Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Orquestração de Contêineres na Nuvem: Um Modelo de Segurança

Gabriel Pereira Fernandez

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Sistemas Distribuídos e Computação em Nuvem

Andrey Elísio Monteiro Brito
(Orientador)

Campina Grande, Paraíba, Brasil
©Gabriel Pereira Fernandez, 06 de Fevereiro de 2018

DIGITALIZAÇÃO:

SISTEMOTECA - UFCG

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

F363o Fernandez, Gabriel Pereira.

Orquestração de cointêineres na nuvem: um modelo de segurança / Gabriel Pereira Fernandez. ï Campina Grande, 2018.

92 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2018.

"Orientação: Prof. Andrey Elísio Monteiro Brito".

Referências.

1. Segurança da Informação. 2. Nuvem de Computadores. 3. Sistemas Distribuídos. 4. Contêineres. 5. Virtualização. I. Brito, Andrey Elísio Monteiro. II. Universidade Federal de Campina Grande, Campina Grande (PB). III. Título.

CDU 004.056.53(043)

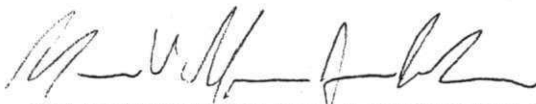
"ORQUESTRAÇÃO DE CONTÊINERES NA NUVEM - UM MODELO DE SEGURANÇA"

GABRIEL PEREIRA FERNANDEZ

DISSERTAÇÃO APROVADA EM 06/02/2017

ANDREY ELÍSIO MONTEIRO BRITO, Dr., UFCG
Orientador(a)

REINALDO CÉZAR DE MORAIS GOMES, Dr., UFCG
Examinador(a)



MARCUS WILLIAMS AQUINO DE CARVALHO, Dr., UFPB
Examinador(a)

CAMPINA GRANDE - PB

Resumo

A computação em nuvem tem se tornado o paradigma estabelecido de organização de infraestrutura de TI. Este fenômeno ocorreu por causa das vantagens de escala e especialização de serviço possíveis neste modelo, que permite que computação seja entregue aos usuários como um serviço e não mais como um produto ou processo. No entanto, este modelo requer a mudança da propriedade dos artefatos que compõem a infraestrutura de TI, além de seus direitos e permissões administrativos. Esta característica causa mudanças profundas no modelo de ameaça e cria relações de confiança frágeis que intimidam a migração de utilizadores de posse de dados sensíveis para esta nova forma de organização.

Este trabalho propõe um modelo de segurança que se adequa à nova realidade de organização de TI interpretando o papel desempenhado pelo serviço de provisionamento de nuvem como um que requer ceticismo por parte do usuário quanto à sua idoneidade. Para implementar um modelo de segurança que não confia os dados do usuário final das aplicações ao provedor de nuvem, dados ao provedor de nuvem, é empregada a utilização de tecnologias contemporâneas de execução código em ambiente confiável associada a ferramentas de gerenciamento e implantação de aplicações em infraestruturas de nuvem. Esta combinação constitui uma plataforma de orquestração de contêineres seguros com uma abordagem adequada às necessidades de disponibilidade modernas dos serviços de nuvem. Além disso, este trabalho demonstra a viabilidade desta proposta e fornece uma metodologia de avaliação de custos de utilização da plataforma para auxiliar a tomada de decisão de negócio.

Abstract

Cloud computing has become the IT infrastructure organization established paradigm.

This scenario has turned into reality because of the scale and service specialization advantages possible in a model that allows for computation to be delivered as a service instead of as a product or process. Although, this model requires changing the IT infrastructure artifacts ownership and its administrative permissions and governing rights. Such a characteristic causes deep changes in the threat model and creates fragile trust relationships that often intimidate the immigration of sensitive data holding utilizers to the new form of organization.

This work proposes a new security model that fits the new IT organization reality by interpreting the role played by the cloud provisioning service as one that requires skepticism from the user regarding its *bona fide*.

To implement a security model that does not trust the users data to the the cloud provider, we employ the use of current trusted environment code execution technologies, associated to management and deployment tools for cloud infrastructures. This combination constitutes a secure container platform that is adequate to modern cloud services needs for availability.

We demonstrate as well the feasibility of this proposal and offer a methodology to evaluate the platform usage costs to support business decisions.

Agradecimentos

O primeiro de meus agradecimentos se destina à minha mãe e meu pai, que nunca pouparam esforços, de corpo e de espírito, para de qualquer forma possível apoiar meu desenvolvimento intelectual, fruto do qual é este trabalho.

Agradeço à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo financiamento desta pesquisa. Agradeço também a meu orientador Andrey Brito, principalmente pelo papel de guia e mentor que desempenhou ao longo desta pesquisa, mas também pelos de defensor, amigo, conselheiro e, em discussões produtivas, adversário.

A Rodolfo Marinho, Lília Sampaio, Amanda Souza, Matteus Silva, Walter Alves, Ionésio Júnior e Dalton Cézane pelas ideias, sugestões, debates e insights que contribuíram tão decisivamente para o sucesso deste trabalho.

Em geral, a agradeço a todos os integrantes do Laboratório de Sistemas Distribuídos por compor um ambiente estimulante ao desenvolvimento de software relevante ao fazer humano moderno.

Conteúdo

1 Introdução **1**

1.1 Nuvem, Virtualização e Contêineres 3

1.2 Problema Abordado 8

1.3 Contribuições e Relevância 9

1.4 Objetivo 11

1.5 Metodologia 11

1.6 Organização da Dissertação 12

2 Tecnologias e Trabalhos Relacionados **14**

2.1 SGX 14

2.2 Trabalhos Relacionados 15

2.2.1 Contêineres seguros e outros ambientes 16

2.2.2 Abordagens de software que protegem aplicativos de código privilegiado 17

2.2.3 *Trusted Hardware Support* 18

3 Orquestração de Contêineres na Nuvem: Um Modelo Seguro **20**

3.1 Modelo de Atacante 20

3.1.1 Adversário Local 21

3.1.2 Adversário de Gerenciamento 21

3.2 Modelo de Segurança e Metodologia 22

3.2.1 *Secure Container Orchestrator* 26

3.2.2 Requerimentos para Orquestração Segura de Contêineres 27

3.3 Arquitetura, Fluxo de Implantação e Execução de aplicações do SCO . . . 29

3.3.1 Arquitetura 29
3.3.2 Fluxo de Implantação 30
3.3.3 Fluxo de Utilização 31
3.4 SCO e Outras Plataformas de Orquestração de Contêiner 32

4 Avaliação de desempenho do SCO 36
4.1 Aplicações SGX no SCO 37
4.1.1 SPPDA: Motivação 37
4.1.2 SPPDA: Arquitetura 38
4.2 Modelagem Experimental 39
4.2.1 Avaliação de Desempenho de *auto scaling* 40
4.2.2 Avaliação de Desempenho de Instanciação de Serviços 41
4.2.3 Avaliação de Desempenho da Aplicação 42
4.2.4 Definição das Variáveis Objetivo 42
4.3 Aplicação de Controle 43
4.3.1 Ambiente 44
4.4 Resultados e Análise 46
4.4.1 Desempenho de *auto scaling* 46
4.4.2 Desempenho de Instanciação de Serviços 48
4.4.3 Avaliação de Desempenho da Aplicação 49

5 Conclusões e Trabalhos Futuros 53
5.1 Trabalhos Futuros 54

A Visualizações dos Resultados dos Experimentos 61

B *Daemon* de Monitoramento 71

C API Docker e exemplos de chamadas 76
C.1 Docker API - Comandos Docker básicos e utilitários 76
C.2 Exemplos de Chamadas 83

D *Daemon* do balanceamento de carga 85

E Imagem base padrão do SCO **91**

Lista de Símbolos

C_{ai} - *Custo relacionado à execução de uma aplicação i*

C_e - *Custo relacionado a sobrecargas de engenharia*

C_s - *Custo relacionado às operações de segurança*

μ_i - *Média de latência para um conjunto de medições i*

α_1 - *Relação de custo entre SCO e Kubernetes para auto scaling*

α_2 - *Relação de custo entre SCO e Kubernetes para instanciação de serviços*

α_3 - *Relação de custo entre as aplicações de teste e controle*

T_0 - *Momento inicial de uma medição*

T_f - *Momento final de uma medição*

Δ_l - *Intervalo da prospecção de dados e pelo módulo de monitoramento e comunicação com o módulo de gerenciamento*

Δ_i - *Intervalo de instanciação de contêiner*

T_{ci} - *Momento do término da instanciação de contêiner i*

Δ_{ci} - *Intervalo entre o início e o término do tempo de instanciação de um contêiner i*

Δ_r - *Intervalo entre o envio e achegada da requisição de criação de um serviço*

T - *Intervalo de tempo da execução de um experimento*

T_{teste} - *Intervalo de tempo da execução de um experimento*

$T_{controle}$ - *Intervalo de tempo da execução de um experimento*

w_1 - *Peso associado à latência de auto scaling na avaliação de custo total*

w_2 - *Peso associado à latência de instanciação de serviço na avaliação de custo total*

w_3 - *Peso associado à latência da aplicação na avaliação de custo total*

C_{total} - *Custo total relativo entre o a utilização do SCO e Kubernetes*

Lista de Figuras

1.1 Arquitetura de um orquestrador de contêineres 7

3.1 Esquema de adulteração de uma aplicação com SGX 23

3.2 Arquitetura do SCO e Fluxo de utilização simplificado 32

3.3 Balanceador de Carga SCO 33

3.4 Esquema de implantação e utlização de aplicações com a atuação da TPAE e cliente SCO 33

4.1 Fluxo de troca de chaves da aplicação de controle 45

4.2 Latência de *auto scaling* com Intervalos de confiança - SCO x Kubernetes . 47

4.3 Latência de instanciação - SCO x Kubernetes 48

A.1 Latência de *auto scaling* com 2 instâncias - SCO x Kubernetes 61

A.2 Latência de *auto scaling* com 4 instâncias - SCO x Kubernetes 62

A.3 Latência de *auto scaling* com 6 instâncias - SCO x Kubernetes 63

A.4 Latência de *auto scaling* com 8 instâncias - SCO x Kubernetes 64

A.5 Latência de Instanciação com 2 instâncias - SCO x Kubernetes 65

A.6 Latência de Instanciação com 4 instâncias - SCO x Kubernetes 66

A.7 Latência de Instanciação com 6 instâncias - SCO x Kubernetes 67

A.8 Latência de Instanciação com 8 instâncias - SCO x Kubernetes 68

A.9 Latência de atestação das aplicações de teste e controle 69

A.10 Latência de atestação da aplicação de teste 70

UFCG/BIBLIOTECA/BC

Lista de Tabelas

1.1 Modelo de Propriedade em Serviços Web -Era Pré Nuvem 2

1.2 Modelo de Propriedade em Serviços Web -Era Nuvem 3

3.1 Comparativo de funcionalidades de soluções de orquestração de contêineres 35

4.1 Médias de Latência de *auto scaling* (ms) 46

4.2 Médias de Latência de Instanciação de Serviços (ms) 49

4.3 Estatísticas de Atestação das aplicações 50

Lista de Códigos Fonte

3.1 Introduzindo *Thread* maliciosa na parte desprotegida da aplicação 23

3.2 Acessando dados sensíveis do usuário de dentro do enclave 24

Capítulo 1

Introdução

Ao longo da década corrente a expansão no volume de acesso dos serviços *web* tornou a infraestrutura de TI existente obsoleta. A eficácia em dimensionar a capacidade de atendimento à demanda requerida passou a ser um fator competitivo crucial para a grande quantidade de serviços surgindo [8]. A utilização de infraestruturas proprietárias dos provedores desses serviços acarretava altos custos de manutenção com baixos níveis de utilização e eram insuficientes para manter a vitalidade das aplicações em picos de acesso [4; 10].

Neste contexto, operadores capazes de oferecer infraestrutura de TI em grande escala se tornaram uma alternativa atrativa para a maioria dos provedores de serviços *web*. Preços competitivos e baixas taxas de indisponibilidade garantidas por SLA's (*Service Level Agreement*), possíveis por causa da grande escala e controle de granularidade, permitiram que aplicações com capacidade de atendimento realmente elástica pudessem existir. Tais operadores, do que ficou conhecido como "a nuvem", hoje hospedam a maioria das aplicações e dados disponíveis na *web* [4].

O crescimento da *web*, e posteriormente da nuvem, e sua afirmação como principal paradigma de comunicação ocasionou também um aumento exponencial na quantidade de dados presentes nos nós da rede, entre os quais aqueles que constituídos de informações sensíveis e de alto valor [10]. Vazamentos desses dados provocariam perdas financeiras e de outras naturezas para seus proprietários e consequentemente para os provedores dessas aplicações.

Soluções de software consagradas para a proteção de dados nos níveis de rede, processamento, armazenamento e memória [11; 14; 21; 24] foram empregadas, e evoluíram para

Tabela 1.1: Modelo de Propriedade em Serviços Web -Era Pré Nuvem

	Usuário	Proprietário de Serviço
Possui Dados do Usuário	✓	✓
Possui Aplicação	✗	✓
Possui Infraestrutura	✗	✓

atender ao ambiente amplo e heterogêneo da *web*. Apesar de ter gêneros e características diferentes, estas soluções de segurança compartilham o intuito de resguardar esses dados de adversários externos com intenções maliciosas.

O arquétipo de segurança de dados que se dedica à proteção de informações dos usuários dos serviços de ameaças exteriores tornou-se paradigmático por herança da era de infraestruturas privadas da *web*, que seguiam um modelo de propriedade antigo (Tabela 1.1). Na atualidade, a propriedade da infraestrutura necessária para executar as aplicações da *web* está, crescentemente, no controle de terceiros, os provedores de serviços de nuvem [31; 37] (Tabela 1.2). Comparativamente, um esforço significativamente menor foi aplicado na criação de ferramentas, protocolos e arcabouços empenhados na tarefa de impedir que os próprios provedores de nuvem ameaçassem a privacidade dos dados dos usuários das aplicações executando em sua infraestrutura.

A falta de respostas para as ameaças a dados mais sensíveis e valiosos em serviços *web* localizados em *data centers* na nuvem teve consequências no ritmo de adoção desta prática. Em Géczy, Izumi, Hasida [18], ficou claro que um grande número de potenciais clientes da nuvem pública, receosos quanto à segurança dos dados dos seus serviços, optou por uma alternativa de nuvem privada, em que princípios que aumentam a eficiência e elasticidade dos serviços são aplicados numa infraestrutura pertencente exclusivamente à organização. Outra alternativa comum é a nuvem híbrida, que é composta de elementos hospedados na nuvem pública mas que efetua operações e armazenamento de dados sensíveis em uma parte privada integrada a esses elementos.

Estas alternativas de modernização de infraestrutura de TI, apesar de aperfeiçoar os serviços [29], acarretam para os provedores de serviços *web* e outros sistemas de informação um aumento no Custo Total de Propriedade de suas aplicações. As vantagens competitivas oferecidas pela escala da nuvem pública, como a capacidade de elastecer uma parte do

UFCG/BIBLIOTECA/BC

Tabela 1.2: Modelo de Propriedade em Serviços Web -Era Nuvem

	Usuário	Proprietário de Serviço	Provedor de Serviço de Nuvem
Possui Dados do Usuário	✓	✗	✓
Possui Aplicação	✗	✓	✓
Possui Infraestrutura	✗	✗	✓

serviço indefinidamente, ficam indisponíveis para provedores operando dados sensíveis [8].

1.1 Nuvem, Virtualização e Contêineres

O conceito de nuvem, essencialmente, é uma categoria de terceirização de serviços. Na era dos serviços *web* em infraestruturas proprietárias, a atividade final de um proprietário de uma aplicação não era oferecer uma infraestrutura de TI, mas um produto ou serviço acessível através dessa aplicação. Provedores de Serviço de Nuvem (PSN) apareceram no final dos anos 2000 oferecendo infraestrutura de TI no formato de um serviço, prática que passou a ser conhecida como *Infrastructure as a Service* (IaaS). Os PSN possuem um alto nível de especialização e grande escala, o que diminui seus custos operacionais para hospedar cada aplicação. A nuvem oferece também a possibilidade de aumentar o poder de atendimento da aplicação com um aumento relativamente pequeno do custo total, devido ao decréscimo progressivo no custo por unidade computacional.

Um dos fatores que se revelou mais significativo para manter altas taxas de utilização dos recursos, uma métrica fundamental para manter a rentabilidade de um PSN, foi o multi-inquilinismo, isto é, a capacidade de manter aplicações de mais de um usuário em uma mesma máquina hospedeira. Esta funcionalidade, fundamental no competitivo mercado de IaaS, no entanto acarreta duros requisitos de segurança [3; 34]. É necessário manter um alto grau de isolamento entre as aplicações dos diferentes inquilinos. No modelo de segurança atualmente empregado, outros usuários são também percebidos como ameaças.

A solução encontrada para este problema foi a virtualização. Esta técnica consiste em criar espaços computacionais virtuais isolados, onde os recursos são disponibilizados criando a ilusão de que há um único usuário presente no ambiente. Variações desta técnica em vários níveis foram criadas e utilizadas nas décadas de 60, 70, 80 e 90. Durante o advento dos

serviços de Nuvem e PSN's, a Virtualização a Nível de Hipervisor, em unidades chamadas de Máquinas Virtuais, era a metodologia de virtualização preponderante. Esta técnica envolvia a utilização de uma camada de software, o Hipervisor, que era responsável por mediar os acessos dos sistemas operacionais, isolados entre si, ao hardware [26].

Por proporcionar pouca liberdade de ação para um inquilino malicioso com a intensão de tomar o controle de instâncias de outros inquilinos [3], permitir uma ampla variedade de diferentes aplicações serem executadas na mesma máquina hospedeira e oferecer facilidade de gerência de instâncias, a virtualização em Máquinas Virtuais obteve desde o início grande adoção pelos PSN's e rapidamente se tornou o paradigma vigente na implantação de aplicações na nuvem.

Apesar de oferecer várias vantagens, as Máquinas Virtuais apresentavam inconveniências que se agravavam à medida que crescia a adoção da nuvem. Com esta técnica, quando um serviço precisava de mais instâncias para atender um pico de acessos, novas Máquinas Virtuais têm que ser criadas e seus Sistemas Operacionais iniciados. Esse processo é lento e comumente não atende a variações rápidas na demanda, o que gera indisponibilidades no serviços. A literatura, em [25; 20], demonstra que do ponto de vista de negócio, o não atendimento de uma requisição tem um custo proibitivo. Uma abordagem a este problema passou a ser a iniciação preemptiva de instâncias, prevendo possíveis aumentos e quedas de demanda, que foi empregada com sucesso limitado, uma vez que os modelos tinham margens de acerto imprecisas e que pioravam à medida que a variação no tráfego se tornava volátil [30].

No início dos anos 2010, uma técnica de virtualização que atua no nível do Núcleo de Sistema Operacional passou a ganhar rapidamente popularidade na implantanção de aplicações em ambientes multi-inquilino. Esta técnica, a conteinerização, consiste em, iniciado um Sistema Operacional, comumente uma distribuição de Linux, *montar* um outro Espaço de Usuário sobre o mesmo Núcleo de Sistema Operacional. Estes Espaços de Usuário possuem um sistema de arquivos e seus binários, espaço de endereçamento de identificadores de processos e espaços de endereçamento de rede independentes uns dos outros [20]. Neste modelo, somente o Espaço de Usuário hospedeiro, que gerencia os demais, chamados de Contêineres, é capaz de acessar a todos os recursos. Os contêineres hospedados atuam como sistemas operacionais completamente independentes, mas compartilham o núcleo do sistema

operacional entre si e com o espaço de usuário do host. Este compartilhamento de infraestrutura de sistema operacional se dá de forma transparente para o usuário, de maneira que para um processo os ambientes de contêiner e de uma máquina não virtualizada são indistinguíveis.

Apesar do alto nível de transparência, a infraestrutura de software compartilhada entre contêineres é significativamente maior e mais complexa do que os hipervisores em máquinas virtuais. Hipervisores são relativamente monolíticos e apresentam uma superfície de ataque simples e de difícil exploração por um atacante. A virtualização por conteinerização, por outro lado, tem em todas as funcionalidades providas pelo núcleo do sistema operacional, como escalonamento de processos, acesso a periféricos de hardware e alocação de memória para processos, possíveis vetores de ataque para um adversário empenhado em tomar o controle da infraestrutura de TI.

Existe, portanto, uma troca de valor entre as metodologias de virtualização em máquinas virtuais e contêineres. Contêineres em geral são mais escaláveis e eficientes no consumo de recursos computacionais, além de serem mais adequados a arquiteturas desacopladas, como será tratado a seguir. Máquinas virtuais, por outro lado, oferecem menos oportunidades de ataque para um eventual inquilino mal intensionado.

Tornou-se crescentemente comum a utilização de contêineres executando sobre máquinas virtuais com múltiplos inquilinos. Essa alternativa, apesar de neturalizar o ganho de eficiência provido pelo contêineres, mantém suas demais propriedades e cria um ambiente mais seguro para a execução de aplicações dos clientes do PSN.

A gerência dos contêineres em uma máquina hospedeira se dá através de uma plataforma de conteinerização que executa como um processo no sistema operacional hospedeiro. Em particular, esta técnica, que surgiu como uma funcionalidade do Linux nos anos 90, ganhou especial protagonismo com o advento do Docker [1], uma plataforma de conteinerização de aplicações que oferece um conjunto de ferramentas para gerência de contêineres. Docker provê controle de ciclo de vida de contêineres, criação de imagens multicamada, arquitetura de serviços multicontêineres, um repositório público de imagens, monitoramento de recursos entre outras utilidades. A completude e facilidade de uso da plataforma impulsionou o rápido crescimento em sua popularidade na indústria e tornou acessíveis as vantagens de implantação de aplicações em contêineres.

A instanciação de um contêiner Linux em si, uma vez que sua imagem esteja construída, consome um tempo da ordem dos milissegundos, adicionando-se a execução de seu processo principal. Para máquinas virtuais, esta sobrecarga orbita a magnitude dos minutos [20; 16]. Esse alto poder de criação e remoção de instâncias rende a plataformas como o Docker a capacidade de escalar aplicações na nuvem de maneira muito mais responsiva a variações do nível de carga do que um sistema de máquinas virtuais.

De fato, o baixo custo de adicionar e remover contêineres a um serviço tem um impacto na forma como as aplicações são organizadas. Com baixa sobrecarga de instanciação, arquiteturas organizadas em microsserviços se popularizaram. Neste paradigma, cada componente de aplicação executa isoladamente em um conjunto de hospedeiros. Com contêineres em geral, mas especialmente com o uso do Docker, escalar os diferentes componentes da aplicação de um microsserviço independemente se torna simples e eficiente. A arquitetura de microsserviços tem as consequências de redução na taxa de indisponibilidade, por melhorar o tempo de recuperação a falhas, e de aumentar a utilização dos hospedeiros por adequar melhor a escala de cada componente à demanda [16].

Para criar microsserviços e coordenar sua implantação nos nós de um *cluster*, ferramentas que automatizam e manejam o controle de contêineres surgiram e se tornaram também rapidamente populares. Essas ferramentas são chamadas orquestradores de contêineres (figura 1.1).

Orquestradores de contêineres funcionam como uma interface simples de implantação de aplicações de múltiplos componentes independentemente alocados em infraestruturas comumente complexas. Estas soluções provêm a infraestrutura básica para a automatização da implantação de serviços de múltiplos módulos, podendo incluir sistemas de balanceamento de carga, redes internas, escala automática, monitoramento de recursos e suporte à federação de diferentes infraestruturas [32].

Outra tarefa importante dos orquestradores de contêineres é prover níveis de abstração para garantir a corretude de roteamento, e a gerência dos serviços, além de atender a requisitos básicos de segurança e controle de acesso a esses serviços. Essas funcionalidades tornam a criação, remoção, atualização e outras operações sobre contêineres de serviços em uma nuvem tarefas simples e automatizáveis.

Do ponto de vista da aplicação, a visão estabelecida pelo emprego de contêineres, a de

aplicações divididas em subserviços independentes e fracamente acoplados, que podem ser utilizados por diferentes componentes e escalam livremente, é factualmente implementada por um orquestrador de contêineres. O orquestrador de contêineres pode gerenciar a escala e ciclo de vida dos contêineres de acordo com as políticas estabelecidas pela aplicação, interligar a infraestrutura sobre a qual os serviços executam de maneira transparente, além de prover o isolamento de rede necessário para isolar componentes de uma mesma aplicação quando necessário e umas das outras, dado que trata-se de um ambiente multi-inquilino.

Com estas qualidades, a implantação de novas funcionalidades, a tolerância a falhas e a elasticidade das aplicações, características crescentemente desejáveis na filosofia moderna de atualização rápida de aplicações, conhecida como DevOps, obtêm uma melhora significativa.

Incorporados às plataformas de nuvem já existentes, os orquestradores de contêineres passam a constituir um dos principais elementos que compõem a infraestrutura de software para a nuvem.

Com a introdução de orquestradores de contêineres à já sofisticada pilha de infraestrutura de contêineres, o protagonismo desta forma de virtualização vive, contemporaneamente, uma acentuação significativa. Esse fenômeno se deu porque sua utilização substituiu arcabouços de nuvem complexos e de difícil manutenção por uma solução simples, barata e leve, que se adequa bem à parcela majoritária dos sistemas de informações em operação.

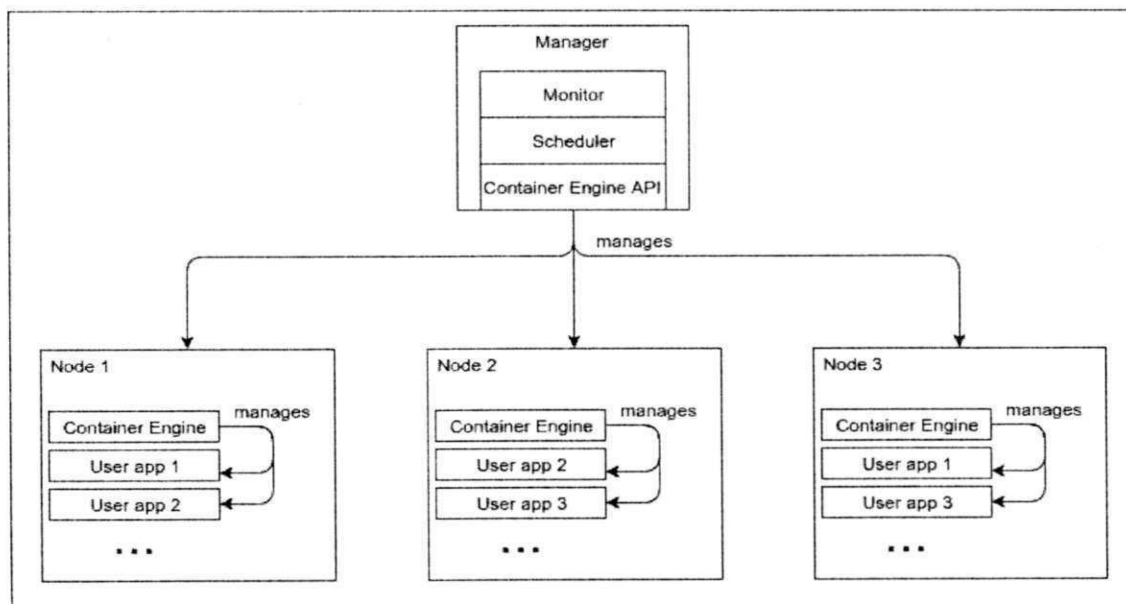


Figura 1.1: Arquitetura de um orquestrador de contêineres

1.2 Problema Abordado

PSN's possuem infraestruturas massivas e por isso se beneficiam dos custos mais baixos da operação em escala e da capacidade de, explorando seu poder de localidade e de redundância, oferecer SLA's atrativos [4; 8].

No entanto, desde que a nuvem pública se tornou uma realidade, em meados da década de 2000, a desconfiança sobre como a perda da propriedade de infraestrutura se reflete na perda da propriedade dos dados representou um impecilho para a adesão de vários potenciais usuários a este tipo de infraestrutura de TI [37].

Essa preocupação se tornou o fator que protagoniza a escolha por alternativas de nuvem privadas e híbridas, que acarretam altos custos e desempenho inferior [4], o que se agrava com a introdução de opções de instâncias não virtualizadas na nuvem pública.

Isso ocorre porque em um ambiente de nuvem, não há meios para garantir a segurança de dados e código sensíveis [37]. Ainda pior, se um provedor de serviço de nuvem é visto como um ator malicioso em potencial [5], o cenário se torna seriamente mais alarmante. Vários vetores de ataque tornam-se possíveis, desde pessoal de suporte agindo independentemente, a agentes externos, até à própria gerência do provedor de serviço de nuvem.

No contexto de aplicações de terceiros executando em um ambiente de nuvem, portanto, assume-se um adversário ativo com intenções maliciososas, gozando privilégios de usuário raiz nos nós que compõem o sistema, total liberdade para interceptar, examinar, modificar e bloquear todo tráfego de rede interno e externo, assim como acesso direto ao nó físico e seus componentes de *hardware*.

Este último aspecto implica que o adversário seja capaz de efetuar congelamentos de memória, o que então permitiria a prospecção de dados a partir de qualquer seção da memória principal, mesmo com proteções de software que forcem a rejeição de leituras a alguma área especial de memória.

As atuais contra-medidas de segurança para proteger aplicações não podem assegurar dados de aplicações remotamente implantados em uma nuvem. De fato, uma das poucas abordagens realizáveis ao problema de ofuscacação de dados quando o adversário possui altos privilégios na máquina hospedeira é o uso de segurança baseada em hardware [5].

Em resumo, a prática de mercado de hospedagem de serviços computacionais na nuvem

carece de um modelo de segurança que prevê ameaças originadas na própria nuvem. Ferramentas de infraestrutura de software, a participação de entidades neutras de autenticação e práticas de segurança adequados para este fim não são adotados.

Este trabalho aborda o problema de fornecer um modelo de segurança que proteja os dados sensíveis dos usuários na conjuntura atual da distribuição de produtos e serviços de TI e ferramentas que implementem as políticas de segurança estabelecidas por ele. Também julgou-se da alçada deste trabalho a avaliação quanto à viabilidade desta solução.

1.3 Contribuições e Relevância

Este trabalho propõe a introdução de um modelo de segurança que busca proteger os usuários de aplicações de desenvolvedores confiáveis de ameaças presentes em um ambiente de nuvem não confiável.

Para implementar este modelo de segurança, é necessário impedir que um agente malicioso dotado de grandes privilégios, como descrito em 1.3, possa ter acesso a dados recebidos do usuário. Para isso, a solução deve ser capaz de impedir que este agente:

1. Leia dados diretamente do processo da aplicação na memória através de uma chamada de sistema operacional.
2. Intecepte dados pela comunicação entre o usuário e o servidor na rede.
3. Capture dados utilizando técnicas de acesso físico à memória.
4. Desvie ou copie dados de dentro do processo da aplicação para outra região de memória sobre a qual ele tenha controle.

Para atender a esses requisitos, é proposto neste trabalho uma metodologia de implantação e utilização que assegura que o usuário está trocando informações com uma aplicação localizada em um enclave SGX e que esta aplicação não foi modificada da original submetida pelo desenvolvedor confiável.

Os componentes desta metodologia são um cliente de implantação, uma Terceira Parte Autenticadora, um cliente de usuário e, principalmente, um orquestrador de contêineres capaz de instanciar contêineres com SGX. Estes componentes foram implementados e fazem

parte das contribuições deste trabalho. Em especial, o orquestrador de contêineres compatível com SGX, que denominou-se de SCO (Secure Container Orchestrator), que implementa alguns mecanismos, discutidos à frente, que tornam possível o cumprimento dos passos desta metodologia.

Utilizando SCO e os demais componentes, este trabalho demonstra ser possível oferecer na nuvem um serviço de hospedagem que respeita os rígidos padrões de privacidade estabelecidos pelo modelo de segurança proposto.

Para a indústria, essa demonstração tem a importância de permitir que PSN's interessados em atrair clientes receosos quanto a perda de dados, o que representa uma parcela significativa e inexplorada do mercado, possam oferecer um serviço que atenda altas demandas de segurança. Esta consequência tem a mesma relevância para os proprietários de aplicações procurando aliviar os altos custos acarretados pela manutenção de uma infraestrutura própria.

Clientes de PSN's que atualmente expõem os dados de seus clientes à infraestrutura não confiável se beneficiam por poder divulgar a seus usuários a adoção de políticas mais seguras com seus dados, e estes clientes deixam de ter seus dados vulneráveis a acesso indevido.

Para avaliar a viabilidade de SCO e sua metodologia quanto ao desempenho, foi realizada uma bateria de testes analisando aspectos tipicamente relevantes ao PSN, como poder de elasticidade e agilidade de implantação, além de investigar o impacto do uso de SGX em aplicações de nuvem escaláveis. O desempenho de SCO em todos estes aspectos foi comparado com a solução de orquestração convencional mais proeminente disponível no mercado, o Kubernetes.

Este trabalho também proporciona um estudo dos custos operacionais da execução de aplicações em nuvem sob os parâmetros de segurança e privacidade mais severos implementados por SCO e sua metodologia. Este estudo de custos operacionais procura proporcionar relacionamentos diretos entre o custo de utilização do SCO, com características de segurança e menor maturidade como artefato de software, e Kubernetes. Estes fatores diminuem a eficiência e portanto a atratividade da solução proposta.

Por isso, relacionamentos entre aspectos de desempenho de orquestração de contêineres avaliados e a importância dada a estes aspectos pela organização são estabelecidos e geram um limiar para orientar decisões de negócio. Esta relação é outra contribuição significativa

deste trabalho.

Para contextualizar o estudo de custos operacionais proposto neste trabalho, alguns perfis de usuário de PSN foram traçados e aplicados, demonstrando como o custo total da utilização do SCO pode afetar diferentemente perfis de usuário distintos.

1.4 Objetivo

O objetivo deste trabalho é propor um modelo de segurança e uma implementação conceitual que orientem a criação de soluções adequadas às necessidades de privacidade e segurança de vários perfis de usuários desamparados pelas alternativas contemporâneas de proteção de dados. Nestes perfis se enquadram, por exemplo, as grandes, médias e pequenas organizações que utilizam serviços de nuvem privada por sigilo industrial, armazenamento de dados médicos, informações pessoais sensíveis, transações e documentos comerciais privados, segredos de estado, etc.

Especificamente, este trabalho tem a intenção de solucionar o problema de potenciais clientes de serviços de hospedagem na nuvem que não podem expor dados sensíveis de seus usuários a este ambiente potencialmente hostil, além de oferecer-lhes meios para orientar a decisão de negócio de utilizar a solução de segurança de orquestração de contêineres na nuvem proposta.

1.5 Metodologia

A etapa inicial deste trabalho foi o estudo das práticas de implantação de aplicações *web* na nuvem e seus métodos e visão de segurança.

Compreendendo estas práticas e isolando suas principais relações de vulnerabilidade, foi feita a concepção de um modelo de segurança que prevenisse a exploração dos participantes mais vulneráveis nesta relação. De fato, neste estudo observou-se que, apesar do Provedor de Serviço de Nuvem ser a entidade mais comumente atacada por agentes maliciosos, é a relação entre usuários de serviços de terceiros hospedados na nuvem e Provedores de Serviço de Nuvem a que revela a maior vulnerabilidade.

O modelo de segurança proposto deveria, portanto, considerar o Provedor de Serviço

de Nuvem um adversário malicioso. Esta abordagem teve a consequência de engajar uma necessidade dos próprios Provedores de Serviço de Nuvem de estabelecer uma relação de confiança com potenciais clientes que se preocupam com a privacidade dos dados em seus serviços.

A seguir, foi realizada a criação de uma metodologia de implantação e utilização de uma aplicação de um usuário do serviço de hospedagem na nuvem e um do usuário da aplicação hospedada. Estas metodologias deveriam permitir, utilizando as técnicas mais contemporâneas de proteção de dados e outros atores e processos necessários, que usuários da aplicação tivessem garantias de que seus dados estivessem inacessíveis para usuários do Sistema Operacional com altos privilégios e que tivessem domínio total da infraestrutura de rede. Caso tais garantias não pudessem ser atendidas, o usuário da aplicação deveria então poder revogar a transação.

Para este fim, a introdução do Intel SGX foi oportuna. As propriedades do Intel SGX foram instrumentais para a fase seguinte, de implementação da metodologia de implantação. Esta etapa foi composta principalmente pela criação de um orquestrador de contêineres conceitual capaz de fornecer contêineres com acesso aos *drivers* do *hardware* especial com SGX na máquina hospedeira, além de prover gerenciamento de ciclo de vida, sistema de imagens e esquema de balanceamento de carga sensíveis a atestação, partes fundamentais para a metodologia de implantação e utilização.

Por fim, etapas de otimização e avaliação de viabilidade da ferramenta foram feitas através de uma metodologia de avaliação isolada de parâmetros relavantes à utilização de *engines* de orquestração de contêineres. Estes parâmetros são quantificados e comparados com o equivalente em outras ferramentas. O fator de comparação é então utilizado em uma análise mais holística para comparar os custos do uso desta ferramenta em um ambiente real.

1.6 Organização da Dissertação

Este trabalho se organiza em:

- Capítulo 2: Tecnologias e Trabalhos Relacionados
- Capítulo 3: Apresentação da solução de orquestração segura de contêineres SCO

- Capítulo 4: Descrição do funcionamento de SCO utilizando aplicações de nuvem segura com SGX

- Capítulo 5: Avaliação do desempenho de SCO

- Capítulo 6: Conclusões e trabalhos futuros

Capítulo 2

Tecnologias e Trabalhos Relacionados

2.1 SGX

Como um requisito tenro dos sistemas de nuvem, a proteção de código privilegiado em ambientes baseados em máquinas virtuais tem sido uma matéria continuamente revisitada na última década. Abordagens diversas atacam este problema. Técnicas como a proteção de chamadas de sistema operacional a uma região de memória [11; 14; 12] e isolamento de aplicações por sandboxing [21] são exemplos de soluções de software de relativo sucesso.

Abordagens de hardware como co-processadores seguros [7; 6] também foram implementadas, especificamente na implantação de Sistemas de Gerenciamento de Bancos de Dados seguros, mas tiveram adoção limitada por sua pouca generalidade e custo proibitivo.

Com o intuito de oferecer uma solução competitiva para este problema mais genérico de execução de aplicações em ambientes potencialmente hostis, fabricantes de *hardware*, proeminetemente a Intel, desenvolveram extensões em seus *chips* que implementam muitos dos princípios utilizados em soluções prévias. A mais recente e completa destas extensões, o *Software Guard Extension* (SGX), permite ao usuário criar enclaves de memória protegidos por *kernel* e *hardware*. O usuário de SGX pode submeter seu código para formar um enclave e, uma vez que o processo de criação é concluído, acessos à área de memória delimitada pelo enclave são bloqueados pelo *kernel* do sistema operacional. Além disso, os dados e código residentes no enclave permanecem encriptados na memória, com uma chave gerada em sua criação, por *hardware* em tempo de escrita. Quando copiados para o processador, são decriptados também por *hardware* e reencriptados quando reescritos na memória, impedindo

UFCG/BIBLIOTECA/BC

a ataques físicos que se baseiam no congelamento do estado da memória para leituras diretas à região protegida. Desta forma, a base computacional confiável não pode ser acessada mesmo por um adversário poderoso sem uma abordagem *side-channel*, ou através de uma falha no código do enclave, que pode ser dificultada com artifícios de engenharia de software [13; 2].

Para conferir segurança aos dados enviados e recebidos do enclave durante a comunicação com um cliente remoto, o Intel SGX também provê um protocolo de atestação remota [27]. Este procedimento consiste, em parte, de um protocolo Sigma modificado para facilitar uma troca de chaves de Diffie-Hellman entre cliente e servidor [9]. A troca resulta em uma chave simétrica compartilhada usada para comunicação com o processo executando dentro do enclave. Esta é uma funcionalidade muito conveninente para estabelecer um relacionamento de confiança entre um cliente e um servidor situado em um ambiente de hospedagem potencialmente hostil, onde agentes agressores podem interceptar dados antes de sua chegada à máquina de destino.

A atestação remota SGX também atesta a identidade do processador SGX do servidor alvo. O serviço de atestação requisita uma assinatura da máquina sendo atestada, que é criada aplicando-se as chaves implementadas em *hardware*, inerentes aos processadores, sobre identificadores do cliente e do servidor sendo atestado. Esta assinatura é verificada pelo Intel Attestation Service (IAS) que garante ao cliente a identidade do processador.

A atestação remota e a proteção de dados na memória, juntos, constituem uma plataforma que favorece o aumento no nível de segurança das transações efetuadas na nuvem, possibilitando modelos de segurança mais céticos quanto aos atores tradicionalmente confiáveis.

2.2 Trabalhos Relacionados

Vários trabalhos foram identificados que se dedicam à proteção de software implantados em ambientes potencialmente hostis. Esses trabalhos foram encontrados em fontes públicas de pesquisa (Google Acadêmico) e repositórios privados, como as bibliotecas digitais da ACM e IEEE, entre outros. Em geral, eles abordam alguns dos aspectos cobertos pela implementação do modelo de segurança proposto neste artigo. Exceto pelo SCONE, cuja relação de custo e benefício é discutida em seu tópico, nenhuma solução foi encontrada que se ade-

quasse ao modelo de ameaça aqui tratado, em especial no que diz respeito à integridade da aplicação. Entre as condições assumidas por seus autores está que a aplicação já está seguramente implantada na sua infraestrutura, um cenário pouco realista no contexto moderno de aplicações *web* hospedadas em um PSN.

Estes trabalhos foram agrupados em três. O primeiro grupo é o de contêineres seguros, que apresenta soluções de contêineres Linux com alguma capacidade de obfuscação ou negação de acesso a dados executando em um sistema operacional e, portanto, sobre um *kernel* malicioso.

Então, são tratadas abordagens de software que protegem aplicativos de software privilegiado. Estas abordagens consistem em limitar o acesso do sistema operacional a áreas da memória através da modificação de elementos do sistema operacional que garantem a proteção de certas áreas da memória. Esta propriedade é obtida inserindo modificações nas chamadas de acesso ao hardware e efetuando criptografia nos dados sensíveis.

Por fim, tratamos de soluções baseadas em componentes de hardware que asseguram a inacessibilidade de dados na memória por chamadas de sistema operacional não autorizadas, e se encarregam da criptografia dos processos residindo nelas.

2.2.1 Contêineres seguros e outros ambientes

O principal trabalho relacionado a contêineres que oferecem algum tipo de proteção a software não autorizado atualmente é o SCONE (Secure CONtainer Environment) [5]. SCONE é uma sofisticada plataforma que supostamente oferece contêineres seguros com baixa sobrecarga protegidos por SGX. Essa solução usa seu próprio mecanismo de compilação para assegurar que a aplicação de nuvem do usuário seja conformante com o subconjunto de instruções C permitido dentro de enclaves do SGX.

Na prática, SCONE consiste em uma imagem de contêiner curada, isto é, analizada minuciosamente e capaz de executar a infraestrutura de software de SGX, um compilador que traduz o programa original C para o idioma C que SGX requer, além de serviços de atestação local em cada nó do serviço de nuvem e de configuração e atestação remota para todos os nós executando na mesma infrestrutura de nuvem.

O SCO difere do SCONE uma vez que não aplica nenhuma sobrecarga adicional no nível de aplicação (chamadas feitas pelo SCONE podem levar até 1.2x o tempo normal), além de

uma consideravelmente grande infraestrutura pré-instalada na nuvem, para qual orquestradores de contêineres correntes e outras plataformas de nuvem não oferecem suporte. Mais notavelmente, a abordagem de uso do *SDK* SGX do SCO deixa ao desenvolvedor a tarefa de customizar suas aplicações de acordo com suas necessidades, o que não envolve o processo de recompilação substancialmente propenso a faltas e que introduz uma ampla base de código gerado automaticamente, característico do SCONE. Por outro lado, esse processo se torna transparente ao usuário, que não tem que escrever codigo nativo SGX.

2.2.2 Abordagens de software que protegem aplicativos de código privilegiado

Como citado na seção 1.2, varias abordagens de proteção de código em ambientes hostis baseadas em software foram propostas nas últimas décadas. Overshadow [11], InkTag [19] e SP^3 [35] são exemplos de sistemas que compartilham a propriedade de proteção de memória de chamadas de sistema operacional potencialmente maliciosas. Virtual Ghost [14], em especial, implementa proteção de aplicações destas chamadas alterando os mecanismos de controle do *kernel* com o hardware, criando a capacidade de dinamicamente reservar espaços de memória criptografados e que não podem ser lidos ou escritos pelo sitema operacional. Apesar de serem alternativas úteis para a maioria dos cenários de ataque, a vulnerabilidade a ataques baseados em BIOS ou memória fria desencorajam a utilização destas soluções no contexto de nuvem, em especial para o cenário, isto é, para o modelo de ameaça, abordado neste trabalho.

Outro trabalho com características excepcionais é o Minibox [21], um arcabouço baseado em isolamento em *sandbox* mutuamente inacessíveis. Como outras abordagens, a estratégia utilizada por este sistema é a utilização de um hipervisor seguro que exerce controle sobre as chamadas de sistema, especificamente sobre as permissões de acesso a regiões específicas de memória. Devido a sua mecânica de atestação da integridade do hipervisor, é uma solução mais adequada ao ambiente de nuvem. Vale ressaltar, no entanto, que demonstra vulnerabilidade por não oferecer garantias da integridade da aplicação durante a fase de implantação, o enfoque do modelo de segurança aqui demonstrado, e de não oferecer proteção a ataques físicos. Outro empecilho identificado nesta solução é a notável diminuição no desempenho

geral das aplicações causada pela trocas caras de contexto de memória, como demonstrado pelos autores.

TrustVisor [22] e CloudVisor [36] são as soluções seminais baseadas em software, na qual várias outras baseiam seus modelos de atestação. Estas soluções utilizam também hipervisores seguros atestáveis, mas com um enfoque em diminuir ao máximo a Base Computacional Confiável (TCB), isto é, a porção de uma aplicação que opera com dados sensíveis e requer proteção do sistema operacional malicioso.

Como consequência destas propriedades, TrustVisor e CloudVisor oferecem baixos níveis de sobrecarga, em volta dos 7% de acordo com os autores. No entanto, assim como seus sucessores, a falta de garantias da integridade da aplicação, e de mecânicas de proteção contra ataques físicos torna a solução inviável para o modelo de ataque proposto aqui.

2.2.3 *Trusted Hardware Support*

Módulos de plataforma para criar ambientes seguros como contêineres e máquinas virtuais podem ser encontrados na literatura relativamente recente, representados pelo Flicker [23], que suporta selamento de dados de aplicações e outras funcionalidades desejáveis também utilizando extensões de *hardware* de gerações anteriores presentes em processadores de *commodity*. Apesar de ser uma solução completa, inclusive com suporte a atestação remota, seu modelo de ameaça também não prevê que o atacante seja responsável pela implantação da aplicação e tenha liberdade para alterá-la. Outra característica indesejável desta solução é a significativa sobrecarga causada, em especial nas etapas de inicialização e estabelecimento da TCB, que pode chegar a 47% do tempo total de CPU para aplicações testadas pelos autores.

Co-processadores seguros têm sido também soluções como ambientes confiáveis para executar aplicações com dados sensíveis. Sion e Bajaj [7] efetuam uma implementação voltada a segurança de Bancos de Dados. Em seu trabalho, eles criam um Sistema de Gerenciamento de Banco de Dados (SGBD) que utiliza co-processadores criptográficos, como o IBM 4764, que impedem livre acesso de um atacante aos dados protegidos, mesmo que este seja um usuário de altos privilégios e até tenha acesso físico à máquina alvo. Neste trabalho, os autores se dedicam a minimizar a TCB, a fim de aumentar o desempenho e, principalmente, diminuir o alto custo de operação dos co-processadores.

A solução implementada com o uso de co-processadores acarreta altos custos e, apesar de generalizável para outros problemas além da proteção de dados armazenados e transitando em um SGBD, não há preocupação dos autores em proteger outros níveis de uma aplicação, tampouco assegurar a integridade durante a execução. O arcabouço proposto baseado em co-processadores seguros também falha no que diz respeito à escalabilidade, já que seu custo total de propriedade por ciclo de CPU é pelo menos uma ordem de magnitude mais alto que o de processadores de *commodity*.

Capítulo 3

Orquestração de Contêineres na Nuvem: Um Modelo Seguro

Como pôde ser visto nas seções introdutórias do capítulo 1, é a visão deste projeto que o *modus operandi* da hospedagem de serviços de computação em geral hoje se encontra obsoleto no que diz respeito à segurança. Isso ocorre porque as soluções de segurança disponíveis são de difícil escalabilidade, devido ao alto custo ou baixo desempenho, ou não se adequam, com excessão do SCONE, ao procedimento de implantação de aplicações na nuvem.

Este trabalho procura uma solução de metodologia de implantação e utilização, aliando tecnologias de privacidade em ambientes hostis e infraestrutura de nuvem já existentes para um modelo de segurança que abrange a metodologia de hospedagem atual.

Neste capítulo, serão definidos as premissas assumidas sobre os atores, processos e características sobre o ambiente estudado, a proposta de implementação de uma metodologia que atenda aos requisitos de segurança e um breve estudo do seu funcionamento.

3.1 Modelo de Atacante

Neste trabalho são assumidos um adversário local e um adversário de gerenciamento que podem ser considerados o mesmo agente ou não. O adversário local tem acesso a todas as máquinas hospedeiras de réplicas da aplicação e o adversário de gerenciamento tem acesso à rede, e ao *software* de infraestrutura de nuvem.

3.1.1 Adversário Local

O adversário local é definido como um atacante malicioso ativo com privilégio no sistema operacional a nível de *kernel*, com a capacidade de executar código com qualquer conteúdo previamente instalado no sistema operacional. Além disso ele é capaz de efetuar leituras à memória através de uma interface externa, ou seja, sem modificar o sistema operacional. Assim como o modelo de segurança da própria plataforma SGX, é considerado um adversário com acesso físico à máquina, podendo monitorar e alterar as interfaces físicas e efetuar leituras de memória fria.

Não é considerado no modelo de ameaça ataques de Negação de Serviço. Para um usuário de alto privilégio no sistema operacional, finalizar o processo hóspede, desautorizar a utilização da rede, encerrar o contêiner que hospeda a aplicação ou simplesmente desligar a máquina hospedeira são alternativas simples para impedir o funcionamento do serviço no contexto estudado.

Neste contexto, o atacante está motivado financeiramente pelo seu *SLA*, que obriga-o a cumprir altas taxas de disponibilidade (orbitando os 99,995% atualmente), a não impedir o funcionamento das aplicações hospedadas, por isso é assumido que este tipo de ataque não se concretizaria.

Além disso, a exemplo do modelo de segurança da plataforma SGX, ataques de canal lateral também não são considerados no modelo de ameaça.

3.1.2 Adversário de Gerenciamento

É utilizado neste trabalho o conceito de adversário de gerenciamento. Este tipo de atacante obtém total controle da rede, adotando-se o padrão Dolev-Yao de modelo de ameaça para comunicação em rede, o que dá ao adversário a habilidade de bloquear, injetar ou modificar o tráfego de rede entre quaisquer entidades do sistema.

Além de atender a este modelo clássico, o adversário de gerenciamento também é o responsável pela implantação da aplicação no nó alvo. É assumido que este adversário deve, portanto, receber do proprietário da aplicação seu código fonte, ativos técnicos e outros utilitários necessários à sua instalação, escolher os nós onde a aplicação será instalada e publicar ao proprietário os endereços de acesso.

Durante esse procedimento, o adversário tem acesso a estes items (doravante referidos como *bundle*), incluindo a liberdade de modificar, incrementar, decrementar, remover, copiar, analisar e auditar quaisquer conteúdos. Espera-se do PSN potencialmente malicioso, aqui no papel de atacante de gerenciamento, somente o cumprimento do prazo de instanciação definido no *SLA*. Pela mesma razão, também não espera-se ataques de Negação de Serviço provenientes do atacante de gerenciamento.

3.2 Modelo de Segurança e Metodologia

Dado o modelo de ameaça exposto na subseção anterior, são necessárias, para a hospedagem segura de aplicações de terceiros em um ambiente com este modelo de atacante, pelo menos as seguintes 4 garantias de segurança:

1. **Negação de acesso a dados na memória:** Chamadas de sistema operacional de usuários com altos privilégios podem capturar dados diretamente no processo da aplicação. É necessário que leituras à TCB sejam bloqueadas a nível de *kernel* ou superior. Também são desejáveis garantias mais severas, como criptografar a TCB residindo na memória, para evitar ataques através da BIOS, de memória fria e outros ataques físicos.

2. **Comunicação segura:** Um atacante não deve ser capaz de capturar pacotes desprotegidos na rede, ou pacotes criptografados cujas chaves estejam acessíveis.

3. **Atestação de identidade:** O sistema deve ser capaz de provar para um usuário remoto ser detentor das propriedades de segurança assumidas.

4. **Verificação da integridade da aplicação:** Deve ser impossível ao atacante modificar a aplicação original, ou pelo menos possível ao usuário desta aplicação verificar se seu conteúdo foi modificado por um atacante antes de efetuar uma transação com a aplicação.

Soluções legadas de privacidade em ambientes potencialmente hostis existentes, como várias das soluções baseadas em software já citadas, são capazes de atender pelo menos a duas destas garantias, 1 e 3 (não considerando ataques físicos).

Como o uso de SGX por si já possibilita o estabelecimento de chaves simétricas para comunicação secretiva, passa a atender à garantia 2 de comunicação segura.

No entanto, em um contexto de nuvem, a tarefa de implantação fica sob a tutela do PSN agressor. O *bundle* enviado pelo proprietário da aplicação para instalação contém todo o código, inclusive as partes residentes na TCB. Isso significa que o código em que consiste a TCB passa a estar comprometido, podendo ser livremente modificado pelo atacante de gerenciamento.

No cenário de um ataque, uma vez que a aplicação é instanciada, código malicioso é introduzido à TCB e o serviço corrompido passa a estar diponível. Quando um usuário inicia uma transação, realiza uma atestação com o IAS, que deve corroborar a autenticidade do serviço SGX e gerar uma chave simétrica. Os dados do usuário na transação são transferidos para o enclave SGX ainda criptografados com as chaves simétricas negociadas durante a fase de atestação. Uma vez decriptografados dentro do enclave, o código malicioso do adversário pode acessar as variáveis que contêm os dados do usuário.

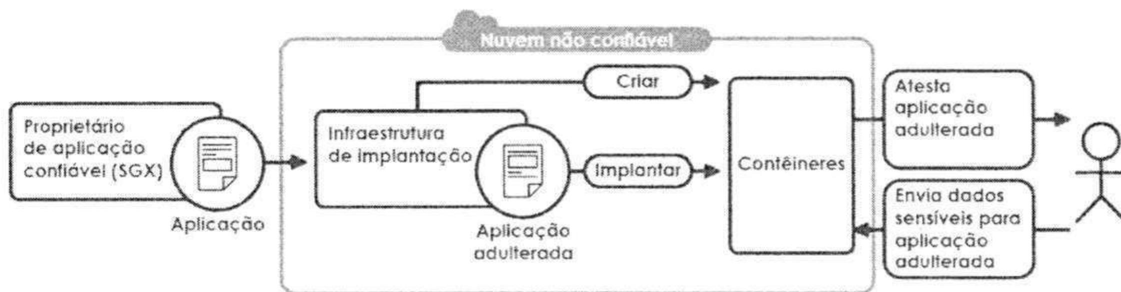


Figura 3.1: Esquema de adulteração de uma aplicação com SGX

Nesta etapa, o atacante pode introduzir no enclave SGX código malicioso capaz de realizar leituras a dados protegidos. Um exemplo de introdução de código malicioso pode ser visto no código fonte 2.1 em que uma thread maliciosa é introduzida na parte desprotegida da aplicação e no código fonte 2.2 em que o código que publica segredos da aplicação utilizado pela *thread* maliciosa é utilizado para enviar segredos do usuário para fora do enclave.

```
1
2 int main()
3 {
4     // Create enclave!
5         ret = sgx_create_enclave(
```

```
6                   ENCLAVE_NAME,            // const char *file_name,
7                   SGX_DEBUG_FLAG,          // const int debug,
8                   &launch_token,           // sgx_launch_token_t *
                        launch_token,
9                   &updated,                // int *launch_token_updated,
10                  &eid,                    // sgx_enclave_id_t *enclave_id,
11                  NULL                     // sgx_misc_attribute_t *
                        misc_attr
12      );
13  char *ret = (char *) calloc(secret_char_n + 1, 1);
14  std::thread legit_thread(
15                  hide_secret_and_publish_string, eid, * ret
16      );
17  legit_thread.join();
```

Código Fonte 3.1: Introduzindo *Thread* maliciosa na parte desprotegida da aplicação

```
1   #include "enclave_t.h"
2   #include <string.h>
3
4   char* my_char = "A user's secret! I must be processed before this
       variable can be made available outside the enclave."
5
6   void hide_secret_and_publish_string( char *ret )
7   {
8   // introducing malicious function call!
9       sgx_get_secret_string( char * ret )
10  //
11      do_some_work_that_protects( my_char );
12      memcpy( ret, my_char, strlen( my_char ) );
13  }
14  /* Malicious function! */
15  void sgx_get_secret_string( char *ret )
16  {
17      memcpy( ret, my_char, strlen( my_char ) );
18  }
```

Código Fonte 3.2: Acessando dados sensíveis do usuário de dentro do enclave

Não é assumido neste modelo, assim como é a prática de mercado, que o proprietário da aplicação tenha controle sobre as etapas de implantação da aplicação, o que, como demonstramos, permite a introdução de código malicioso na TCB, desrespeitando a garantia 4 do modelo de segurança prosposto neste cenário.

A alternativa mais intuitiva para garantir a integridade da aplicação seria obter evidências de que o processo de implantação ocorre idoneamente. Atestar cada componente de *software* potencialmente agressor na pilha de infraestrutura, desde os servidor de entrega da aplicação, incluindo os mecanismos de orquestração e gerência de rede, até hipervisores e outras plataformas de virtualização. Essa abordagem, apesar de requerer a criação de um extenso e elaborado arcabouço de mecanismos de atestação e autenticação, é conceitualmente possível. Outros trabalhos, como TrustVisor [22], fiam a confiabilidade de seus serviços na possibilidade de verificar a integridade da ferramenta em si.

Além da evidente complexidade envolvida na atestação de integridade sistêmica de toda a infraestrutura remota de um PSN, esta alternativa também deve acarretar altos níveis de sobrecarga, cuja magnitude é desconhecida, mas que representa altos riscos para o tipo de proprietário de aplicação representado aqui.

Por estas razões, a possibilidade de garantir a probidade e robustez contra ataques internos de toda pilha de infraestrutura de software na nuvem é de difícil realização e pode ser proibitivamente cara. Outra alternativa, no entanto, é possível se uma garantia de disponibilidade for assumida por parte do PSN. Esta técnica consiste em um processo de verificação da integridade da aplicação implantada.

Para isso, o desenvolvedor deve atestar sua própria aplicação. A atestação gera, além de um par de chaves simétricas e garantias da autenticidade do hardware SGX, um *quote* Q que pode resumidamente ser descrito como $Q = \{v, sig, G, qe, pce, n, vcpu, A, MRENCLAVE, MRSIGNER, pid, ve, l(sig), sig\}$, dos quais v, qe, pce, $vcpu$ e ve são identificadores de versões de segurança de diferentes componentes, G e N são identificadores do processador, A é agregador de atributos do enclave sig é um assinatura encriptada do enclave e $len(sig)$ seus tamanho em bytes e $MRSIGNER$ é um identificador único criptografado do processador. Similar ao $MRSIGNER$, o $MRENCLAVE$ é também um identificador criptografado, mas que identifica unicamente um enclave.

Neste ponto é proposta a introdução de um novo ator na metodologia. Além do proprietário da aplicação, do PSN não confiável e do usuário final da aplicação, ocorre a participação de um agente confiável cuja finalidade é publicar *MRENCLAVEs* de aplicações seguras. Doravante este ator é referido como *Third Part Authentication Entity* (TPAE). Na prática, esse papel pode ser desempenhado por autoridades certificadoras que já são consideradas confiáveis na maioria dos modelos de segurança no contexto de certificados digitais.

O usuário da aplicação final deve também ser capaz de realizar uma atestação SGX com o processo do enclave, utilizando um cliente capaz realizá-la e tratar seus valores de saída. Este cliente é capaz de extrair o *MRENCLAVE e* da aplicação, do *quote Q* adquirido no processo de atestação e submetê-lo à TPAE. A TPAE então compara o valor do *MRENCLAVE* submetido com o publicado pelo proprietário da aplicação e responde ao cliente do usuário sobre o sucesso da comparação. Se os valores de *MRENCLAVE* correspondem, a transação deve prosseguir. Caso contrário, se os *MRENCLAVES* da aplicação atestada na nuvem e o publicado pelo proprietário da aplicação divergirem, a transação deve ser revogada e o usuário alertado de que a aplicação original sumbetida ao PSN pelo proprietário da aplicação teve seu código fonte ou outros recursos modificados e não pode mais ter sua confiabilidade garantida.

3.2.1 *Secure Container Orchestrator*

Tornou-se urgente, no contexto deste trabalho, a criação de uma ferramenta de infraestrutura de nuvem compatível com o modelo de segurança proposto. Nesta década ganhou popularidade um tipo de virtualização que melhor se compatibiliza com as necessidades do típico cliente de um PSN. Se demonstrou necessário prover um crescimento horizontal, isto é, no número de nós participantes, de um serviço de nuvem com maior agilidade, a fim de atender os exigentes padrões de disponibilidade de serviços do típico cliente comercial. Como discutido no capítulo 1, a virtualização por contêineres, que compartilha o *kernel* do sistema operacional hospedeiro e isola os espaços virtuais a nível de usuário, mantendo sistemas de arquivos, binários, espaços de identificadores de processos e de rede também separados, se tornou a solução capaz de responder a esses requisitos.

No modelo de segurança proposto, é assumido que o PSN tenha uma infraestrutura de nuvem capaz de provisionar recursos computacionais nativamente habilitados a executar apli-

cações em enclaves SGX. Especificamente, estes recursos computacionais devem se adequar às necessidades dos proprietários de aplicações baseadas na nuvem contemporâneos, o que inclui grande elasticidade de serviço em curtos espaços de tempo.

Estas necessidades de agilidade e segurança requeridas pelo modelo de serviço de nuvem proposto neste trabalho são atendidas pelo *Secure Container Orchestrator* (SCO). O SCO é um orquestrador de contêineres destinado a operar em uma infraestrutura de nuvem com *hardware* SGX disponível. Utilizando a popular plataforma de conteinerização de aplicações Docker, e unindo soluções consagradas de balanceamento de carga como HAProxy e de servidores web leves como Flask, SCO fornece orquestração de contêineres propícios à hospedagem de aplicações SGX.

3.2.2 Requerimentos para Orquestração Segura de Contêineres

Quatro principais funcionalidades são necessárias a um orquestrador de contêineres para, ora facilitar, ora possibilitar a instanciação de aplicações seguras nestes contêineres.

1. **Contêineres com acesso aos *drivers* SGX:** O simples acesso aos *drivers* que permitem utilizar as funcionalidades introduzidas no *kernel* do sistema operacional é essencial para a utilização de suas funcionalidades. Como a virtualização por contêineres isola os sistemas de arquivos dos contêineres hóspedes daquele da máquina anfitriã, o acesso ao *device* virtual do SGX não é possível sem a explícita vinculação deste *device* a nível de plataforma de conteinerização. Através desse *device* o kernel do sistema operacional pode se comunicar com o *hardware* especial para operações de gerência de enclave. SCO orquestra nativamente contêineres com acesso ao *device* virtual SGX, e consequentemente a seus *drivers*.

2. **Imagens Docker padronizadas para instanciação de contêineres com enclaves:** Docker utiliza imagens multicamada para permitir a reprodutibilidade de contêineres. Explorando essa funcionalidade, SCO possui uma imagem padronizada base que possui todas as dependências, binários, adaptações de ambiente e, principalmente, ao *kit* de desenvolvimento de *software*, para executar contêineres habilitados a hospedar aplicações SGX seguras, diminuindo a chance de incompatibilidade da imagem do usuário. Esse *kit* de desenvolvimento de *software* é utilizado para acessar as chamadas

SGX e passa a estar disponível no ambiente. Espera-se que a aplicação SGX do usuário realiza chamadas ao SDK nos diretórios da instalação padrão disponibilizada pela Intel.

3. **Balanceamento de Carga sensível a atestação remota com IAS:** Aplicações que manipulam dados sensíveis em um ambiente de nuvem pública utilizando SCO devem iniciar uma interação com uma atestação remota SGX. Uma vez realizada, somente o enclave atestado no contêiner alvo é capaz de decriptar os dados provenientes do alvo, e é o único confiável para isso. Por isso as transações seguras requerem uma relação $N \rightarrow 1$ entre clientes e contêineres. Uma consequência direta disso é que os mecanismos de balanceamento de carga devem obedecer uma política de redirecionamento de pacotes que fidelizam o tráfego provindo de um cliente para o contêiner por ele atestado. Devido à heterogeneidade dos tipos de conexão envolvidas (atestações são conexões TCP que não possuem recursos como *Sticky Sessions*, enquanto o tráfego das aplicações é, em geral, feito sobre HTTP), o balanceador de carga deve implementar um esquema de redirecionamento customizado que mantenha controle de que contêineres foram atestados por que clientes.

4. **Controle de remoção de contêineres** *Scale down*, isto é, a diminuição no número de contêineres executando um mesmo serviço, tipicamente devido à diminuição na carga de trabalho requerida por seus usuários, implica na remoção dos contêineres menos utilizados. Usualmente, se ainda houver conexões HTTP em curso com seus usuários em um contêiner terminado devido a *scale down*, a aplicação é migrada para um contêiner hóspede mais ativo. No entanto, contêineres hospedando aplicações com enclaves SGX que foram atestados por um cliente estabelecem conexões fixas com os usuários que o atestaram. Adotando as políticas usuais de redução de recursos em *scale down*, que são de difícil manipulação manual, clientes em transação com um contêiner hospedando um enclave atestado perderam completamente o contato. Isso ocorre porque aplicações com enclaves SGX não podem ser migradas para outros nós, mesmo que estes tenham também capacidades SGX. Cada enclave SGX é criptografado por uma chave exclusiva do processador que a criou. SCO torna a orquestração de contêineres sensível a conexões entre cliente e contêiner hóspede de seu enclave

atestado não permitindo a remoção até que um conjunto de requisitos que indicam o fim da transação sejam atendidos:

- O tempo máximo de reenvio de uma conexão TCP tenha se excedido sem que nenhum novo pacote tenha sido enviado ao contêiner alvo.
- O contêiner esteja realizando operações de Entrada/Saída desde o fim do limite de tempo estabelecido no item anterior.

3.3 Arquitetura, Fluxo de Implantação e Execução de aplicações do SCO

3.3.1 Arquitetura

A arquitetura de um serviço executando no SCO é composta por um gerenciador global de instâncias mestre, o *manager*, um processo auxiliar escravo local nos nós físicos que integram o *cluster*, um balanceador de carga global, localizado no nó do *manager* por padrão, e balanceadores de carga locais que executam em cada nó. Além disso, processos auxiliares de monitoramento executam localmente em cada nó e um agregador de dados executa no nó do *manager*.

O ***manager*** é o sistema de gerência de serviços do SCO que executa ele próprio em um contêiner Docker. O *manager* recebe as requisições de criação e encerramento de serviços, adição e remoção de contêineres em serviços existentes do proprietário da aplicação. Além disso, monitoramento e *auto scaling* de contêineres são responsabilidade deste módulo utilizando Docker Stats para aferir metadados dos contêineres.

O ***node daemon*** é um processo auxiliar que deve executar localmente em cada nó participante do *cluster*. Este módulo inclui servidores que aguardam por requisições de criação e remoção de instâncias enviados pelo *manager* e executa essas requisições na instalação Docker local. O *node daemon* também fornece estatísticas de uso de recursos para *auto scaling*.

O **auxiliar de monitoramento** é outro processo que executa, canonicamente, no nó do *manager*. É responsável por receber dados de consumo de recursos das instâncias executando

nos nós participantes do *cluster*. O auxiliar de monitoramento agrega os dados de consumo por seviço e os submete às políticas de *auto scaling* dos serviços definidas. Quando os gatilhos dessas políticas são acionados, o auxiliar de monitoramento requisita ao *manager* crescimento ou diminuição do serviço avaliado.

O **sistema de balanceamento de carga** é composto por um balanceador de carga principal, que é atualizado pelo *manager* quanto à localização dos nós que possuem instâncias de um certo serviço. Cada serviço por sua vez possui um balanceador de carga em cada nó onde tem instâncias. Quando são adicionadas instâncias de um certo serviço que não hospedava nenhum contêiner deste serviço, um novo balanceador de carga deste serviço é instanciado. Essa abordagem foi utilizada porque reduz o *down time* em um contexto em que é necessário reiniciar o balanceador de carga (que no SCO são contêineres Docker executando HAProxy) a cada novo usuário que estabelece uma conexão com um contêiner. Apesar de rápida, essa operação pode ser frequente em momentos de pico de acesso e acumular longas intermitências sem serviço. Com múltiplos balanceadores por nó, as atualizações ficam divididas entre os serviços. Adiante será tratada a razão da atualização no balanceador de carga a cada conexão entre o usuário e um contêiner estabelecida.

3.3.2 Fluxo de Implantação

O fluxo de implantação se inicia com uma requisição de criação de um serviço partindo do proprietário da aplicação para o servidor de requisições de submissão de aplicações do *manager*. Essa requisição fornece ao manager um *bundle*, que é composto por uma quadrupla $B = (i, d, p, S)$, sendo i um identificador, d um um *script* de instalação de dependências e execução de aplicação, que no contexto do Docker se denomina Dockerfile, p a aplicação empacotada e S um conjunto composto pelos parâmetros do contêiner. Essa requisição é respondida pelo *manager* com uma nova versão do Dockerfile inicial acrescida da imagem base Docker que cria o ambiente para instanciação de contêineres com enclaves citada na seção anterior. Sendo a imagem base pública e auditável. Essa resposta será utilizada para que o proprietário da aplicação gere um *MRENCLAVE* correspondente ao gerado pelo enclave no contêiner instanciado pelo SCO.

Na etapa seguinte, o *manager* faz a instanciação dos contêineres nos nós participantes da infraestrutura, e em seguida torna-as acessíveis à rede externa com a implantação dos

balanceadores de carga locais e atualização do balanceador de carga global. Os endereços de pontos de entrada da aplicação são publicados ao proprietário da aplicação e o fluxo de implantação se conclui.

Neste ponto, o desenvolvedor já deve ter publicado o *MRENCLAVE* aferido da aplicação já utilizando a imagem docker padrão do SCO (note que assume-se que, uma vez que tenha criado uma aplicação SGX, o desenvolvedor tenha acesso a *hardware* SGX) na TPAE junto a um identificador da aplicação.

3.3.3 Fluxo de Utilização

O usuário da aplicação utiliza para este fluxo um cliente especial que inclui os mecanismos necessários de atestação e autenticação da aplicação. O pacote do SCO dispõe de clientes específicos para cada uma de suas aplicações exemplo, mas as etapas são suficientemente automatizáveis para que clientes genéricos, que poderiam executar como um plugin ou um protocolo suportado por um navegador, sejam criados. É assumido um cliente SCO neste fluxo.

Na primeira etapa, o cliente envia para o ponto de entrada do serviço na nuvem a mensagem inicial do processo de atestação remoto SGX. O balanceador de carga global deve então redirecionar esta requisição para um balanceador de carga local do serviço em um dos nós da infraestrutura hospedando um de seus contêineres (figura 3.2). Este processo é feito com uma conexão TCP reservada e os balanceadores de carga estabelecem a conexão resiliente entre o cliente e o contêiner.

O sistema de balanceamento de carga no SCO que, como exposto anteriormente, é constituído de um contêiner Docker executando uma instância do balanceador de carga HAProxy, acompanha em cada uma de suas unidades um *daemon* que monitora nos *logs* do HAProxy o estabelecimento de uma nova conexão TCP de atestação. Quando esta conexão é detectada, uma nova regra de configuração é inserida no HAProxy roteando todo o tráfego originário do IP de origem ao destino escolhido pelo algoritmo de *scheduling* utilizado (*round robin por padrão*), como demonstrado na figura 3.3.

No final do processo de atestação SGX, o IAS envia o *quote* para o cliente SCO que extrai seu *MRENCLAVE* e, junto a um identificador da aplicação, envia uma requisição para um TPAE cadastrado. O TPAE verifica em sua base se o *MRENCLAVE* corresponde ao subme-

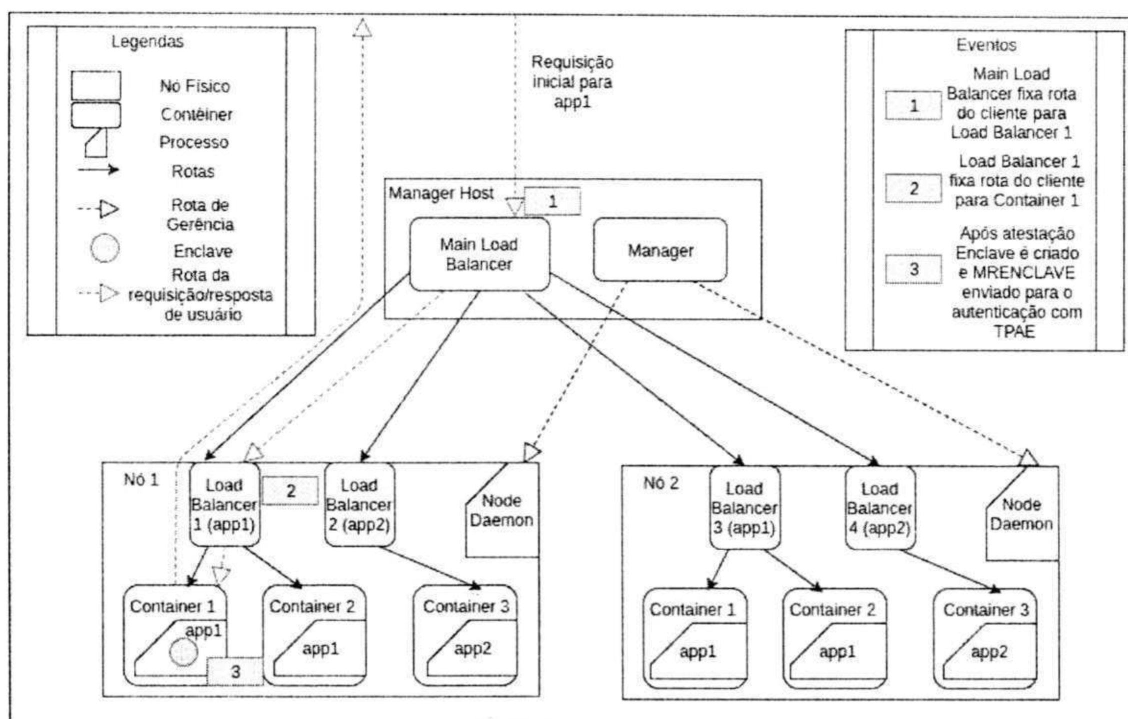


Figura 3.2: Arquitetura do SCO e Fluxo de utilização simplificado

tido pelo proprietário da aplicação. Em caso positivo, o cliente procede com a transação da aplicação. Do contrário, a conexão é fechada pelo cliente e o incidente é reportado, podendo ser o serviço introduzido em uma lista negra, dependendo da implementação do cliente.

Junto ao fluxo de implantação, o fluxo de utilização implementa o modelo de segurança proposto, impedindo a adulteração de aplicações pelo PSN, como visto na figura 3.1. A nova configuração, com a atuação da TPAE neste modelo de segurança é ilustrada na figura 3.4

3.4 SCO e Outras Plataformas de Orquestração de Contêiner

O SCO é um orquestrador de contêineres que pretende demonstrar a viabilidade da provisão de contêineres seguros, em sinergia com outros componentes que constituem um modelo de segurança adequado à realidade de infraestrutura de TI contemporânea.

Há, portanto, várias funcionalidades presentes em ferramentas de orquestração de contêineres sofisticadas que são abdicadas no SCO, como serviços de URL, federação de nuvens,

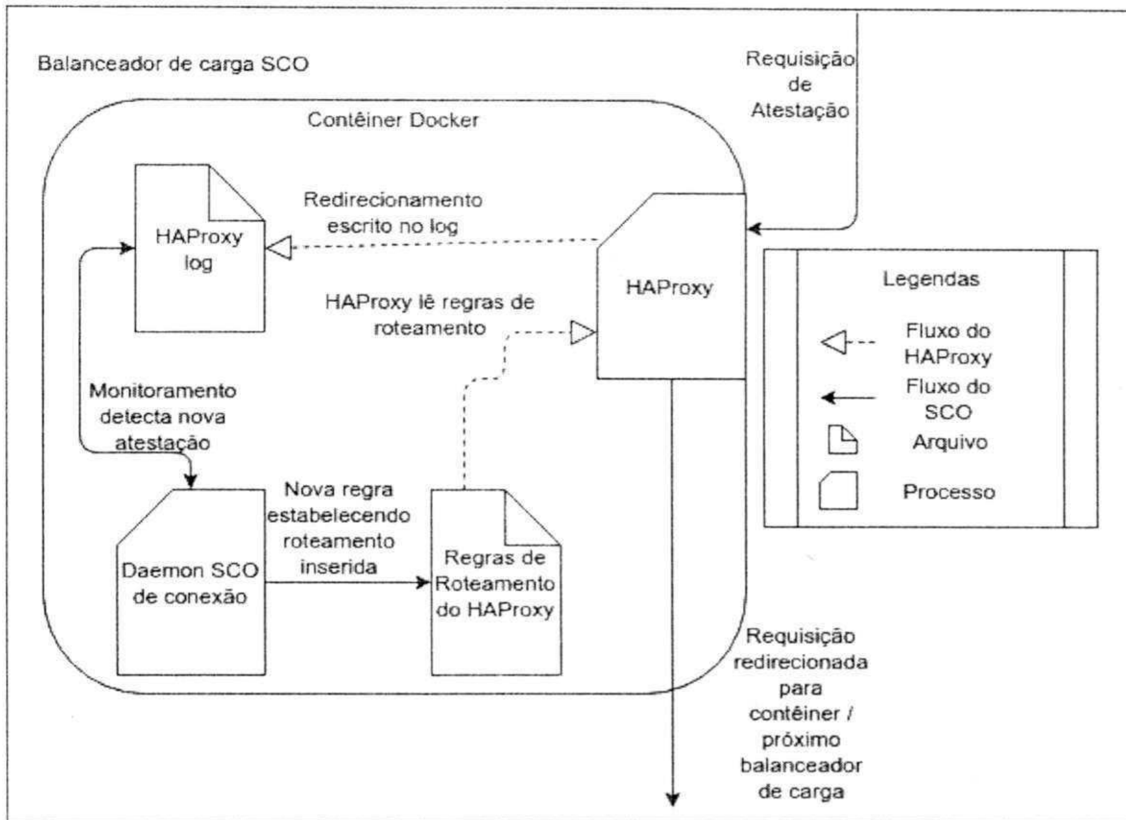


Figura 3.3: Balanceador de Carga SCO

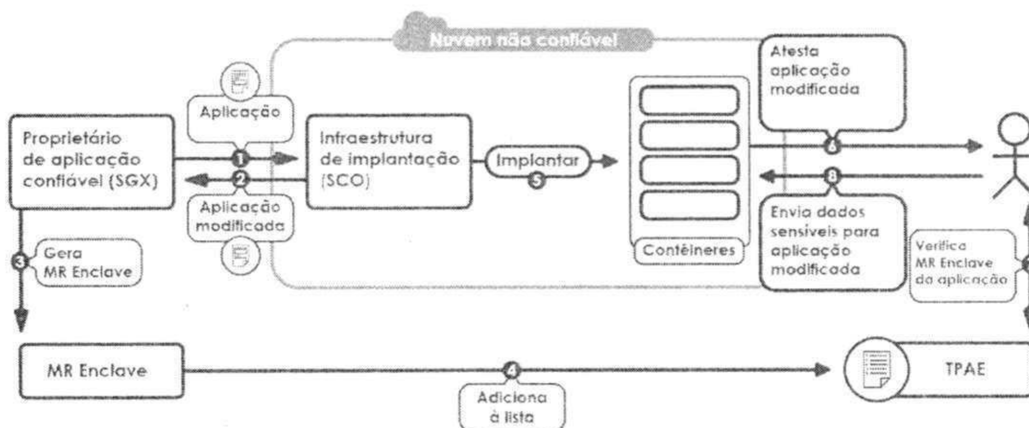


Figura 3.4: Esquema de implantação e utlização de aplicações com a atuação da TPAE e cliente SCO

orquestração de micro-arquiteturas, políticas de *auto scaling* de múltiplos fatores, entre outras. Outras estão ausentes devido a restrições impostas pelas próprias funcionalidades do SCO, como migração de aplicações, tornada impossível pela necessidade de manter conexões entre cliente e enclave.

A fim de servir de base para avaliação dos custos de suas funcionalidades de segurança, o SCO recebeu um arranjo de mecânicas e características relativamente sofisticado, incluindo um processo ágil de instalação, implantação e publicação de aplicações.

Diferentemente de várias soluções de mercado, o SCO possui balanceamento de carga nativo, e não necessita de integração com os componentes de balanceamento padrão da plataforma de nuvem instalada.

Outra vantagem é o suporte a *auto scaling*, uma funcionalidade complexa que requer módulos de monitoramento e uma gerência completa dos estados dos serviços executando, além de ser uma funcionalidade importante para um dos requisitos mais essenciais dos serviços em nuvem, a capacidade de escalar o serviço com o crescimento ou diminuição da demanda.

Apesar de desejáveis mas ausentes no SCO, um serviço de descoberta, a capacidade de federar nuvens e um serviço próprio de resolução de URL independente da nuvem são funcionalidades que podem ser implementadas facilmente em versões futuras.

A implementação de contêineres seguros em outras *engines* de orquestração de contêineres, por sua vez, é signficativamente mais difícil. Nas soluções comparadas aqui, o acesso a aos *drivers* SGX, a implantação de fluxos de balanceamento de carga customizados e, principalmente as políticas de gerenciamento do ciclo de vida de contêineres se apresentam de difícil implementação devido à arquitetura legada. No Kubernetes, por exemplo, o acesso aos drivers é incompatível com o ciclo de vida do contêiner e, quando possível, requer que o contêiner tenha acesso a todos os *drivers* da máquina, o que incorre em uma grave falha de segurança. Ainda no Kubernetes, não existe controle sobre o ciclo de vida de um contêiner, o que impede a gerência de conexões e *scale down* controlado do SCO. Alterar isso só é possível realizando mudanças profundas no seu complexo *scheduler*, o que acarretaria mudanças em cascata em outras partes da arquitetura.

Tabela 3.1: Comparativo de funcionalidades de soluções de orquestração de contêineres

	SCO	Docker Swarm	Kubernetes
Balanceador de Carga Nativo	✓	✗	✗
Auto Scaling	✓	✗	✓
Serviço de Descoberta	✗	✓	✓
Federação de Nuvem	✗	✗	✓
Serviço de URL's	✗	✓	✓
Contêineres Seguros	✓	✗	✗

Capítulo 4

Avaliação de desempenho do SCO

Assim como o que ocorre em vários outros trabalhos, uma preocupação central no desenvolvimento dos items que compõem este estudo é sua viabilidade quanto a aspectos de desempenho e usabilidade. Em especial, é característico desse tipo de solução a atenção ao desempenho, uma vez que os PSN são frequentemente atrelados a SLA's que requerem níveis exigentes de disponibilidade. 0 Das contribuições deste estudo, o SCO é seguramente o componente de *software* mais complexo e cuja classe, de orquestradores de contêineres, é a sujeita aos mais rígidos requisitos operacionais.

Por isso, este estudo avalia o SCO quanto a seu desempenho em atividades que foram julgadas mais essenciais a um orquestador de contêineres operando em uma nuvem comercial.

Para a escolha do meio de comparação, buscou-se um orquestrador de contêineres comercial capaz de proporcionar a mesma classe de serviços e funcionalidades do SCO, ao menos no que diz respeito às operações básicas esperadas deste tipo de infraestrutura. A ferramenta deveria ser capaz de oferecer balanceamento de carga, *auto scaling* e algum nível de isolamento. Outra característica desejável é uma relativa popularidade na indústria.

O Kubernetes demonstrou atender a estas exigências. Além destes, esta ferramenta dispõe de muitos outros recursos e características sofisticadas, como um nível de abstração de serviços mais complexo e configurável que outras soluções de mercado e inclusive o SCO.

Foi também necessário encontrar aplicações SGX adequadas ao escopo do SCO com uma arquitetura compatível, principalmente pelo uso de SGX. A escolha feita é discutida na seção seguinte.

4.1 Aplicações SGX no SCO

Para testar o SCO buscou-se utilizar aplicações já existentes que se enquadrassem em um perfil que atenda aos seguintes requisitos:

- Se fiar em SGX para assegurar a privacidade de comunicação entre as entidades e proteger dados sensíveis processados em um ambiente não confiável.
- Ser constituído de uma arquitetura distribuída e altamente escalável, isto é, não requerer quaisquer mudanças para permitir o aumento na cardinalidade de elementos de processamento com o aumento da carga.
- Ser altamente conteinerizável, isto é, constituir-se de serviços facilmente isoláveis e replicáveis com poucas dependências.
- Não requerer a manutenção de estados. A mudança na escala efetuada por um orquestrador de contêineres pode remover o contêiner hospedeiro de uma aplicação mantendo um estado significativo.

Poucas aplicações com estas características podem ser encontradas na literatura, em especial que atendam à interseção entre os items 1 e 2. Apesar disso, em [28] pode-se encontrar *Security and Privacy Preserving Data Aggregation* (SPPDA), uma arquitetura de componentes escaláveis, dirigida a um caso de uso pertinente para a utilização de meios profusos de segurança e de simples implantação como uma aplicação conteinerizada.

Esta aplicação provê todas as garantias de segurança propostas na seção 2.2, exceto a número 4, uma adição proporcionada pelo modelo de segurança proposto neste trabalho.

4.1.1 SPPDA: Motivação

SSPDA tem o intuito de oferecer um meio seguro para o problema de quebra de privacidade na aferição de consumo de energia elética automatizada. A aferição automatizada idealizada na literatura [17] prevê um componente eletrônico que substitui os medidores eletromecânicos comuns por medidores inteligentes, os *Smart Meters* (SM's). Este medidores são capazes de fornecer por uma rede ao Provedor de Serviço Elétrico (PSE) o nível corrente de consumo

elétrico em tempo real. O PSE pode utilizar estes dados para gerenciar a demanda de energia, oferecer tarifas horo-sazonais e alertar o usuário de aumentos no consumo.

No entanto, o monitoramento do consumo residencial com alta granularidade provido por um SM acarreta um sério risco à privacidade dos usuários. Foi demonstrado em [33] que com a análise do desenvolvimento da curva de consumo residencial é possível inferir o intervalo de utilização de diversos eletrodomésticos e equipamento médico especial.

Assumindo-o neste modelo de segurança como uma entidade não confiável, o PSE deve ter acesso aos dados somente em um nível mais baixo de granularidade, com o qual a inferência das atividades dos aparelhos domésticos não é possível, mas os serviços desejáveis sejam. SSPDA ataca este problema com agregação de leituras de dados em um ambiente seguro com SGX.

Na organização da arquitetura de SPPDA voltada para a gerência de demanda de energia em uma região, se almeja descobrir, dado um grupo de SM's $S = \{SM_1, SM_2, ..., SM_n\}$, o seu consumo e_{ij} no tempo j em um intervalo de medições tal que $j \in \{1, 2, ..., t\}$, localizados nesta região sem que o PSE tenha acesso aos dados dos SM's individualmente. Para isso, o componente agregador de dados que executa no ambiente não confiável do PSE efetua uma soma $\sum_{i=1}^{|S|} e_{ij}$, que gera um valor único para toda uma região em um intervalo de tempo.

Neste ponto, o problema resume-se a impedir que o PSE tenha acesso aos dados antes da agregação, o que requer garantias de privacidade definidas na subseção 2.2, das quais todas exceto a 4 são providas por SGX sozinho e a quarta pode ser implementada com o modelo de segurança proposto aqui apenas com adição de um código envólucro sobre o código do smartmeter para capturar o MRENCLAVE da aplicação do agregador e submetê-lo à TPAE.

4.1.2 SPPDA: Arquitetura

Como exposto na seção anterior, SPPDA relaciona conjuntos de SM's com agregadores em uma relação de $N : 1$, mas com múltiplos agregadores, o que pode ser expresso como $M \times (N : 1)$, sendo M um número de agregadores.

Na versão de SPPDA utilizada para avaliar o SCO, os SM's são dotados do endereço e porta por onde estabelecem uma conexão TCP com os agregadores e através dela iniciam o processo de atestação SGX com o IAS e extraem o MRENCLAVE da aplicação do quote resultante. A verificação da integridade da aplicação com a TPAE é feita com o MRENCLAVE

extraído e os SM's podem iniciar a corrente de dados.

Para permitir a escalabilidade da comunicação entre smartmeters e agregadores, um barramento público, implementando uma política de publicar/assinar, serve de meio de entrega de dados dos SM's para os agregadores. Nele, um agregador a_{ti} tal que $a_{ti} \in \{a_{t1}, a_{t2}, ..., a_{tn}\}$ assina o tópico t_i, neste caso associado a uma região, pertencente ao conjunto de tópicos $T = \{t_1, t_2, ..., t_n\}$. Os SM's do tipo $s_{t_{ij}} | s_{t_{ij}} \in \{s_{t_{i1}}, s_{t_{i2}}, ...s_{t_{in}}\}$, por sua vez, publicam no tópico t_i.

Para garantir a privacidade de comunicação, os dados publicados nos tópicos do barramento público são criptografados utilizando as chaves simétricas estabelecidas entre um SM e o agregador por ele atestado na fase de troca de chaves de Diffie-Hellman. Nota-se neste ponto o problema em que mais de um agregador é necessário para consumir os dados de um mesmo tópico. Neste caso, um agregador a_1 atestado por um SM s_{1,t_1} pode realizar a leitura de uma medição publicada no tópico t_1 por um SM s_{2,t_1}, que realiza atestação com a_2, e portanto criptografada com a chave combinada com este agregador e ilegível para a_1. Para contornar este problema, um identificador do agregador atestado pelo SM é adicionado à medição, tornando a mensagem escrita uma 2-upla $m = (i_n, k_{i_n}(d))$, tal que i_n é um identificador do agregador atestado n e $k_n(d)$ são os dados da leitura d criptografados com a chave simétrica k combinada com o agregador n.

Este identificador é utilizado pelos agregadores que, ao realizar uma leitura na fila de mensagens escritas pelos SM's em um tópico do barramento público, verificam se a mensagem contém seu identificador, ignorando-a caso contrário ou elegendo-a para agregação em caso positivo.

4.2 Modelagem Experimental

Devido a seu estado de engenharia superior, sua escolha de tecnologias mais inclinada à eficiência e sua maior maturidade como plataforma comercial, as expectativas neste estudo são de que o desempenho do Kubernetes deva ser visivelmente superior, mesmo levando em conta os custos adicionais das operações relacionadas ao modelo de segurança e do SGX.

Apesar disso, espera-se estabelecer com a comparação com o SCO limites superiores do custo em desempenho (generalizável para custo financeiro) de implementar o modelo de

segurança proposto. A comparação de custos agregados proposta aqui pode ser expressa por:

$$C_s + C_e + C_{a_1} = \alpha \times C_{a_2} \tag{4.1}$$

Em que C_s é o custo total das operações relacionadas a segurança, C_e as sobrecargas causados pela maior sofisticação na engenharia e $C_{a_1} e C_{a_2}$ os custos intríssecos à execução das aplicações.

A variável objetivo buscada α relaciona o custo da segurança e sobrecarga de engenharia das duas ferramentas e o custo de execução de operações da própria aplicação. Desta maneira, α pode ser interpretado como a razão que estabelece o limite superior do custo da segurança com SCO.

4.2.1 Avaliação de Desempenho de *auto scaling*

Na indústria, uma das funcionalidade mais desejáveis de uma ferramenta de provisionamento de infraestrutura é sua habilidade de instanciar novos recursos na velocidade requerida pelo cliente, devido às consequências negativas de indisponibilidade e resultantes quebras de SLA's.

Para mensurar este aspecto, foram adicionados novos contêineres a um serviço às unidades experimentais e calculou-se o tempo em que se tornaram disponíveis.

A fim de melhor emular o comportamento apresentado por estas ferramentas em um ambiente real, serão exploradas as suas capacidades de *auto scaling*. No contexto da aplicação de teste, apresentada no capítulo anterior, esta operação consiste em criar uma carga de leituras originárias no módulo de simulação de *smartmeters* em uma frequência alta o bastante a ponto de fazer que os agregadores extrapolem os limites de CPU estabelecidos nas configurações de *auto scaling* das ferramentas. Isso deve provocar a instanciação de um novo contêiner pelos orquestradores. Então, sendo T_0 o momento do aumento da carga e T_f o momento em que o novo contêiner está disponível, o tempo total T pode ser descrito como:

$$T = T_f - T_0 \tag{4.2}$$

Internamente, este tempo pode ser descrito em tarefas como Δ_l, o tempo decorrido para que o módulo de monitoramento acione o módulo de gerenciamento e Δ_i, o tempo de ins-

tanciação do contêiner.

$$T = \Delta_l + \Delta_i \tag{4.3}$$

Se denotarmos de T_{teste} e $T_{controle}$ estes tempos para o SCO e Kubernetes, respectivamente, teríamos:

$$T_{teste} = \alpha_1 \times T_{controle} \tag{4.4}$$

4.2.2 Avaliação de Desempenho de Instanciação de Serviços

O segundo aspecto destas ferramentas avaliado é a velocidade de instanciação de um serviço. Esta métrica é de especial importância para serviços que sofrem constantes modificações e por isso precisam ser frequentemente reinstanciados. Esta preocupação tende a se tornar crescentemente popular com a introdução de novas filosofias de práticas de implantação que propõem ciclos de atualização mais curtos e frequentes, como o DevOps.

Este experimento consiste em calcular o tempo desde que um cliente submete uma requisição para a criação de serviço com uma certa quantidade inicial de contêineres até o tempo em que o serviço esteja habilitado. A política utilizada aqui é de considerar um serviço disponível somente quando o último contêiner estiver instanciado.

Define-se então T_0 como o momento do envio da requisição e, sendo para cada contêiner i, tal que $i \in N$, T_{ci} o momento em que este contêiner se torna disponível e definindo ainda n como o número mínimo de contêineres requisitados, temos que o tempo total T pode ser descrito como:

$$T = max(T_{c1}, T_{c2}, ..., T_{cn}) - T_0 \tag{4.5}$$

As tarefas internas executadas neste intervalo podem ser divididas em Δ_r, o intervalo de tempo entre o envio e a chegada da requisição de criação do serviço e, para cada contêiner i, tal que $i \in N$, Δ_{ci} o intervalo de tempo entre o início da instanciação e o momento em que cada contêiner torne-se disponível. Tem-se que estas tarefas podem ser descritas como:

$$max(T_{c1}, T_{c2}, ..., T_{cn}) - T_0 = \Delta_r + max(\Delta_{c1}, \Delta_{c2}, ..., \Delta_{cn}) \tag{4.6}$$

Analogamente ao experimento anterior, denotando-se T_{teste} e $T_{controle}$ como os tempos totais para SCO e Kubernetes respectivamente, teríamos:

$$T_{teste} = \alpha_2 \times T_{controle} \tag{4.7}$$

4.2.3 Avaliação de Desempenho da Aplicação

Além destes aspectos, outra variável que deve apresentar uma forte influência no desempenho global da nuvem no contexto analisado neste trabalho é o impacto de SGX no desempenho da aplicação. Algumas pesquisas já demonstraram aferições de métricas de desempenho de aplicações em um ambiente isolado no SGX [2], mas uma análise melhor inserida no experimento é necessária para constatar o nível deste impacto em relação às demais métricas estudadas aqui.

Neste experimento, compararemos o desempenho da aplicação de controle com a aplicação original de teste que utiliza SGX. Como a aplicação de teste utiliza um barramento de publicar/assinar para a leitura dos dados criptografados (em ambos os casos) e este é um componente de desempenho pouco preditível, opta-se neste caso por estabelecer o tempo inicial T_0 como o momento em que o componente agregador começa a processar a primeira mensagem. O momento final é definido pelo momento em que se conclui a agregação em um resultado de um número N de conjuntos de leituras, como definido no capítulo anterior. Este experimento, portanto, pode ser descrito simplesmente como:

$$T = T_f - T_0 \tag{4.8}$$

Assim como nos experimentos anteriores, denotando-se T_{teste} e $T_{controle}$ como os tempos totais para SCO e Kubernetes respectivamente, temos:

$$T_{teste} = \alpha_3 \times T_{controle} \tag{4.9}$$

4.2.4 Definição das Variáveis Objetivo

É possível, de posse das variáveis α para os aspectos de interesse, construir uma ponderação que relaciona o custo total a essas razões da seguinte maneira:

$$C_{total} = \frac{(w_1\alpha_1 + w_2\alpha_2 + w_3\alpha_3)}{3} \tag{4.10}$$

Sendo w_1, w_2 e w_3 pesos relacionados a cada um dos aspectos expressos por seus respectivos α's tais que.

$$w_1 + w_2 + w_3 = 3 \tag{4.11}$$

Desta maneira, o C_{total} estabelece o custo total da utilização de SCO em relação a Kubernetes em uma proporção direta (um custo total de 2 significa que a utilização de SCO é duas vezes mais cara do que a de Kubernetes).

Esta metodologia de avaliação é generalizável para uma quantidade arbitrária de farores α, o que poderia ser descrito como:

$$C_{total} = \frac{\sum_{i=1}^{n} w_i\alpha_i}{n} \tag{4.12}$$

Com pesos w_i tais que:

$$\sum_{i=1}^{n} w_i = n \tag{4.13}$$

4.3 Aplicação de Controle

Como alguns fatores, já discutidos na seção 2.4, impossibilitam a implantação de aplicações SGX no Kubernetes, a aplicação de teste SPPDA teve de ser alterada para funcionar como em um modelo de segurança de provedor de nuvem confiável. Para isso, as operações de garantia de privacidade de comunicação efetuadas na metodologia de aplicação SGX tiveram de ser substituídas por uma metodologia convencional de criptografia.

Consideramos portanto na unidade experimental de controle um modelo em que o usuário e o desenvolvedor da aplicação têm preocupação com a privacidade da aplicação mas não possuem acesso a recursos de criptografia em *hardware* no PSA.

Essencialmente, a aplicação original teve suas operações de enclave transplantadas para o ambiente inseguro e uma nova mecânica de troca de chaves passou a ser usada. Esta mecânica substitui a atestação SGX e consiste em os *smartmeters* enviarem suas chaves públicas

para os agregadores, que geram chaves simétricas de 32 bits associadas ao *smartmeter* que as enviou e guardadas. Em seguida, estas chaves simétricas são criptografadas com RSA utilizando a chave pública recebida do *smartmeter* e enviadas para serem decriptografadas com as chaves privadas dos respectivos *smartmeters*. Uma vez decriptografadas em seus smarmeters, estas chaves simétricas passam a ser utilizadas com AES para enviar as medições.

O fluxo da aplicação adaptada pode ser observado mais claramente na figura 4.1

4.3.1 Ambiente

Todos os experimentos foram executados em servidores Intel Xeon E3-1280 v6 CPU com 4 núcleos a 3.6 GHz, 8 *hyper-threads* e 8 MB cache.

Para melhor emular o ambiente virtualizado, comum na nuvem, além de dimensionar o ambiente ao tipo de recursos que comumente é utilizado na indústria, os experimentos foram realizados em máquinas virtuais sobre um hipervisor KVM manufaturado especialmente para execução de aplicações com acesso às primitivas SGX. Esta arquitetura executa, por sua vez, sobre uma plataforma de nuvem OpenStack versão Pike.

As imagens utilizadas nestas máquinas virtuais são Ubuntu 16.04.4 LTS também criadas especialmente para permitir acesso às primitivas SGX. A utilização destas imagens não acarreta nenhuma sobrecarga adicional quando executando em modo inseguro, uma vez que as ligações de arquivos de Linux *devices* e outras adaptações necessárias são inteiramente estásticas e o processo *daemon* SGX, necessário para o gerenciamento de enclaves, pode ser encerrado sem prejuízo ao funcionamento da instância.

A rede a que o hardware utilizado neste experimento está conectada é isolada por VLAN, portanto, o comportamento do tráfego de rede esperado é de estabilidade.

Cada uma das 3 máquinas virtuais utilizadas como nós trabalhadores possuem acesso a 1 núcleo real, 1GB RAM e 10GB de acesso a disco, o que constitui um *flavor* típico para hospedar pequenas aplicações na nuvem. No contexto do SSPDA, estes são os nós que executam os agregadores, a parte segura da aplicação e que tem o uso mais intensivo de CPU.

O barramento de Publicar/Assinar [15] de SPPDA e o *manager* do SCO executam em um nó isolado com 2GB RAM e dois núcleos para impedir que a vazão da aplicação seja limitada, em algum momento, por este módulo.

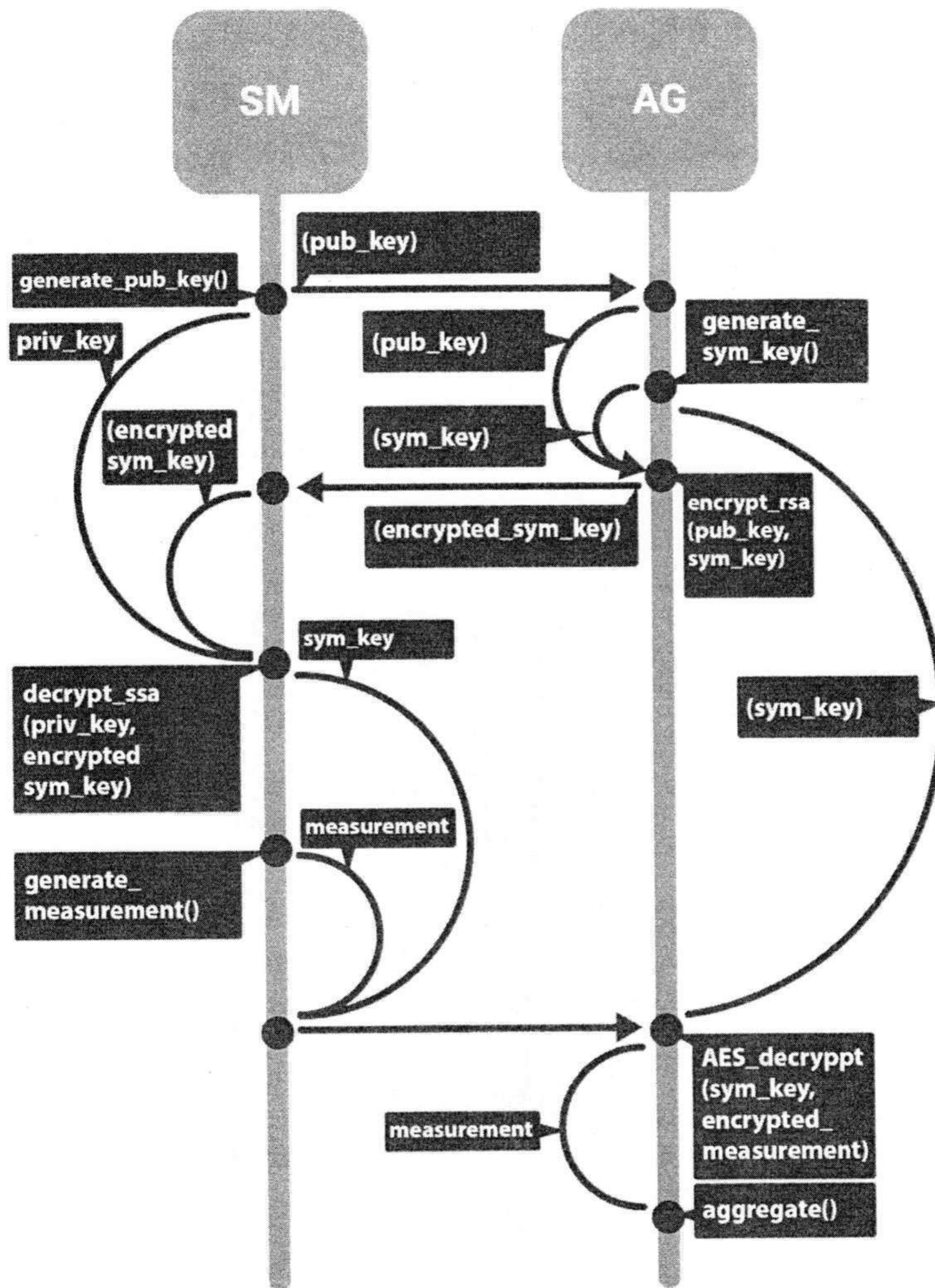


Figura 4.1: Fluxo de troca de chaves da aplicação de controle

Tabela 4.1: Médias de Latência de *auto scaling* (ms)

	SCO	Kubernetes	μ_i
2 instâncias	19627.12	8579.87	2.27
4 instâncias	20611.70	8856.25	2.32
6 instâncias	21963.37	9155.25	2.39
8 instâncias	24052.50	9480.25	2.53

Para garantir o cenário de estresse nos agregadores, o gerador de carga, que desempenha o papel dos *smartmeters*, tem capacidade computacional superior aos nós. Este módulo executa em um nó Intel(R) Core(TM) i7-6700 3.40GHz com 8 *hyperthreads* e 8 GB RAM.

4.4 Resultados e Análise

Os resultados dos experimentos que são intervalos de confiança foram gerados a partir de 30 repetições, a fim de conferir força estatística para o experimento. O nível de confiança utilizado foi de 95%.

4.4.1 Desempenho de *auto scaling*

A figura 4.2 descreve o desempenho de *auto scaling* do mecanismo de orquestração do grupo de controle (Kubernetes) contra a do mecanismo de orquestração do grupo de teste (SCO).

Devido ao bom comportamento dos intervalos de confiança obtidos, as médias parecem demonstrar-se boas estatísticas para explicar o comportamento dos desempenho para cada um dos níveis (2, 4, 6 e 8 contêineres).

O relacionamento entre os desempenhos pode ser então estabelecido como a médias das razões entre os desempenhos médios. Descrevendo formalmente:

$$\alpha_1 = \mu(X) | X = \{\mu_2, \mu_4, \mu_6, \mu_8\} \tag{4.14}$$

Onde $\mu_2, \mu_4, \mu_6, \mu_8$ são as razões médias para cada um dos níveis.

Um μ_i é a razão entre a média $\mu_{teste,i}$ e $\mu_{controle,i}$ para i tal que $i \in \{2, 4, 6, 8\}$

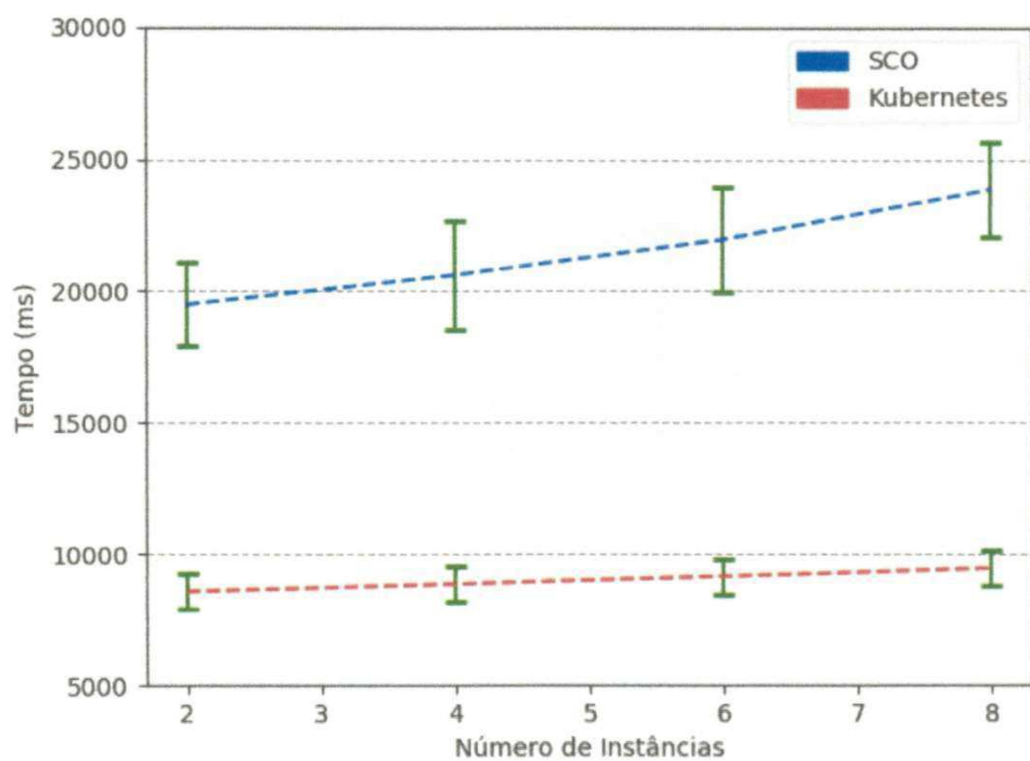


Figura 4.2: Latência de *auto scaling* com Intervalos de confiança - SCO x Kubernetes

Portanto, utilizando os valores de μ_i da tabela 4.1, a diferença média para os níveis analisados pode ser sumarizada em $\alpha_1 = 2.37$. Aplicando-se esta constante na fórmula desenvolvida na equação 4.8, o custo relativo desta diferença, é de $w_1 \times 2.37$.

Uma melhor visualização do comportamento dos valores de latência de *auto scaling* pode ser vista nas figuras A.1, A.2, A.3 e A.4.

4.4.2 Desempenho de Instanciação de Serviços

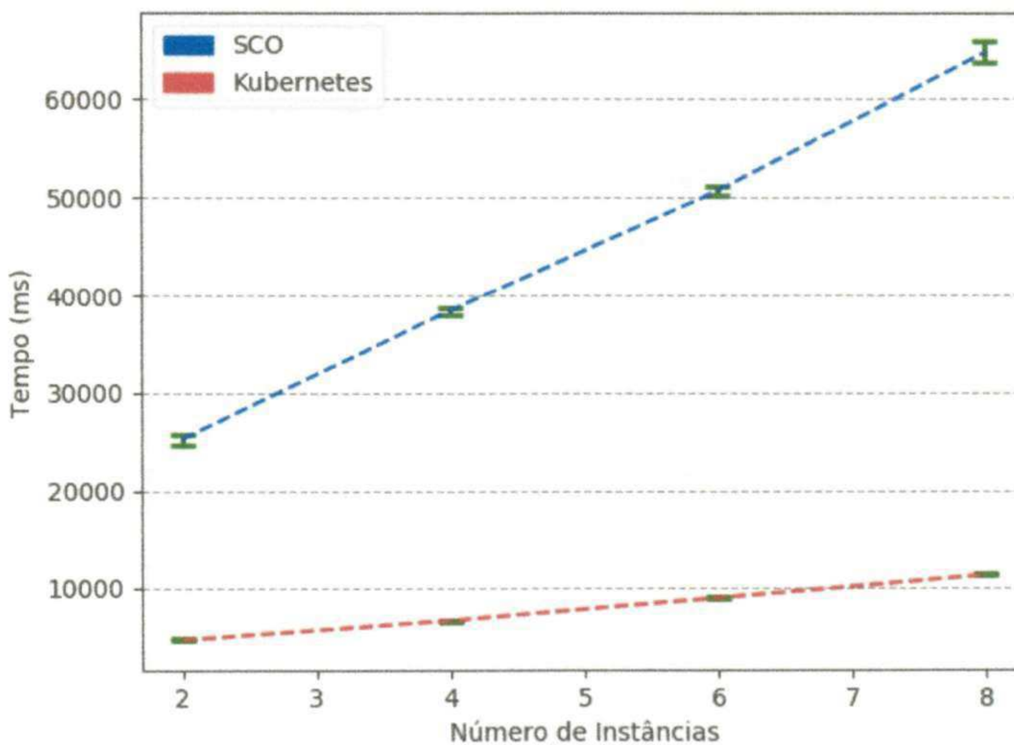


Figura 4.3: Latência de instanciação - SCO x Kubernetes

A figura 4.3, por sua vez descreve o desempenho de instanciação da aplicação para ambos os orquestradores.

Novamente, o bom comportamento dos intervalos de confiança permitem que as médias sejam utilizadas comos estatísticas confiáveis para os níveis analizados, que novamente foram de 2, 4, 6 e 8 contêineres.

Tabela 4.2: Médias de Latência de Instanciação de Serviços (ms)

	SCO	Kubernetes	μ_i
2 instâncias	25304.01	4664.12	5.42
4 instâncias	38435.62	6607.62	5.81
6 instâncias	50652.25	8908.62	5.68
8 instâncias	64776.75	11226.25	5.77

Assim como no experimento anterior, os desempenhos podem ser relacionados como a médias das razões entre os desempenhos médios.

$$\alpha_2 = \mu(X) | X = \{\mu_2, \mu_4, \mu_6, \mu_8\} \tag{4.15}$$

Onde $\mu_2, \mu_4, \mu_6, \mu_8$ são as razões médias para cada um dos níveis. A descrição destas razões segue a mesma metodologia do experimento de desempenho de *auto scaling*.

Para o desempenho de instanciação de serviços, a diferença média para os níveis analisados, utilizando-se os dados de μ_i na tabela 4.2, pode ser sumarizada em $\alpha_2 = 5.67$. Aplicando-se este valor na fórmula desenvolvida na equação 4.8, o custo relativo desta diferença é de $w_2 \times 5.67$.

Uma melhor visualização do comportamento da latência de instanciação pode ser vista nas figuras A.5, A.6, A.7 e A.8.

4.4.3 Avaliação de Desempenho da Aplicação

Na figura A.9, pode-se observar a comparação do desempenho da aplicação de teste contra a aplicação de controle. Como discutido na subseção 4.1.3, este valor corresponde ao tempo necessário para fazer o processo de atestação SGX, no caso da aplicação de teste, e para a troca completa de chaves na aplicação de controle.

É possível observar neste *boxplot* um comportamento bastante previsível na latência apresentada pela aplicação de teste. Já aplicação de controle teve um comportamento muito pouco previsível. Na tabela 4.3 pode-se observar as estatísticas que descrevem a distribuição dos valores de latência para estas aplicações.

Com o apoio das estatísticas apresentadas na 4.3, podemos considerar, com o grau de

Tabela 4.3: Estatísticas de Atestação das aplicações

	Média (ms)	Intervalo de Confiança (95%)	Desvio Padrão
SCO	12977.45	[12775.65, 13179.25]	2432.78
Kubernetes	9722.27	[8008.13, 11436.41]	286.40

certeza estabelecido, que as médias devem ser boas estatísticas para analisar os custos da utilização de uma aplicação SGX em relação a metodologias convencionais de comunicação segura, uma vez que os intervalos de confiança para o grau de certeza de 95% não se intercedem.

Dessa forma, a relação de custo entre as aplicações de teste e controle pode ser descrita como:

$$\alpha_3 = \mu_r | \mu_r = \frac{\mu_{teste}}{\mu_{controle}} \quad (4.16)$$

Onde μ_{teste} e $\mu_{controle}$ são as médias de latência para cada uma das aplicações.

Este desempenho para os valores de latência obtidos pode ser sumarizado em $\alpha_3 = 1.33$.

Aplicando-se este valor na fórmula desenvolvida na equação 4.8, o custo relativo desta diferença, é de $w_3 \times 1.33$.

Devido ao baixo desvio padrão apresentado pela aplicação de teste quando comparado à magnitude dos valores em milissegundos, a visualização do comportamento das latências foi prejudicada. Na figura A.10, o *boxplot* está disponível uma escala mais adequada.

Análise do modelo de custo

Utilizando-se os valores obtidos em α_1, α_2 e α_3, é possível simular perfis de usuários de um PSN e assim estimar os custos da utilização de uma plataforma de provisionamento de contêineres segura, como o SCO.

Aqui é estabelecido um perfil 1, em que o usuário dá grande importância à capacidade de atendimento e importância média ao desempenho da aplicação e ao tempo de instanciação do serviço. Este perfil é o adotado por serviços em que os custos do não atendimento a uma requisição são proibitivamente altos, mas o desempenho baixo é tolerado pelos clientes da aplicação, como em *e-commerces*.

A título de ilustração, podemos denotar a modelagem dos pesos neste caso como:

$$w_1 = w_2 + w_3 \tag{4.17}$$

E se, como proposto na equação 4.11:

$$w_1 + w_2 + w_3 = 3 \tag{4.18}$$

Segue que:

$$w_1 = 1.5, w_2 = w_3 = 0.75 \tag{4.19}$$

Nesta conjectura, a variável objetivo Custo Total é expressa como:

$$C_{total} = \frac{(1.5 \times 2.37 + 0.75 \times 5.67 + 0.75 \times 1.33)}{3} \tag{4.20}$$

$$C_{total} = 2.93 \tag{4.21}$$

Um perfil 2 pode ser previsto como de um usuário de PSN cujos utilizadores têm pouca tolerância a baixo desempenho da aplicação, mas o não atendimento de uma requisição em um instante de alta utilização do sistema é aceitável, como em jogos on-line. Neste perfil, atualizações do sistema ocorrem raramente e podem ser programadas, diminuindo a importância da performance deste fator.

Seria possível modelar este perfil como:

$$w_2 = w_1 + 2w_3 \tag{4.22}$$

De maneira análoga ao demonstrado no perfil 1:

$$w_2 = 2, w_1 = 1, w_3 = 0.5 \tag{4.23}$$

Nesta conjectura, a variável objetivo Custo Total é expressa como:

$$C_{total} = \frac{(2 \times 2.37 + 1 \times 5.67 + 0.5 \times 1.33)}{3} \tag{4.24}$$

$$C_{total} = 3.69 \tag{4.25}$$

Outros perfis podem ser empregados, ficando a cargo do proprietário da aplicação metrificar quais aspectos devem ser priorizados através do controle de pesos para seus serviços específicos de acordo com suas necessidades, e assim quantificar os custos de uma escolha por segurança.

Capítulo 5

Conclusões e Trabalhos Futuros

Esse trabalho demonstrou uma alternativa de modelo de segurança adequada a uma nova realidade de infraestrutura de TI, que adquiriu protagonismo ao longo da década de 2010. Esta realidade se caracteriza pela utilização de recursos computacionais remotos ao proprietário da aplicação, com a organização das aplicações de terceiros em contêineres compartilhando estes recursos.

Desde o início de sua popularização, o crescimento deste modelo de infraestrutura de TI acompanha um receio por partes dos utilizadores quanto à segurança dos dados de seus clientes. Uma necessidade que, apesar de ter sido abordada em vários trabalhos, não foi devidamente atendida devido à incompletude dos sistemas propostos.

A alternativa demonstrada aqui se baseia somente no relacionamento de confiança estabelecida entre o proprietário da aplicação e seu usuário (que requer a participação de terceiros e a utilização de software especializado pelo cliente), nas utilidades oferecidas por uma robusta plataforma de computação segura, o SGX, e uma *engine* de orquestração de contêineres capaz de lidar com as especificidades do fluxo de utilização desta plataforma, o SCO.

No modelo de segurança proposto neste trabalho, o usuário pode confiar na integridade da aplicação porque, uma vez que o cliente da aplicação faça a atestação nativa do SGX, um serviço oferecido por um terceiro (TPAE) pode ser utilizado para verificar, através da comparação de uma chave que identifica unicamente a aplicação, se ela foi adulterada na nuvem.

As dificuldades de implantação de uma aplicação conteinerizada com capacidade de

acessar os recursos SGX e de garantir o fluxo de vida e balanceamento de carga dos contêineres são resolvidas pela *engine* de orquestração de contêineres SCO.

Depois é demonstrado qual é o custo da utilização do SCO em relação ao padrão de mercado em *engines* de orquestração de contêineres, aqui representado pelo Kubernetes, líder de mercado no segmento. Este custo é avaliado em vários aspectos e é efetuado um estudo que permite ao utilizador do serviço de hospedagem na nuvem quantifica-lo.

O estudo de custo demonstra as razões de custo da utilização do SCO para latência de *auto scaling*, latência de instanciação de um serviço, e latência de atestação da aplicação. Estas variáveis, denotadas de α_1, α_2 e α_3 orbitam, respectivamente, os valores de 2.37, 5.67 e 1.33.

O estudo de perfis de usuário permite também demonstrar como esses valores podem variar com as necessidades do usuário. Foi demonstrando que o aumento de custo com a utilização da infraestrutura segura para um usuário típico é em muitos casos proibitivo e em outros aceitável.

Com estas contribuições, este trabalho permite aprimorar o estado da arte no que diz respeito à computação segura na nuvem. Outras alternativas promissoras, como o SCONE, não levam em conta os aspectos complexos da orquestração de contêineres seguros, abordados com minúncia no SCO, ou não estão adequadas aos mais altos níveis de exigência de segurança, como Overshadow e outros.

5.1 Trabalhos Futuros

O SCO é uma alternativa viável para a orquestração e execução de código nativo SGX, isto é, que não é recompilado para se tornar compatível, em uma infraestrutura de nuvem genérica. No entanto as ressalvas de que melhorias de desempenho e de funcionalidade oferecida para esta plataforma são válidas.

Intrissecamente, o SCO tem problemas de desempenho por ter sido escrito em Python, linguagem de programção conhecida pelo desempenho inconsistente e comumente inferior, além de mecanismos de orquestração pouco otimizados em muitos aspectos.

Além disso, várias funcionalidades, como monitoramento de rede e memória como métricas de *auto scaling*, definição de diferentes políticas de balanceamento de carga, criação

simplificada de microsserviços, redes overlay entre outras são necessidades ainda não resolvidas pelo SCO.

Por isso, há um grande desafio de refíno e expansão de funcionalidades para que o SCO alcance as primeiras versões de produção. Este se caracteriza como o mais direto trabalho futuro derivado deste.

A integração com ambientes de nuvem é também um aspecto desejável e compreendido por outras *engines* de orquestração de contêineres.

No âmbito conceitual, expandir o modelo de segurança proposto neste trabalho se faz possível, por exemplo, criando maneiras de impedir um ator malicioso no PSN de ter acesso ao código fonte, de onde vulnerabilidades no código do enclave SGX podem ser descobertas e exploradas.

O estudo de custo pode também ser expandido com a introdução de novas variáveis relevantes aos proprietários de aplicações, além do tempo de resposta de *auto scaling*, tempo de criação de um serviço e performance da aplicação. A hierarquização e, idealmente, o estabelecimento de valores característicos de diferentes perfis de utilizador de serviço de nuvem são os passos mais desejáveis nesta frente.

Bibliografia

[1] Docker docs. https://www.docker.io/, 2018. [Online no dia 9 de Janeiro de 2018].

[2] Metodologia de atestação sgx. https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing, 2018. [Online no dia 9 de Janeiro de 2018].

[3] Hussain AlJahdali, Abdulaziz Albatli, Peter Garraghan, Paul Townend, Lydia Lau, and Jie Xu. Multi-tenancy in cloud computing. In *Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium on*, pages 344–351. IEEE, 2014.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. Technical report, Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009.

[5] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, et al. Scone: Secure linux containers with intel sgx. In *OSDI*, pages 689–703, 2016.

[6] Sumeet Bajaj and Radu Sion. Correctdb: Sql engine with practical query authentication. *Proceedings of the VLDB Endowment*, 6(7):529–540, 2013.

[7] Sumeet Bajaj and Radu Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *IEEE Transactions on Knowledge and Data Engineering*, 26(3):752–765, 2014.

[8] Sushil Bhardwaj, Leena Jain, and Sandeep Jain. Cloud computing: A study of infrastructure as a service (iaas). *International Journal of engineering and information Technology*, 2(1):60–63, 2010.

[9] Dan Boneh. *The Decision Diffie-Hellman problem*. 1998.

[10] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009.

[11] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 2–13. ACM, 2008.

[12] Yueqiang Cheng, Xuhua Ding, and Robert H. Deng. Efficient virtualization-based application protection against untrusted operating system. 2015.

[13] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[14] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual ghost: Protecting applications from hostile operating systems. *ACM SIGPLAN Notices*, 49(4):81–96, 2014.

[15] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[16] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[17] Will Gans, Anna Alberini, and Alberto Longo. Smart meter devices and the effect of feedback on residential electricity consumption: Evidence from a natural experiment in northern ireland. *Energy Economics*, 36:729–743, 2013.

[18] Peter Géczy, Noriaki Izumi, and Koiti Hasida. Cloudsourcing: managing cloud adoption. 2011.

[19] Owen S Hofmann, Sangman Kim, Alan M Dunn, Michael Z Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 265–278. ACM, 2013.

[20] Ann Mary Joy. Performance comparison between linux containers and virtual machines. In *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in*, pages 342–346. IEEE, 2015.

[21] Yanlin Li, Jonathan M McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. Minibox: A two-way sandbox for x86 native code. In *USENIX Annual Technical Conference*, pages 409–420, 2014.

[22] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 143–158. IEEE, 2010.

[23] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 315–328. ACM, 2008.

[24] Marcus Peinado, Yuqun Chen, Paul England, and John Manferdelli. Ngscb: A trusted open system. In *Australasian Conference on Information Security and Privacy*, pages 86–97. Springer, 2004.

[25] Benjamin Quétier, Vincent Neri, and Franck Cappello. Scalability comparison of four host virtualization tools. *Journal of Grid Computing*, 5(1):83–98, 2007.

[26] Mendel Rosenblum. The reincarnation of virtual machines. *Queue*, 2(5):34, 2004.

[27] Dries Schellekens, Brecht Wyseur, and Bart Preneel. Remote attestation on legacy operating systems with trusted platform modules. *Electronic Notes in Theoretical Computer Science*, 197(1):59–72, 2008.

[28] Leandro Ventura Silva, Rodolfo Marinho, Jose Luis Vivas, and Andrey Brito. Security and privacy preserving data aggregation in cloud computing. In *Proceedings of the Symposium on Applied Computing*, SAC '17, 2017.

[29] Borja Sotomayor, Rubén S Montero, Ignacio M Llorente, and Ian Foster. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet computing*, 13(5), 2009.

[30] S Swathi, S Saravanan, and V Venkatachalam. Preemptive virtual machine scheduling using cloudsim tool. *Int. J. Adv. Eng*, 1(3):323–327, 2015.

[31] Hassan Takabi, James BD Joshi, and Gail-Joon Ahn. Security and privacy challenges in cloud computing environments. *IEEE Security & Privacy*, 8(6):24–31, 2010.

[32] Andrea Tosatto, Pietro Ruiu, and Antonio Attanasio. Container-based orchestration in cloud: State of the art and challenges. In *Ninth International Conference on Complex, Intelligent, and Software Intensive Systems, CISIS 2015, Santa Catarina, Brazil, July 8-10, 2015*, pages 70–75, 2015.

[33] Markus Weiss, Adrian Helfenstein, Friedemann Mattern, and Thorsten Staake. Leveraging smart meter data to recognize home appliances. In *Pervasive Computing and Communications (PerCom), 2012 IEEE International Conference on*, pages 190–197. IEEE, 2012.

[34] Hanqian Wu, Yi Ding, Chuck Winer, and Li Yao. Network security for virtual machine in cloud computing. In *Computer Sciences and Convergence Information Technology (ICCIT), 2010 5th International Conference on*, pages 18–21. IEEE, 2010.

[35] Jisoo Yang and Kang G Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 71–80. ACM, 2008.

[36] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 203–216. ACM, 2011.

[37] Dimitrios Zissis and Dimitrios Lekkas. Addressing cloud computing security issues. *Future Generation computer systems*, 28(3):583–592, 2012.

Apêndice A

Visualizações dos Resultados dos Experimentos

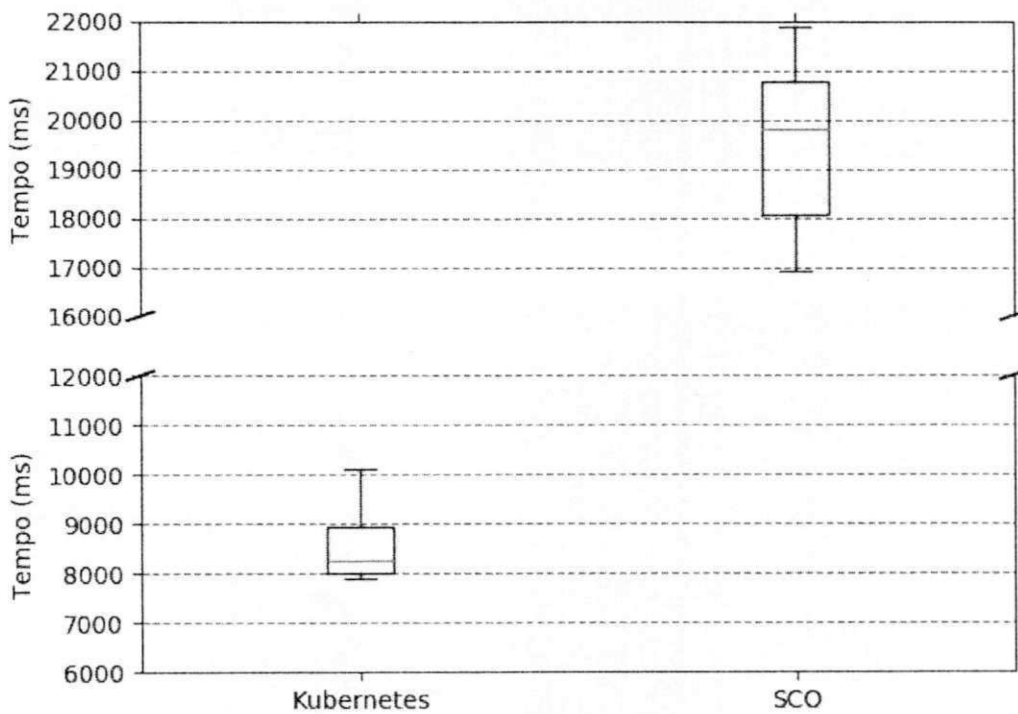


Figura A.1: Latência de *auto scaling* com 2 instâncias - SCO x Kubernetes

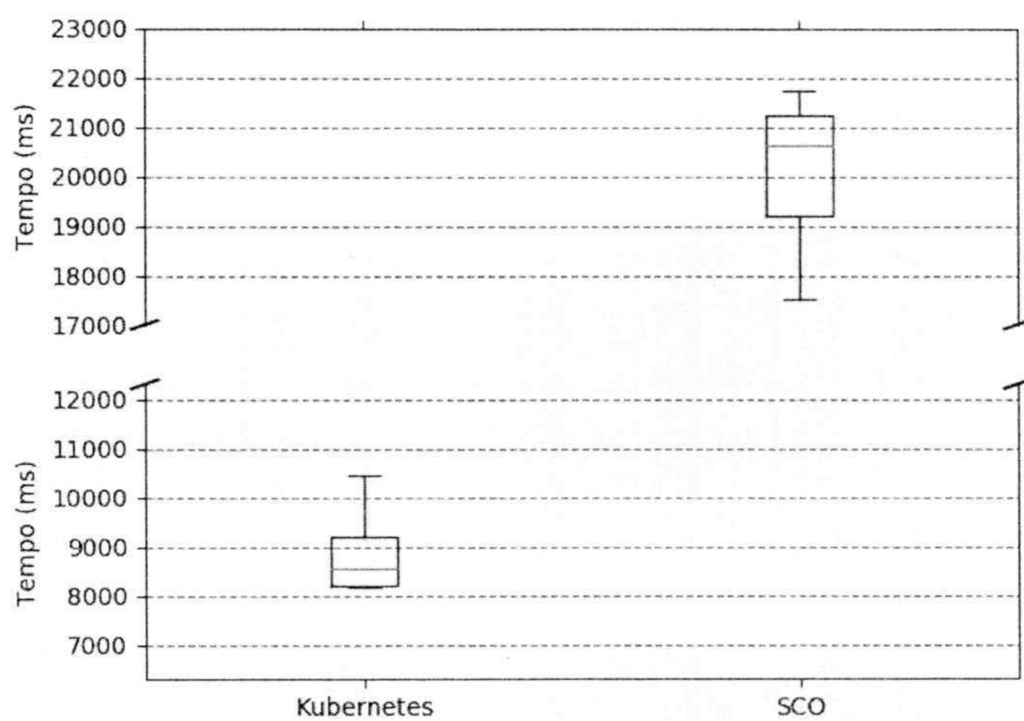


Figura A.2: Latência de *auto scaling* com 4 instâncias - SCO x Kubernetes

UFCG/BIBLIOTECA/BC

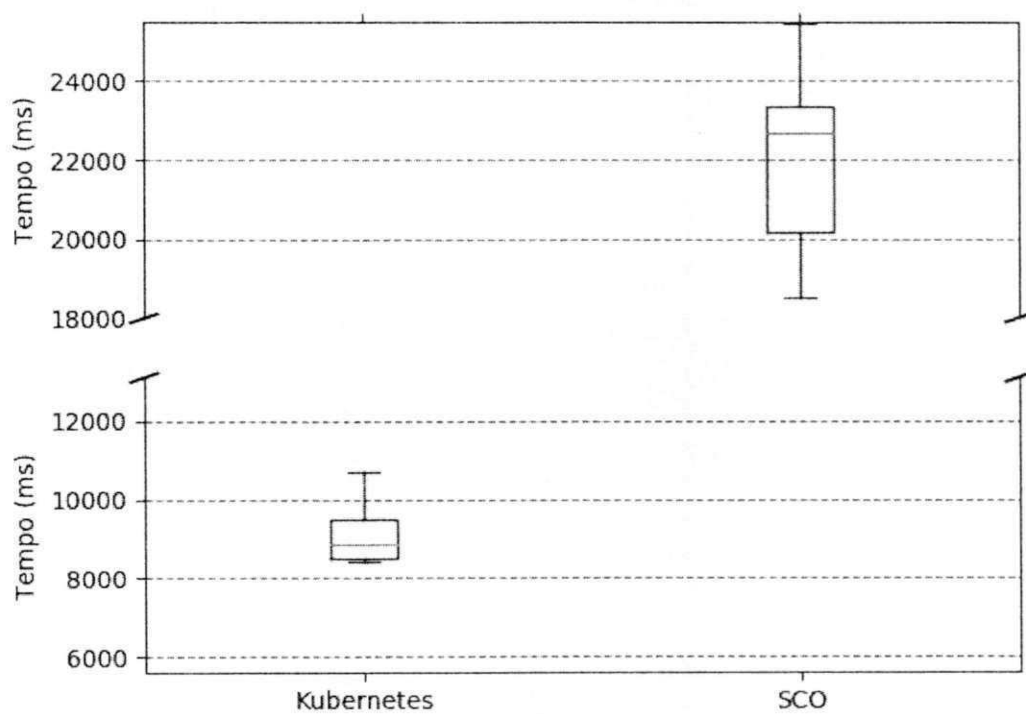


Figura A.3: Latência de *auto scaling* com 6 instâncias - SCO x Kubernetes

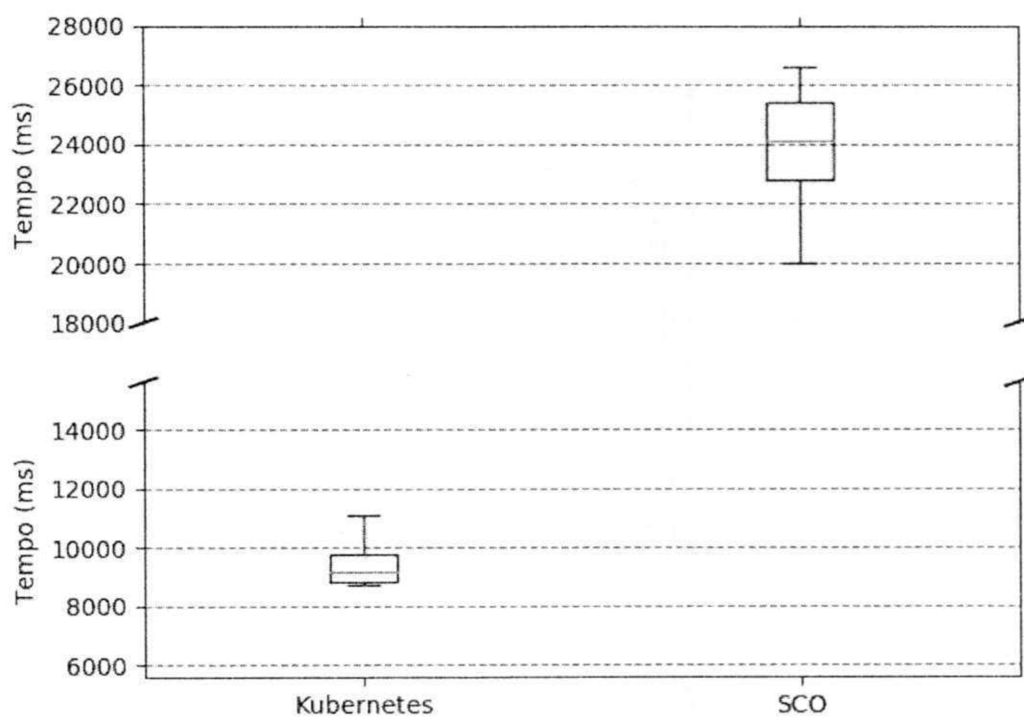


Figura A.4: Latência de *auto scaling* com 8 instâncias - SCO x Kubernetes

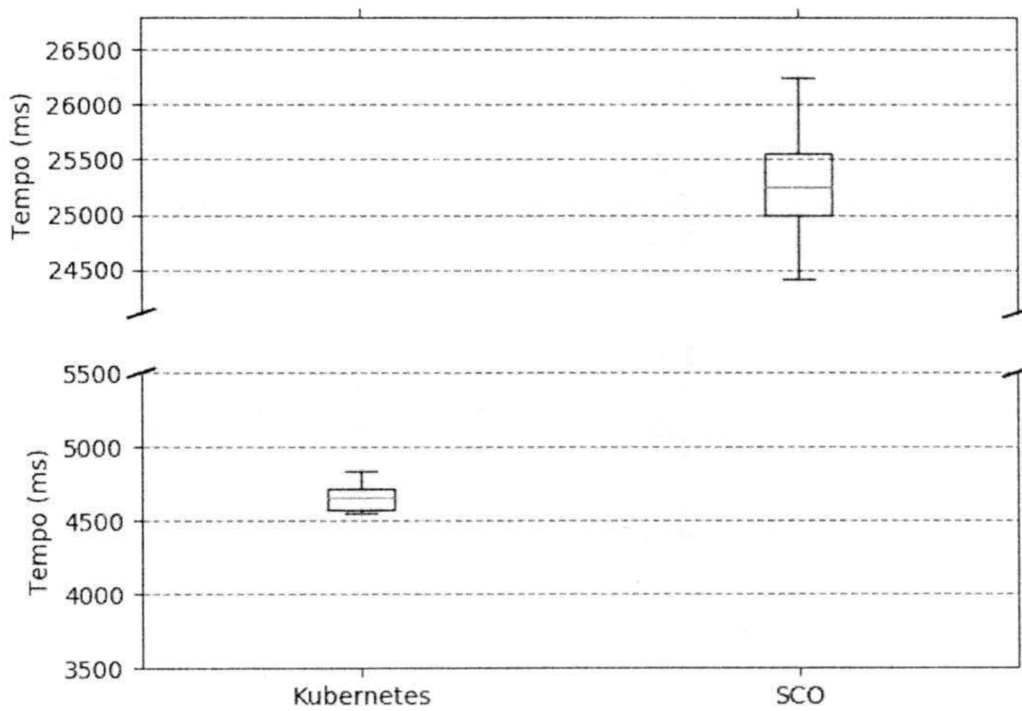


Figura A.5: Latência de Instanciação com 2 instâncias - SCO x Kubernetes

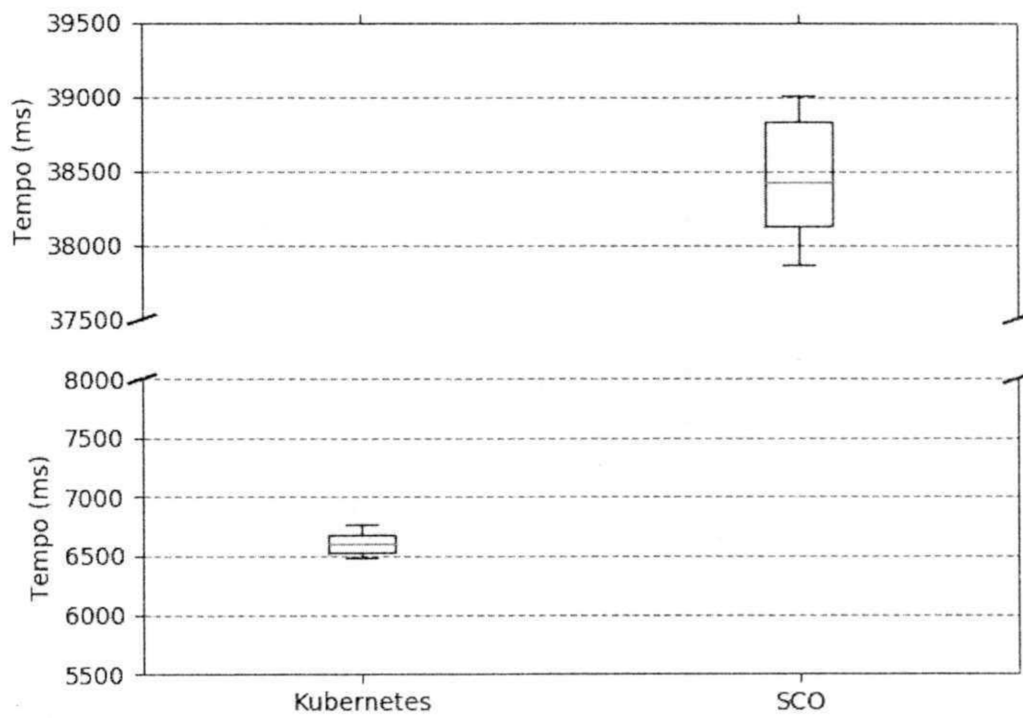


Figura A.6: Latência de Instanciação com 4 instâncias - SCO x Kubernetes

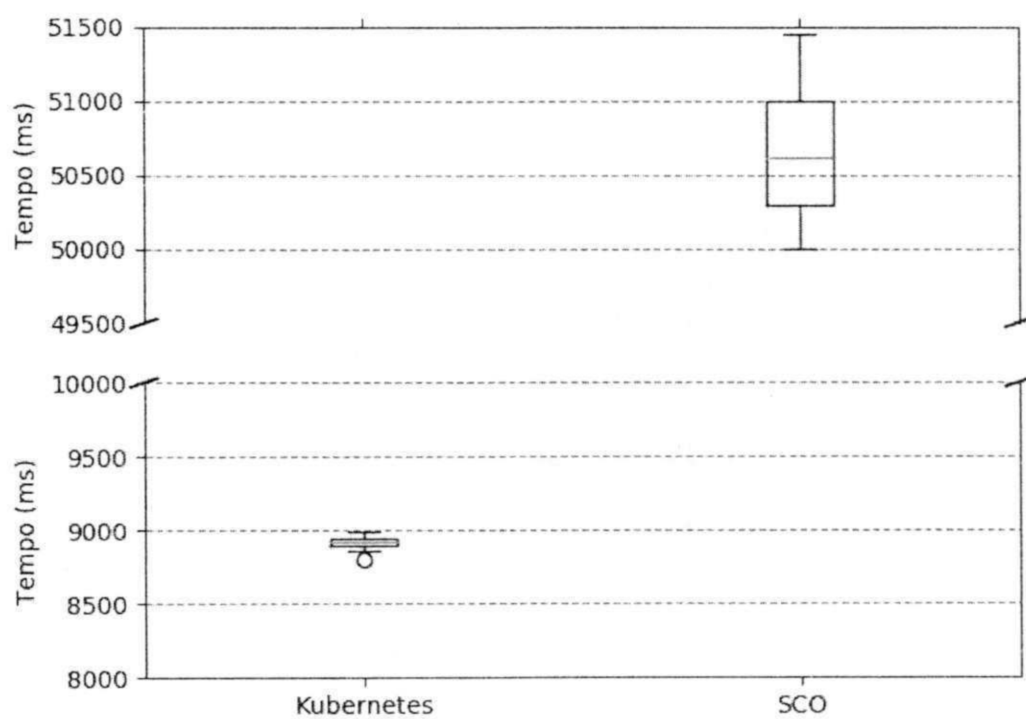


Figura A.7: Latência de Instanciação com 6 instâncias - SCO x Kubernetes

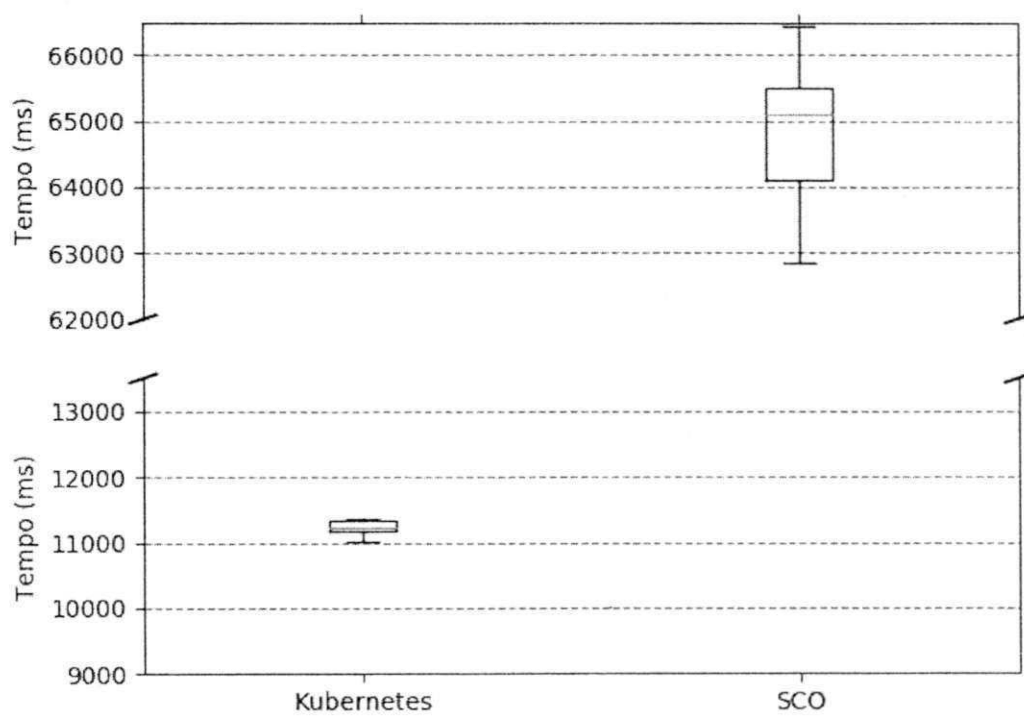


Figura A.8: Latência de Instanciação com 8 instâncias - SCO x Kubernetes

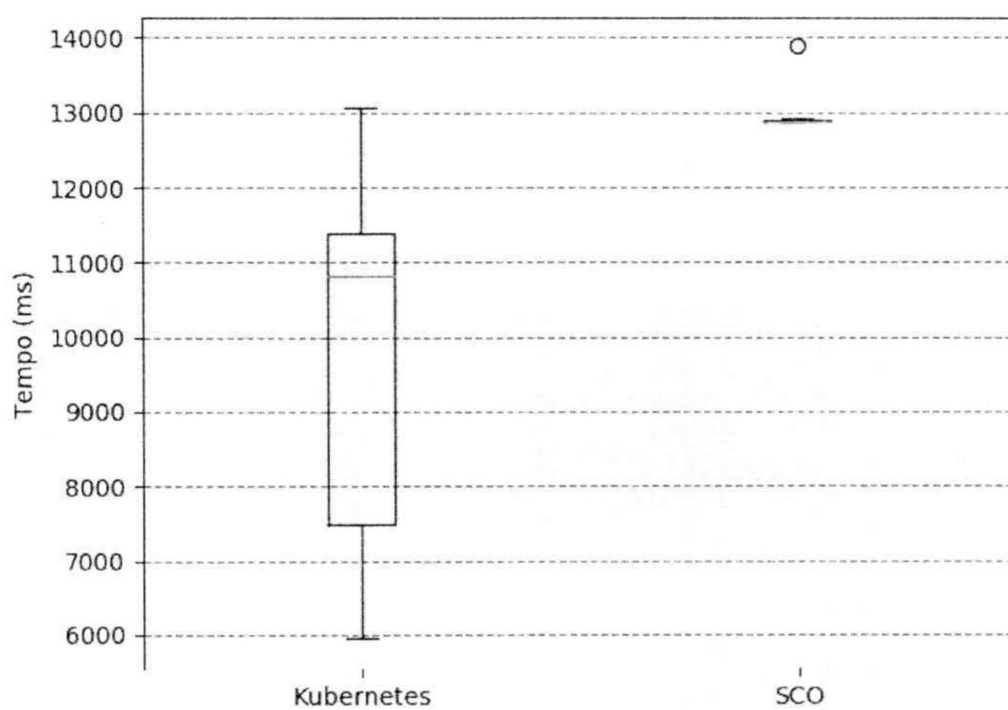


Figura A.9: Latência de atestação das aplicações de teste e controle

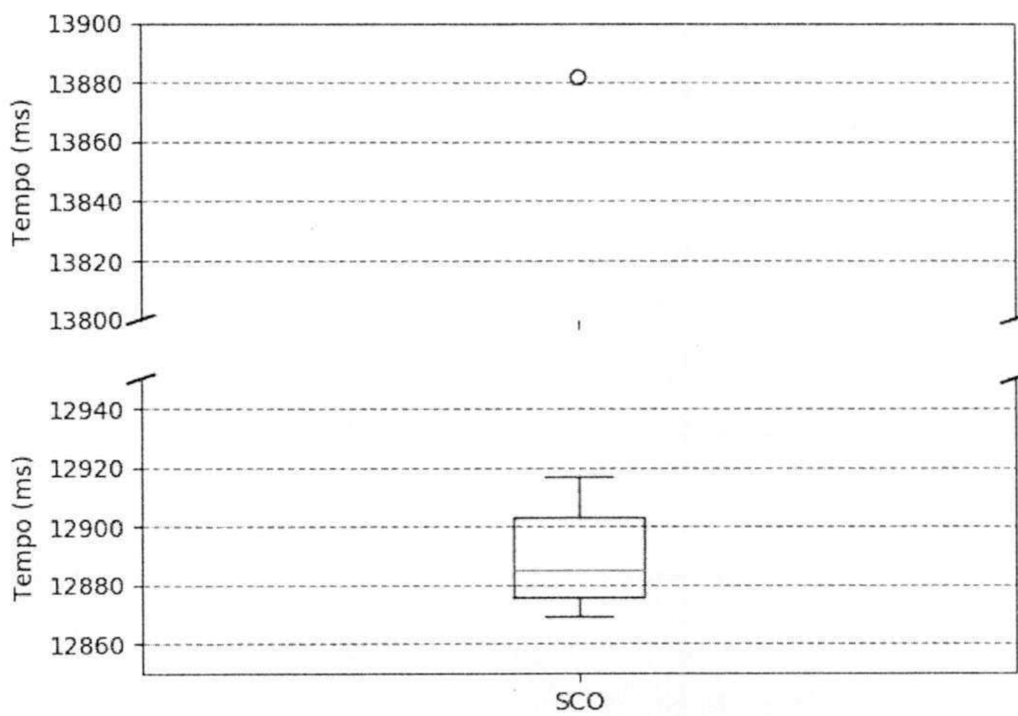


Figura A.10: Latência de atestação da aplicação de teste

Apêndice B

Daemon de Monitoramento

```
import definition
import app
import urllib
import time
import os.path
import json
import urllib2
import poster
import copy
from docker import docker



UPPER_THRESHOLD = 80
LOWER_THRESHOLD = 25



def stats_string_to_dict(stats_lines):
    stats_dict = {}
    for line in stats_lines:
```


UFCG/BIBLIOTECA/BC

```
        line = line.strip('\n')
        if len(line) > 2:
            key_value = line.split(": ")
            stats_dict[key_value[0][1:]] = key_value[1]
    return stats_dict


def get_cpu_stats(cluster_id):
    available_nodes = app.get_available_nodes()
    stats_dict = {}
    stats_lines = []
    #GETTING DOCKER CPU STATS FROM ALL INSTANCES IN ALL NODES
    for node in available_nodes:
        if node.get_ip() != "localhost":
            url = 'http://' + node.get_ip() + \
               ":5001/cpu_stats/" + cluster_id
            node_stats_table = urllib2.urlopen(url).read()
            node_stats_lines = node_stats_table.split('\n')
            stats_dict.update(stats_string_to_dict(node_stats_lines))
        else:
            node_stats_table = docker.stats_cpu()
            node_stats_lines = node_stats_table.split('\n')
            #TURNING STATS STRING LINES
            #INTO {INSTANCE_ID -> CPU_STAT} DICTIONARY
            all_stats_dict = stats_string_to_dict(node_stats_lines)
            cluster_instances_ids = docker. \
               get_instance_ids_by_id(cluster_id)

            #REMOVING STATS FROM INSTANCES THAT ARE
            #NOT FROM THE DESIRED CLUSTER
            for key in all_stats_dict:
```

```
                if key in cluster_instances_ids:
                    stats_dict[key] = all_stats_dict[key]
    print "[INFO] current instances for " + \
       cluster_id + " are: " + str(stats_dict)
    stats = stats_dict.values()
    index = 0
    #REMOVING PERCENTAGE CHARACTER FROM CPU STATS
    #VALUES AND TURNING THEM INTO INTS
    for stat in stats:
        stat = stat[:-2]
        stat = float(stat)
        stats[index] = stat
        index += 1
    return stats

def add_instance(image_id, mem, port):
    #SENDING REQUEST AN "ADD NEW CONTAINER INSTANCE TO THE MANAGER
    url = 'http://localhost:5001/add'
    form_data = {'image_id' : image_id, 'network' : image_id, \
       'mem' : str(mem), 'port' : port}
    params = urllib.urlencode(form_data)
    response = urllib2.urlopen(url, params)
    result = response
    return response

def remove_instance(image_id):
    url = 'http://localhost:5001/remove/' + image_id
    response = urllib2.urlopen(url, data="")
    result = response
    return response
```

```
#EVERY 4 SECONDS, VERIFY CURRENTLY CREATED CLUSTERS MEAN CPU
#USAGE STATS. IF OVER THRESHOLD, ADD NEW INSTANCE TO CLUSTER
stats_mean_record = {}
while True:
    if os.path.isfile((definition.ROOT_PATH + "/data/clusters.json")):
        #Open clusters file. It gets written to in app.py
        #in cluster creation methods
         with open(definition.ROOT_PATH + \
            "/data/clusters.json", 'rw') as clusters_file:
             clusters = json.load(clusters_file)
             clusters_file.seek(0)
             #update instances for each registered cluster
             for cluster in clusters:
                 cluster_id = cluster['cluster_id']
                 mem = cluster['mem']
                 port = cluster['port']
                 min_instances = cluster['instances']
                 #get cpu stats for all containers in this
                 #cluster in all nodes it has been deployed
                 stats = get_cpu_stats(cluster_id)
                 amount_of_instances = len(stats)
                 print "[DEBUG] min_instances is " + str(min_instances)
                 #get the mean cpu consumption of the containers
                 stats_mean = sum(stats)/amount_of_instances
                 #add the last mean cpu consumption to
                 #the history of cpu consumption measurements
                 if not cluster_id in stats_mean_record.keys():
                     stats_mean_record[cluster_id] = []
                 stats_mean_record[cluster_id].append(stats_mean)
                 #remove cpu consumption measure if it is earlier than
                 #three iterations old (should be customizable)
```

```
            if (len(stats_mean_record[cluster_id]) > 3):
                stats_mean_record[cluster_id] = \
                   stats_mean_record[cluster_id][1:]
                stats_ratio = sum(stats_mean_record[cluster_id])
                if stats_ratio > UPPER_THRESHOLD:
                    response = add_instance(cluster_id, mem, port)
                    print response
                elif stats_ratio < LOWER_THRESHOLD and \
                   amount_of_instances > min_instances:
                    response = remove_instance(cluster_id)
        time.sleep(4)
        print "I'm alive"
        clusters_file.close()
else:
    time.sleep(4)
```

UFCG/BIBLIOTECA/BC

Apêndice C

API Docker e exemplos de chamadas

C.1 Docker API - Comandos Docker básicos e utilitários

```
import argparse
import subprocess
import sys

ps_values = {'CONTAINER_ID': 0, 'IMAGE': 1, 'COMMAND': 2, \
  'CREATED': 3, 'STATUS': 4, 'PORTS': 5, 'NAMES': 6}
inspect_get_ip_query = \
   '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
default_volume_path = '/src/volume'
networkID_string_start_position = 14
networkID_string_end_position = -2
IPAddress_string_start_position = 14
IPAddress_string_end_position = -2


#PRIVATE; EXECUTE: runs the bash command built by the api
#functions with the giver arguments
def _execute(args, opts, stdout = False):
    for o in opts:
```

```
        args.append(o)
    result = None
    if stdout:
        result = subprocess.check_output(args)
    else:
        result = not subprocess.call(args)
    return result

#PRIVATE; EXECUTE_ORDERED: same as execute, but the opts arguments
#can be inserted in the desired order
def _execute_ordered(args, opts, opts_order, stdout = False):
    position = 0
    for o in opts:
        args.insert(opts_order[position], o)
        position += 1
    result = None
    if stdout:
        result = subprocess.check_output(args)
    else:
        result = not subprocess.call(args)
    return result

#BUILD: builds an image with the given
#parameters (quiet mode by default)
def build(*opts):
    args = ['docker', 'build', '-q']
    opts = list(opts)
    return _execute(args, opts)

#REMOVE_IMAGE: removes an image with the given parameters.
#Forcefully removes containers running with that image
```

```
def rmi(*opts):
    args = ['docker', 'rmi', '-f']
    opts = list(opts)
    return _execute(args, opts)

#RUN: creates an instance with the given parameters
#and returns its id (removes last \n character)
def run(*opts):
    args = ['docker', 'run']
    opts = list(opts)
    return _execute(args, opts, stdout = True)[:-1]


def run_get_name(*opts):
    args = ['docker', 'run']
    opts = list(opts)
    return _execute(args, opts, stdout = True)

#REMOVE_INSTANCE: removes an instances with the given parameters
def rm(*opts):
    args = ['docker', 'rm']
    opts = list(opts)
    return _execute(args, opts, stdout = True)

#STOP_INSTANCE: stops an instance with the given parameters
def stop(*opts):
    args = ['docker', 'stop']
    opts = list(opts)
    return _execute(args, opts)

#PS: lists all instances, running and exited
```

```
def ps(*opts):
    args = ['docker', 'ps', '-a']
    opts = list(opts)
    return _execute(args, opts, stdout = True)

#IMAGES: Lists all images
def images(*opts):
    args = ['docker', 'images']
    opts = list(opts)
    return _execute(args, opts, stdout = True)

#INSPECT: returns metadata from a container
#with the giver paramenters
def inspect(*opts):
    args = ['docker', 'inspect']
    opts = list(opts)
    return _execute(args, opts, stdout = True)

#CREATE_VOLUME: creates a volume with the given parameters
def create_volume(*opts):
    args = ['docker', 'volume', 'create', '-d', 'local', '--opt', \
    'type=tmpfs', '--opt', 'device=tmpfs', '--opt', '--name']
    opts = list(opts)
    positions = [10, 12]
    return _execute_ordered(args, opts, positions, stdout = True)

#REMOVE_VOLUME: removes a volume with the given parameters
def remove_volume(*opts):
    args = ['docker', 'volume', 'rm']
    opts = list(opts)
    return _execute(args, opts)
```

```
#CP: copies files from instances to containers and vice versa
def cp(*opts):
    args = ['docker', 'cp']
    opts = list(opts)
    return _execute(args, opts)

#CREATE_NETWORK: creates a network with the given parameters
def create_network(*opts):
    args = ['docker', 'network', 'create', '--driver', 'bridge']
    opts = list(opts)
    return _execute(args, opts)

#REMOVE_NETWORK: removes a network with the given parameter
def remove_network(*opts):
    args = ['docker', 'network', 'rm']
    opts = list(opts)
    return _execute(args, opts)

def stats_cpu(*opts):
    args = ['docker', 'stats', '--no-stream', '--format', \
        '"{{.Container}}: {{.CPUPerc}}"']
    opts = list(opts)
    return _execute(args, opts, True)

def get_image(instance_id):
    return inspect("--format", '"{{.Config.Image}}"', instance_id)


#GET_INSTANCE_NAMES_BY_ID: return the name of the
#containers running an image by image id
```

```
def get_instance_names_by_id(id):
    names = []
    instances = ps('--filter', 'ancestor=' + id,  \
       '--filter', 'status=running')
    instances_lines = instances.split('\n')
    instances_lines.pop(0)
    instances_lines.pop(-1)
    for line in instances_lines:
        line = line.split()
        names.append(line[-1])
    return names

#GET_INSTANCE_IDS_BY_ID: return the docker id from containers
#running an image by image id
def get_instance_ids_by_id(id):
    names = []
    instances = ps('--filter', 'ancestor=' + id,  \
       '--filter', 'status=running')
    instances_lines = instances.split('\n')
    instances_lines.pop(0)
    instances_lines.pop(-1)
    for line in instances_lines:
        line = line.split()
        names.append(line[0])
    return names



#GET_IP_LIST_BY_ID: retuns the ips from containers
#running an image by image id
def get_ip_list_by_id(id):
```

```
    names = get_instance_names_by_id(id)
    address = ""
    ips = []
    for name in names:
        found = False
        raw_inspect = inspect(name)
        inspect_lines = raw_inspect.split('\n')
        for line in inspect_lines:
            line = line.strip()
            if line.startswith('"' + 'IPAddress') and not found:
                address = line[IPAddress_string_start_position:]
                address = address[:IPAddress_string_end_position]
                if address == r"":
                    found = False
                else:
                    ips.append(address)
                    found = True
    return ips

def get_network_name_by_id(id):
    first_instance_name = get_instance_names_by_id(id)[0]
    raw_inspect = inspect(first_instance_name)
    inspect_lines = raw_inspect.split('\n')
    networkID = ""
    found = False
    for line in inspect_lines:
        line = line.strip()
        if line.startswith('"' + 'NetworkID') and not found:
            networkID = line[networkID_string_start_position:]
            networkID = networkID[:networkID_string_end_position]
            found = True
```

```
    return networkID

def get_ip_by_name(name):
    raw_inspect = inspect(name)
    inspect_lines = raw_inspect.split('\n')
    address = ""
    found = False
    for line in inspect_lines:
        line = line.strip()
        if line.startswith('"' + 'IPAddress') and not found:
            address = line[IPAddress_string_start_position:]
            address = address[:IPAddress_string_end_position]
            if address == r"":
                found = False
            else:
                found = True
    return address
```

C.2 Exemplos de Chamadas

```
#Criando um contêiner docker com acesso ao driver SGX (/dev/isgx),
#0.5 CPU, 800MB de memória e para um certo id de image (que
#também nomeia a rede e o volume padrão alocados).
container_id = docker.run('--network', image_id, '-m', 800 + 'M', \
   '--device=/dev/isgx','-d', '--cpus=0.5', '-v', image_id + ':' + \
   docker.default_volume_path, image_id)

#Construindo a imagem de um contêiner Docker com um certo
#id e um certo dockerfile.
docker.build('-t', image_id, dockerfile_dir)
```

```
# Criando uma subrede docker para isolar as aplicações.
#ip_strings.get_current_network() retorna um endereço de rede
#válido e consistente com as demais redes já criados.
docker.create_network('--subnet=' + \
   ip_strings.get_current_network() + "/16", id)
```

Apêndice D

Daemon do balanceamento de carga

```
import os
import time
import subprocess


LOG_LINE_SIZE = 14
HOST_LABEL_POSITION = 5
SERVER_LABEL_POSITION = 8


#log_path = "/home/gabrielf/dev/sco/assets/support-" + \
"containers/load_balancer/log.test"
log_path = '/var/log/haproxy.log'
conf_path = '/etc/haproxy/haproxy.cfg'
#conf_path = "/home/gabrielf/dev/sco/assets/support-" + \
   "containers/load_balancer/haproxy.cfg.test"
custom_connection_block_line = "cookie SERVERID insert indirect nocache
tcp_conf_line = "frontend sco 0.0.0.0:81"


n_src_addresses = 0
```

```
host_labels = []
lines = []
last_lines = []



def read_last_line(in_file):
    tail(in_file, 1)

def tail( f, lines=20 ):
    total_lines_wanted = lines

    BLOCK_SIZE = 1024
    f.seek(0, 2)
    block_end_byte = f.tell()
    lines_to_go = total_lines_wanted
    block_number = -1
    blocks = [] # blocks of size BLOCK_SIZE, in reverse order starting
                # from the end of the file
    while lines_to_go > 0 and block_end_byte > 0:
        if (block_end_byte - BLOCK_SIZE > 0):
            # read the last block we haven't yet read
            f.seek(block_number*BLOCK_SIZE, 2)
            blocks.append(f.read(BLOCK_SIZE))
        else:
            # file too small, start from begining
            f.seek(0,0)
            # only read what was not read
            blocks.append(f.read(block_end_byte))
        lines_found = blocks[-1].count('\n')
        lines_to_go -= lines_found
```

```
        block_end_byte -= BLOCK_SIZE
        block_number -= 1
    all_read_text = ''.join(reversed(blocks))
    return '\n'.join(all_read_text.splitlines()[-total_lines_wanted:])

def get_host_label(log_line):
    if is_log_line(log_line):
        fields = str.split(log_line)
        full_address = fields[HOST_LABEL_POSITION]
        short_address = full_address.split(":")[0]
        return short_address


def get_server_label(log_line):
    if is_log_line(log_line):
        words = str.split(log_line)
        full_name = words[SERVER_LABEL_POSITION]
        print "FULL SERVER NAME IS " + full_name
        short_name = full_name.split("/")[1]
        return short_name

def is_log_line(line):
    words = str.split(line)
    if len(words) != LOG_LINE_SIZE:
        return False
    elif words[13] != "0/0":
        return False
    else:
        return True
```

```
def add_rule(server_label, host_label):
    with open(conf_path, 'r+') as conf_file:
        lines = conf_file.readlines()
        position_src = get_next_src_rule_position(lines)
        new_rules = assemble_new_rule(server_label, host_label)
        print "[DEBUG] positions_src is " + str(position_src)
        lines.insert(position_src, new_rules[0])
        position_bind = get_next_bind_rule_position(lines)
        lines.insert(position_bind, new_rules[1])
        conf_file.seek(0)
        conf_file.writelines(lines)
        conf_file.close()
        args = ['/etc/init.d/haproxy', 'reload']
        subprocess.call(args)



def get_next_src_rule_position(lines):
    print "[DEBUG] next src position lines are " + str(lines)
    current_position = 0
    reached_tcp_conf_line = False
    for i in lines:
        if i.rstrip() == tcp_conf_line:
            reached_tcp_conf_line = True
        if reached_tcp_conf_line and i == "\n":
            return current_position + 1
        current_position += 1


def get_next_bind_rule_position(lines):
    current_position = 0
```

```
    reached_src_rules = False
    for i in lines:
        if i.startswith("acl"):
            reached_src_rules = True
        if reached_src_rules and not i.startswith("acl"):
            return current_position
        current_position += 1

def assemble_new_rule(server_label, host_label):
    global n_src_addresses
    src_line = "acl rule" + str(n_src_addresses) + " src " +  \
       host_label + '\n'
    bind_line = "use_backend " +  server_label + "_bknd" +  \
       " if rule" + str(n_src_addresses) + '\n'
    n_src_addresses += 1
    return (src_line, bind_line)


current_change_time = os.stat(log_path).st_mtime

debug_int = 0
while True:
    last_change_time = os.stat(log_path).st_mtime
    if current_change_time != last_change_time:
        with open(log_path, 'rw') as logfile:
            lines = logfile.readlines()
            #verify difference between last and current reads
            last_line = lines[-1]
            if not is_log_line(last_line):
                logfile.close()
                current_change_time = last_change_time
```

```
            continue
        last_host_label = get_host_label(last_line)
        last_server_label = get_server_label(last_line)
        if(last_host_label not in host_labels):
            add_rule(last_server_label, last_host_label)
            host_labels.append(last_host_label)
        logfile.seek(0)
        last_lines = logfile.readlines()
        logfile.close()
time.sleep(5)
current_change_time = last_change_time
```

Apêndice E

Imagem base padrão do SCO

```
FROM ubuntu:16.04

MAINTAINER Amanda Souza <amandasouza@lsd.ufcg.edu.br>

RUN DEBIAN_FRONTEND=noninteractive apt-get update &&  \
   apt-get install -y software-properties-common && \
   add-apt-repository -y ppa:ubuntu-toolchain-r/test && \
   apt-get update && \
   apt-get install --no-install-recommends -y git build-essential
   ocaml automake python autoconf libtool \
       libcurl4-openssl-dev libprotobuf-dev libprotobuf-c0-dev protobuf
       unzip wget libssl-dev g++-5 cmake nano kmod && \
       rm -rf /var/lib/apt/lists/*

#Install SGX Drivers and PSW/SDK
WORKDIR /home/sgx/drivers
RUN  git clone https://github.com/01org/linux-sgx.git
WORKDIR /home/sgx/drivers/linux-sgx

RUN ./download_prebuilt.sh && \
    make && \
```

```
    make psw_install_pkg && \
    make sdk_install_pkg && \
    mkdir -p /opt/intel && \
    cd /opt/intel && \
    /home/sgx/drivers/linux-sgx/linux/installer/bin/sgx_linux_x64_psw_*
    python -c "print 'no\n/opt/intel'" | /bin/bash -c '/home/sgx/driver
    /bin/bash -c 'source /opt/intel/sgxsdk/environment' && \
    rm -rf /home/sgx/drivers/*

RUN apt-get clean && \
    apt-get autoclean -y && \
    apt-get autoremove -y  && \
    rm -rf /bd_build && \
    rm -rf /tmp/* /var/tmp/* && \
    rm -rf /var/lib/apt/lists/* && \
    rm -f /etc/ssh/ssh_host_* && \
    du -sh /var/cache/apt/archives

CMD ["/bin/bash"]
```