threat.*

Resumo. *A execução de aplicações em um ambiente isolado e com manuseio prático é a tecnologia oferecida pelos contêineres. Devido à sua popularidade, questionamentos sobre a segurança da ferramenta e proximidade do contêiner com o host são elencados. A detecção de intrusão por anomalias oferece uma via para o monitoramento do comportamento do sistema, possibilitando a detecção de comportamentos que diferem do que é considerado normal para a aplicação. Este artigo visa analisar e comparar métodos utilizados para a detecção de anomalias em tempo real com uma abordagem voltada a aplicações executando em um ambiente de um contêiner, cujo intuito é identificar o impacto na detecção em um ambiente isolado para um host sem isolamento. Métodos de aprendizado de máquina foram utilizados como detectores para classificar se um comportamento em determinada janela é uma ameaça.*

1. Introdução

A virtualização em nível de sistema operacional conhecida como contêiner, é um recurso possível devido ao nível de isolamento oferecido pelo kernel, ao qual isola múltiplos espaços de usuários. Responsável por isolar o código de uma aplicação e suas dependências, onde os recursos necessários para a execução da aplicação são controlados pelo próprio contêiner. A praticidade dos recursos consiste em não depender do sistema base ao qual o recurso está executando, permitindo múltiplos contêineres com diferentes estruturas e aplicações executarem um ao lado do outro e ainda compartilhar o mesmo kernel e sistema operacional.

A detecção de intrusão elabora meios para identificar e prevenir atividades maliciosas no ambiente. Recursos utilizados para estas atividades são *Intrusion Detection*

System (IDS) e *Intrusion Prevention System (IPS)* [Lam 2005]. Especificamente um IDS realiza a identificação de intrusão através de diferentes técnicas, estas sendo com base em assinatura, que realiza a comparação de assinaturas com uma base já conhecida de ameaças ou com base em anomalia, que define um comportamento normal para o sistema e busca identificar anomalias [Yassin et al. 2013].

Uma avaliação que explora técnicas por anomalia, possibilita a identificação de ameaças não classificadas ou até mesmo desconhecidas já que este recurso visa definir um comportamento "normal" do sistema ou aplicação [Taha and Hadi 2019]. A *system call* é responsável por converter chamadas/requisições realizadas pelas aplicações para uma linguagem compreensível pelo kernel, a observação deste recurso permite a definição de um comportamento "normal" do sistema.

Visando a detecção de intrusão em um ambiente de virtualização, o respectivo trabalho aplica um conjunto de métodos para comparar a eficácia na identificação de atividades maliciosas em um contêiner. A técnica aplicada consiste em treinar um classificador utilizando *system calls* para definir um comportamento "normal" em cada contêiner por consequência realizando a identificação de anomalias.

Este trabalho está dividido em seis Seções. A Seção 2 apresenta os conceitos necessários para o entendimento da pesquisa. A Seção 3 apresenta os trabalhos relacionados. A Seção 4 apresenta a proposta do estudo. A Seção 5 discute os resultados e a Seção 6 conclui o estudo.

2. Fundamentação Teórica

Esta Seção apresenta os principais conceitos para a compreensão do trabalho. Discutindo pontos como detecção de anomalias, *system calls* para IDS e virtualização por contêiner.

2.1. Detecção de Anomalias

A crescente preocupação em respeito à segurança de computadores, suas comunicações e demais aspectos envolvidos, promovem estudos e pesquisas voltadas à prevenção e identificação de ameaças que acontecem em tempo real ou que podem vir a acontecer eventualmente [Deshpande et al. 2018]. Inicialmente, as técnicas são baseadas na análise de *logs* gerados por aplicações ou pelo próprio sistema operacional, com o objetivo de identificar comportamentos anormais ou incomuns para o sistema. Por sua vez, é a 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.

Os Sistemas de Detecção de Intrusão (*Intrusion Detection System (IDS)*) ganharam notoriedade neste contexto nas últimas décadas, são sistemas designados para identificar ameaças em tempo real ou que podem ter acontecido. Subdividido em duas classes em relação a sua área de atuação: para um *host* específico (*Host-based Intrusion Detection System (HIDS)*) ou para uma rede (*Network Intrusion Detection System (NIDS)*) [Kumar 2007, Brown et al. 2002]. O NIDS realiza o monitoramento do tráfego da rede enquanto o HIDS é voltado a monitorar o *host* visando identificar ameaças em arquivos e recursos do sistema operacional (*logs*, memória, *system calls*, etc).

Um IDS pode ter base por assinatura, onde o sistema realiza a identificação de acordo com uma base de padrões de ameaça, este conjunto requer que a ameaça

seja conhecida já que é realizada uma comparação dentre as "assinaturas" conhecidas [Debar et al. 1999, Yassin et al. 2013]. A detecção por anomalia realiza o monitoramento de recursos do sistema em busca de padrões que desviam do comportamento normal que possam indicar possíveis ameaças.

A detecção de anomalias é uma técnica aplicada em diversos contextos não só limitado à segurança de sistemas, como estatística, finanças e análise de fraude [Chandola et al. 2009]. Ao trabalhar com a identificação de anomalias três categorias de classificação são possíveis [Chandola et al. 2009]. A detecção não supervisionada detecta anomalias com dados não categorizados, assumindo que a proporção das instâncias "normais" seja predominante na base de amostras. A detecção supervisionada atua com dados categorizados em que o *dataset* apresenta um *label* para cada amostra, como "normal" e "anormal", utilizando um classificador multiclasse. A detecção semi-supervisionada é responsável por identificar desvios ou "anomalias" em relação a um *dataset* já rotulado utilizado anteriormente.

2.1.1. System Calls

Uma *system call* é o recurso disponível para a interação entre uma aplicação (ou processo) e o kernel do sistema operacional. Quando um programa precisa realizar operações do kernel, as requisições são feitas através das *system calls* disponíveis, geralmente porque processos comuns executando no sistema não podem (ou não tem permissão) realizar tais operações, e isso ocorre para uma variedade de tarefas como: interagir com recursos do hardware, gerenciamento de memória, processamento de entrada/saída, uso da interface de rede, operações com o sistema de arquivos e dentre outras. Todas as *system calls* representam uma interface essencial entre sistema operacional e processos, onde cada chamada representa uma operação básica ou capacidade [Mitchell et al. 2001]. O conjunto de *system calls* encontradas em um sistema está diretamente relacionado com o *Sistema Operacional (SO)* e a arquitetura utilizada. Ferramentas como *strace* e *ftrace* [Cespedes and Machata 2013, ftrace 2018], permitem mostrar a sequência de todas as *system calls* utilizadas por um comando ou um processo em execução.

Devido a posição em que as *system calls* são encontradas, este recurso acaba sendo poderoso e tem direta influência no sistema. Suas funcionalidades permitem o desligamento do sistema, alocação de recursos, restrição de acesso como a redução de permissões para um usuário. Desta forma, a arquitetura dos sistemas modernos acaba implementando camadas de segurança, que especifica alguns níveis de segurança que determina quais tipos de programas podem interagir com determinados recursos. Esse 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 das mais 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.

Entretanto, independente das abordagens utilizadas para executar código malici-

oso em um sistema, ataques compartilham uma característica [Jain and Sekar 2000]: "eles normalmente exploram a interface de *system calls* para operações maliciosas". É apenas através desta interface que uma aplicação comprometida pode escrever em disco, enviar dados pela rede ou interagir com o sistema. O monitoramento de *system calls* é uma técnica amplamente utilizada que faz o uso dessa característica em comum para detectar um comportamento suspeito de uma aplicação que pode ter sido comprometida, para que seja possível uma contramedida para minimizar o problema [Rajagopalan et al. 2006].

2.2. Virtualização Baseada em Contêiner

Embora o conceito de contêiner foi 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. Atualmente é conhecida como uma técnica de virtualização de aplicações para os mais diversos propósitos, geralmente associada às suas ferramentas populares como o *docker*. O contêiner é um ambiente virtual que executa em um único SO e permite o carregamento e execução de uma aplicação específica e suas dependências contidas em uma virtualização de um sistema operacional [Merkel 2014]. Desta forma, o contêiner reúne os componentes necessários para a execução da aplicação, que inclui código, bibliotecas e variáveis de ambiente enquanto o SO do *host* gerencia o acesso aos recursos de *hardware* como consumo de memória e processamento, além de todas as operações necessárias do kernel.

A técnica promove um isolamento porém não é um isolamento total do SO do *host* como uma máquina virtual, onde existe o *hypervisor* que é responsável por gerenciar e executar os sistemas convidados. O *hypervisor* é a camada intermediária entre as máquinas virtuais e o *host* (pode estar diretamente no *hardware* ou SO), responsável pela conversão de instrução e solicitação de recursos [Litty 2005]. Este provê um isolamento total mas requer que os recursos sejam alocados para cada virtualização sem a possibilidade de compartilhamento, que ocorre ao utilizar contêineres [Durairaju 2018].

Pelo fato de contêineres compartilharem o kernel do SO, uma única instância é capaz de suportar múltiplos contêineres isolados, e além disso, o SO virtualizado no contêiner é mais leve em relação a uma máquina virtual que requer todos os recursos de um sistema completo, com apenas os recursos necessários para a execução da aplicação [Sharma et al. 2016]. Essa característica acaba criando certas desvantagens, pois contêineres compartilham o kernel do SO, recursos específicos de contêineres (incluindo contêineres e a ferramenta de gerenciamento de contêineres), o que implica que duas ou mais aplicações executando em um único contêiner podem interferir uma a outra, e *software* executando em um contêiner pode impactar um outro contêiner sob o mesmo *host*, além da possível interferência no próprio *host*.

3. Trabalhos Relacionados

Os recursos oferecidos pela virtualização oferecem *trade-offs* para o monitoramento dos recursos presentes no ambiente, como visibilidade e capacidade [Bridges et al. 2019]. Um IDS não deve estar presente dentro do *host* virtualizado, onde seria suscetível ao *hypervisor* e teria uma visão parcial do ambiente. Observando o lado de um atacante, possibilidades existem como explorar a virtualização para extrair informações privadas dos usuários, lançar ataques *Distributed Denial of Service (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. Alguns exemplos de abordagens seriam *Kernel-based Virtual Machine (KVM)*[Pfoh et al. 2011] ou *Bag of System Calls* [Alarifi and Wolthusen 2012]. Já para abordagens com base na virtualização de contêiner é possível destacar [Srinivasan et al. 2018], que apresenta um modelo para identificação de anomalias em aplicações executando dentro do Docker. A estratégia consistem 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 [Systems 1998], ao qual acaba não sendo representativo para eventos e aplicações atuais ou ocorrentes no ambiente virtualizado.

Ainda no contexto de contêineres, [Flora and 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 *Sequence Time-Delay Embedding (STIDE)* e *Bag of System Calls (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. 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 não é disponibilizada.

O estudo de [Abed et al. 2015], foca na detecção de intrusão por anomalias em ambientes de contêiner, aplicando 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 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.

4. Proposta

A literatura existente relacionada ao desenvolvimento de métodos para a detecção e análise de intrusão em ambientes conteinerizados possui limitações. Entretanto, existem fontes relacionadas a métodos tradicionais para a detecção de intrusão disponíveis, que podem ser utilizadas para comparação com novas técnicas utilizadas em contêineres. Tais adaptações de métodos existentes para este ambiente são uma tendência promissora para a segurança de contêineres [Hickman 2018]. Um exemplo de ferramenta desenvolvida para este 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 (ML)*.

Embora a tecnologia de contêineres seja uma técnica viável para executar uma

aplicação em um ambiente isolado, é sabido que não existe um isolamento completo do contêiner para o *host* [Xavier et al. 2015]. Sendo uma possível porta de entrada de ataques ao sistema, aplicações em execução, imagem com vulnerabilidades ou até uma imagem "maliciosa" [Combe et al. 2016, Kwon and Lee 2020] .

O monitoramento de vulnerabilidades em nível de aplicação, é uma rotina com uma carga de trabalho alta, sendo inviável realizar um monitoramento de dentro do contêiner por conta de suas restrições e recursos limitados [Abed et al. 2015]. Portanto, uma opção seria monitorar os processos no sistema *host* que representam a execução do contêiner em questão, utilizando como dados algum recurso que seja compartilhado com o *host*, como neste caso, as *system calls* geradas pelos processos que executam o contêiner.

Apesar da dificuldade em filtrar *system calls* geradas pelo ambiente já que um trace irá apresentar um conjunto misto de chamadas da aplicação e de controle do contêiner. O *host* e o contêiner compartilham o mesmo kernel, por consequência as *system calls* são uma boa opção para detecção de anomalias, pois representam todas as instruções que são executadas dentro do contêiner. As *system calls* permitem uma maneira de observar e analisar o que é executado pela aplicação isolada dentro do contêiner a partir do próprio *host*, sem precisar interferir com modificações dentro deste ambiente isolado.

Nesse contexto, [Forrest et al. 1996] introduz um método com base em janelas de sequências de *system calls*, que apresenta simplicidade e eficácia para detecção em tempo real. Abordagens utilizando tabelas de frequências [Abed et al. 2015, Alarifi and Wolthusen 2012], algoritmos de *Machine Learning (ML)* [Liao and Vemuri 2002, Yuxin et al. 2011] e cadeias de markov [Wang et al. 2004] também apresentaram bons resultados referentes a taxas 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 um ambiente de contêiner, realizando uma observação pelo próprio *host*. Por fim, realizamos uma avaliação da detecção de intrusão no ambiente.

4.1. Estratégia

Devido ao volume de dados gerado, a eliminação de "ruído" do conjunto de *system calls* obtidas foi considerado. Esta eliminação consiste em descartar *system calls* avaliadas como inofensivas a partir de um mapeamento de nível de ameaça, que distribui as *system calls* existentes em grupos com níveis de ameaça.

O *Wordpress* foi a aplicação escolhida para ser executada no ambiente. A escolha é em decorrência à sua popularidade em meio a aplicações web, e atenção devido à existência de um conjunto de vulnerabilidades. A possibilidade de constante uso de *plugins* e temas personalizados com falhas de segurança, permite que atacantes explorem vulnerabilidades como *Cross-site scripting (XSS)*, *SQL Injection*, *Remote Code Execution (RCE)*, entre outras.

Para o processo de avaliação cinco classificadores multiclasse e dois classificadores de uma classe foram definidos. Estes métodos de ML são *Random Forest*, *Naive Bayes*, *K-Nearest Neighbors (KNN)*, *Multilayer Perceptron (MLP)*, *Ada Boost*, *One-class SVM* e o *Isolation Forest*. Um *dataset* que reúne um conjunto de janelas de comportamento normal e anormal, para treinar e testar os classificadores também foi proposto.

Os experimentos realizados permitem avaliar a viabilidade da detecção de anomalia pela análise de *system calls* de uma aplicação em um ambiente de contêiner. Possibilitando realizar uma comparação de eficiência/precisão de detecção entre os diferentes métodos citados, com comparação de parâmetros e abordagens.

5. Experimentos

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, com uma imagem do *Wordpress* na versão 4.9.14. Para realizar a avaliação é necessário um *dataset* que represente o comportamento a ser estudado[1]. Desta forma, foi elaborado uma base de comportamento normal e anormal, que seja representativo ao ambiente de virtualização ao qual o estudo tem foco. Posteriormente foi aplicada a técnica de janela deslizante para gerar os modelos que são utilizados para o treinamento do classificador.

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. As próximas seções discutem o processo de coleta, os experimentos realizados e resultados obtidos.

5.1. Coleta de Dados

A coleta de *system calls* geradas pelo contêiner foi feita pelo *strace*, uma ferramenta para sistemas Linux utilizada para monitorar as interações entre processos e o kernel que inclui o fluxo 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 pelo contêiner executando o *Wordpress*, e com um parâmetro que redireciona todos os resultados para um arquivo específico, onde cada coleta é direcionada a um arquivo diferente.

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

Foram feitas modificações para preservar a sequência de *system calls* do processo antes da coleta, sendo definido no servidor *Apache* responsável pelo *Wordpress* um número mínimo de processos filhos e *threads*. Estas medidas garantem a captura das *system calls* sem alterações e com todos os parâmetros e valores de retorno. Com o método de coleta dos dados definido, para a detecção de anomalia, é preciso construir uma base com comportamentos normais, com sequências de *system calls* que representam o funcionamento normal da aplicação, e com sequências que representam a mesma sob um ataque ou comportamento mal intencionado. Após a coleta inicial, as bases são "reconstruídas" porém desta vez formadas por janelas, de tamanho variando entre 3, 5, 7, 9, 11, 13 e 15, deslizadas pelos arquivos contendo as *system calls*, técnica utilizada em métodos baseados em sequência como o primeiro caso de estudo utilizando janelas deslizantes de [Forrest et al. 1996]. As janelas podem conter o nome das *system calls* ou um número inteiro único, atribuído a cada *system call* a partir de uma tabela. A base

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

formada contém 367.342 *system calls* representando a aplicação *Wordpress* em seu funcionamento normal e 1.628 *system calls* representando a aplicação sob um comportamento suspeito/malicioso. A base só retem as *system calls* da execução dos processos.

A simulação de comportamento normal do *Wordpress*, contou com um conjunto de *hosts* dentro da própria rede. Foi optado por variar os sistemas operacionais que seriam responsáveis por interagir com o *Wordpress*, com *hosts* Linux e Windows sendo utilizados. Este garante uma variação nas rotinas e interações normais para a aplicação, como criação de novos *posts* e comentários, e manutenções corriqueiras no blog de teste implantado como alvo da pesquisa.

Para a obtenção dos dados de comportamento anormal, foi feita a avaliação de um *host* específico, que explorou uma vulnerabilidade RCE [NVD 2020], onde as bases de teste são formadas a partir de execuções similares.

Ao explorar a literatura por *datasets* públicos que representem o ambiente de virtualização voltado para a intrusão em contêiner é possível identificar uma lacuna, além de limitações em relação a evolução do conjunto de dados encontrados nas poucas bases disponíveis, as quais falham em ser representativas. Assim, além de desenvolver um estudo comparativo um foco da pesquisa é o desenvolvimento de uma base que consiga representar esta realidade.

5.2. Avaliação com Conjunto Total de Chamadas

O primeiro conjunto de experimentos consiste da utilização das *system calls* sem nenhum tratamento com o foco em auxiliar os classificadores. *System calls* foram representadas por um identificador numérico, com o auxílio de uma tabela associando cada *system call* a um *id* único.

O *dataset* contém a junção das duas bases porém possui um *label* categorizando cada janela em "normal" ou "anormal". Para a aplicação nos métodos de avaliação o *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 podem ser encontrados na Tabela 1. O *Random Forest*, MLP e o KNN são os algoritmos com a melhor *precision* 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 um valor razoavelmente alto 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, primeiramente tentamos 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 and Schapire 1997], no nosso caso, nos exemplares de ataques. Para isso, utilizamos o *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, 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.

Observando somente a variação crescente no tamanho de janela, não é possível identificar uma preferência de melhor caso ou recomendação de um tamanho ideal para um conjunto de *system calls* sem nenhum tipo de tratamento.

Tabela 1. Desempenho dos algoritmos de ML considerando todas as chamadas.

Tipo	**Classificador**	**Metrica**	**Janela**						
			3	**5**	**7**	**9**	**11**	**13**	**15**
Multiclasse	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

5.3. Avaliação com Seleção de Chamadas

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 *Denial of Service (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 2, que representa parte da estrutura definida por [Bernaschi et al. 2002].

Tabela 2. *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

Esta classificação de segurança inofensiva é atribuída, porque a maioria das *system calls* listadas neste nível tem como funcionalidade o retorno de algum valor ou atributo sobre um arquivo específico, como no caso das chamada *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 no sistema, portanto podem ser classificadas como inofensivas em relação a segurança do sistema.

A partir das *system calls* apresentadas na Tabela 2, 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 das bases de teste. 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 3 apresenta os resultados para o conjunto de classificadores avaliados com os dados filtrados. 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.

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 identifi-

Tabela 3. Desempenho dos algoritmos de ML com filtragem de chamadas.

Tipo	Classificador	Metrica	Janela						
			3	5	7	9	11	13	15
Multiclasse	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

que 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 testamos 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 multiclasse, resultando em um baixo falso negativo, isto é, os classificadores de uma classe conseguem detectar anomalias muito melhor que os classificadores multiclasse em geral, provavelmente devido ao grande desbalanceamento de classes do *dataset* (cenário que favorece esse tipo de classificador).

6. Conclusão

Neste artigo, foi estudado a possibilidade e viabilidade da detecção de intrusão por anomalias em um ambiente de contêiner, com a possibilidade de analisar a diferença de resultados dos métodos quanto aos tamanhos de janela utilizadas nas sequências e também do tipo de *system calls* levadas em consideração para treinar e testar classificadores multiclasse e de uma classe. Embora não tenha sido possível obter resultados relevantes utilizando classificadores multiclasse, 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. Como este experimento foi inspirado na classificação feita

por [Bernaschi et al. 2002], fica também para trabalhos futuros uma nova avaliação de ameaça de *system calls*, já que o trabalho não cobre todas as chamadas presentes em sistemas modernos. Além disso, classificadores de uma classe obtiveram resultados superiores em relação aos métodos tradicionais de ML, mesmo não realizando a filtragem de *system calls* apresentada.

Outro ponto a se discutir foi o *dataset* levantado para este experimento, levando em consideração os trabalhos relacionados, foi constatado que é possível a coleta de dados a partir de uma aplicação isolada em um contêiner, e também necessário, já que não há exemplos de conjuntos de dados formados neste ambiente disponíveis publicamente como foi discutido anteriormente, considerando a crescente demanda de análises de segurança de ambientes conteinerizados, devido à crescente popularidade desta tecnologia. É necessário discutir que o *dataset* utilizado nos experimentos foi uma versão inicial e limitada, o que consequentemente influenciou nos resultados finais. Para trabalhos futuros o objetivo é produzir uma base mais representativa, tanto de comportamento normal quanto de comportamento anormal, explorando diferentes casos de uso e tentativas de ameaça para gerar um conjunto de dados mais robusto.

Vale lembrar que nosso trabalho trata da detecção de anomalia em tempo real (detecção *online*), isto é, fazemos a análise em cima das janelas de *system calls* disponibilizadas, e não do conjunto das mesmas disponibilizadas após a execução completa (detecção *offline*). Desta forma, o problema torna-se muito mais desafiador, uma vez que estamos observando apenas uma parte do traço, já que não queremos que uma ação maliciosa termine sua execução para tomarmos uma decisão. Logo, consideramos que o problema de detecção em tempo real é uma área em aberta na literatura hoje e deve ser melhor investigada por novas pesquisas.

Referências

[Abed et al. 2015] Abed, A. S., Clancy, C., and Levy, D. S. (2015). Intrusion detection system for applications using linux containers. In *International Workshop on Security and Trust Management*. Springer.

[Abed et al. 2015] Abed, A. S., Clancy, T. C., and Levy, D. S. (2015). Applying bag of system calls for anomalous behavior detection of applications in linux containers. In *2015 IEEE Globecom Workshops (GC Wkshps)*.

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

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

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

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

[Cespedes and Machata 2013] Cespedes, J. and Machata, P. (2013). ltrace(1), linux manual page. https://man7.org/linux/man-pages/man1/ltrace.1.html.

[Chandola et al. 2009] Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. *ACM computing surveys (CSUR)*.

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

[Debar et al. 1999] Debar, H., Dacier, M., and Wespi, A. (1999). Towards a taxonomy of intrusion-detection systems. *Computer Networks*.

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

[Durairaju 2018] Durairaju, S. S. (2018). Intrusion detection in containerized environments.

[Flora and Antunes 2019] Flora, J. and Antunes, N. (2019). Studying the applicability of intrusion detection to multi-tenant container environments. In *2019 15th European Dependable Computing Conference (EDCC)*.

[Forrest et al. 1996] Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A. (1996). A sense of self for unix processes. In *IEEE Symposium on Security and Privacy*.

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

[ftrace 2018] ftrace (2018). perf-ftrace(1) — linux manual page. https://man7.org/linux/man-pages/man1/perf-ftrace.1.html.

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

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

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

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

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

[Lam 2005] Lam, A. (2005). New ips to boost security, reliability and performance of the campus network. *Newsletter of Computing Services Center*.

[Liao and Vemuri 2002] Liao, Y. and Vemuri, V. R. (2002). Using text categorization techniques for intrusion detection. In *USENIX Security Symposium*.

[Litty 2005] Litty, L. (2005). *Hypervisor-based intrusion detection*. University of Toronto.

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

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

[Mitchell et al. 2001] Mitchell, M., Oldham, J., and Samuel, A. (2001). *Advanced linux programming*. New Riders Publishing.

[NVD 2020] NVD (2020). National vulnerability database: Rce wordpress. `https://nvd.nist.gov/vuln/search/results?form_type=Basic&results_type=overview&query=RCE+wordpress&search_type=all`.

[Pfoh et al. 2011] Pfoh, J., Schneider, C., and Eckert, C. (2011). Nitro: Hardware-based system call tracing for virtual machines. In *Int. Workshop on Security*. Springer.

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

[Sharma et al. 2016] Sharma, P., Chaufournier, L., Shenoy, P., and Tay, Y. (2016). Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*.

[Srinivasan et al. 2018] Srinivasan, S., Kumar, A., Mahajan, M., Sitaram, D., and Gupta, S. (2018). Probabilistic real-time intrusion detection system for docker containers. In *International Symposium on Security in Computing and Communication*. Springer.

[Systems 1998] Systems, C. I. (1998). Sequence-based intrusion detection. `http://www.cs.unm.edu/~immsec/systemcalls.htm`.

[Taha and Hadi 2019] Taha, A. and Hadi, A. S. (2019). Anomaly detection methods for categorical data: A review. *ACM Computing Surveys (CSUR)*.

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

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

[Xavier et al. 2015] Xavier, M. G., De Oliveira, I. C., Rossi, F. D., Dos Passos, R. D., Matteussi, K. J., and De Rose, C. A. (2015). A performance isolation analysis of disk-intensive workloads on container-based clouds. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE.

[Yassin et al. 2013] 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. In *Proc. 4th Int. Conf. Comput. Informatics, ICOCI*, number 49.

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

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