Capítulo 1

Processamento confidencial de dados de sensores na nuvem

Andrey Brito (UFCG), Clenimar Souza (UFCG), Fábio Silva (Scontain), Lucas Cavalcante (UFCG), Matteus Silva (UFCG)

Abstract

This tutorial presents how to use confidential computing tools for developing IoT applications that process potentially sensitive data in the cloud. For confidential processing, our tutorial uses Intel SGX through both Intel SGX SDK (for developing applications and services from scratch) and a runtime environment, SCONE (ideal for existing applications). For the dissemination of IoT data, we use Apache Kafka, and for orchestrating applications, we use Kubernetes. The concepts presented will be illustrated through a distributed application for power consumption monitoring.

Resumo

Este minicurso apresenta como ferramentas de computação confidencial podem ser usadas para o desenvolvimento de aplicações que processam dados potencialmente sensíveis de aplicações de Internet das Coisas na nuvem. Para o processamento confidencial, nosso minicurso usará tanto o SDK (Software Development Kit) para SGX da Intel (para novas aplicações e serviços) como uma plataforma de execução, SCONE (ideal para execução de aplicações existentes). Para disseminação de dados de Internet das Coisas, nós utilizaremos Apache Kafka e para a orquestração de aplicações, usaremos Kubernetes. Os conceitos apresentados aqui serão ilustrados através de uma aplicação distribuída de processamento de dados de consumo de energia elétrica.

1.1. Introdução

Computação na nuvem e Internet das Coisas têm mudado o papel da computação na vida das pessoas. Por um lado, computação na nuvem mudou como empresas e empreendedores individuais abordam novos projetos. O longo processo de planejamento para investimentos em infraestrutura para hospedar novas aplicações foi substituído pelo modelo pague-por-uso da computação na nuvem, onde um custo mínimo de infraestrutura é necessário para publicar uma nova aplicação na Internet. Além disso, uma vez publicada, a aplicação pode ser expandida sob demanda, com a

nuvem fornecendo tantos recursos quanto necessário para o crescimento da capacidade de atendimento dos usuários.

Por outro lado, a Internet das Coisas trouxe aspectos avançados da computação para ainda mais perto do usuário final. Sensores conectados podem monitorar detalhadamente os hábitos das pessoas e, com a ajuda de técnicas de inteligência artificial, prover sugestões como rotas otimizadas para o caminho de casa no momento em que o carro é ligado (e antes mesmo que o usuário manifeste alguma intenção de usar o GPS) ou sugestões como o aumento do número de passos ao longo do dia ou uma maior obediência dos horários de dormir à noite.

Infelizmente, a Internet da Coisas não traz apenas benefícios para o público em geral. Com hábitos e ações monitorados constantemente, a privacidade e a segurança dos dados têm se tornado uma preocupação cada vez mais pertinente no dia-a-dia das pessoas. Assim, as aplicações inovadoras que usam cada vez mais dados para melhorar o nosso dia-a-dia têm que ser desenvolvidas com mais preocupação a respeito da segurança dos dados que processam. Esta necessidade faz com que mesmo pequenos empreendedores, que só conseguem fazer suas aplicações atingirem o público geral graças às facilidades de provisionamento de infraestrutura permitidas pela computação na nuvem, tenham que desenvolver aplicações que apresentem boas garantias de segurança de dados. Finalmente, e felizmente para o público geral, esta necessidade técnica se torna uma obrigação, dada a vigência das novas leis de proteção de dados no Brasil (LGPD, Lei Geral de Proteção de Dados) e na Europa (GDPR, *General Data Protection Regulation*, ou Regulamento Geral sobre a Proteção de Dados).

Do ponto de vista técnico, as tecnologias de computação na nuvem e segurança de dados têm evoluído para atender às necessidades de eficiência de custo e proteção. **Computação nativa da nuvem** (ou, do inglês, *Cloud Native Computing*) é uma área que foca na gerência eficiente de recursos na nuvem, minimizando, por exemplo, máquinas virtuais ociosas (usando grãos menores de escalonamento, contêineres) e problemas de escalabilidade ou disponibilidade (com monitoramento e orquestração automatizada). Já **computação confidencial** tem como objetivo minimizar a necessidade de confiança assumida nas infraestruturas e seus operadores, deslocando-a para software e hardware, que são menos passíveis de vazamentos acidentais.

Neste tutorial, combinamos estas duas tendências tecnológicas em uma área que chamamos de **computação confidencial nativa da nuvem** (CCNN) e validamos a aplicação deste conceito em uma aplicação de processamento de dados de sensores de forma confidencial na nuvem.

O restante do capítulo está organizado da seguinte forma: na Seção 1.2, apresentamos os principais conceitos de computação nativa da nuvem e duas ferramentas populares que serão utilizadas neste capítulo; na Seção 1.3 apresentamos computação confidencial, com foco na tecnologia Intel SGX, a mais popular atualmente; já na Seção 1.4 combinamos os dois conceitos para a construção de uma aplicação confidencial nativa da nuvem, detalhando seus funcionamento e implantação; finalmente, a Seção 1.5 apresenta boas práticas e caminhos promissores nesta área. Como complemento, o Apêndice A detalha como máquinas virtuais podem ser criadas em um provedor de nuvem público para a execução de aplicações confidenciais.

1.2. Computação nativa da nuvem

Embora muitos dos conceitos fundamentais estivessem sendo discutidos há bastante tempo, o termo Computação Nativa da Nuvem (do inglês, *Cloud Native Computing*) ganhou impulso com a fundação da ***Cloud Native Computing Foundation*** (CNCF) em 2015[1]. A CNCF é parte da Linux Foundation, uma organização sem fins lucrativos criada para disseminar o uso de Linux e a criação e adoção de projetos de código aberto. A CNCF, por sua vez, tem como objetivo promover o uso de contêineres e o alinhamento da indústria em torno do desenvolvimento desta tecnologia, que permite o empacotamento de aplicações de forma mais compacta e sua instanciação de forma mais ágil, melhorando a eficiência da operação das aplicações.

A CNCF define computação nativa da nuvem a partir das características de suas aplicações:

- As aplicações são criadas e executam de forma escalável em ambientes modernos e dinâmicos, como nuvens públicas, privadas e híbridas;
- As aplicações usam tecnologias como contêineres, redes de serviços (*service meshes*), microsserviços, infraestrutura imutável, e APIs declarativas;
- Os componentes da aplicação são desacoplados e sua implantação é suportada por altos níveis de automação, que permitem mudanças frequentes com impactos previsíveis e limitados.

O foco em nuvem ajuda a priorizar a redução de custos de operação e o uso de infraestruturas genéricas. Na mesma direção, o conceito de **infraestrutura imutável** valoriza a automação e repetibilidade. A ideia é que uma vez que servidores não podem ser ajustados depois de colocados em operação, elimina-se a ideia de que servidores precisam de cuidado individual, o que tipicamente é uma atividade manual e custosa. Assim, diminui-se o tempo gasto em operações de manutenção e aumenta-se a utilização de mecanismos automatizados para retirada e inserção de servidores.

Em cima dessa infraestrutura genérica e imutável vêm as tecnologias para gerência das aplicações. O uso de tecnologias como contêineres, gerenciados por plataformas de redes de serviços, e que hospedam aplicações de responsabilidade limitada na forma de microsserviços permitem a agilidade de ativação e desativação das instâncias de um componente, mantendo a infraestrutura ajustada à real necessidade momentânea, o que garante tanto escalabilidade como eficiência. Este conjunto de características é então complementado com o uso de APIs declarativas, que priorizam a especificação do que deve ser feito e não a forma que deve o objetivo deve ser atingido, em contraste com a ideia de programação e especificação imperativa, onde o processo é descrito e não o objetivo. Uma vez que o sistema tem informações do estado esperado, o monitoramento é simplificado, aumentando o suporte do sistema a operações de auto-recuperação (do inglês, *self-healing*).

1.2.1. Microsserviços

Um microsserviço é um componente da arquitetura de uma aplicação que tem uma responsabilidade clara e limitada. Microsserviços idealmente seguem estilos de

[1] https://www.cncf.io/

interface usando padrões de troca de dados abertos e populares como **JSON** (*JavaScript Object Notation*). Estas mensagens JSON são então transportadas por sistemas de **comunicação direta ou indireta**.

Um sistema de comunicação é direto quando existe uma ligação entre o remetente e os destinatários. Por outro lado, a comunicação é indireta quando existe um intermediário que desacopla o remetente dos destinatários. A ausência de intermediários permite maior eficiência e desempenho na comunicação direta, no entanto, implica em maior acoplamento. Já a comunicação indireta permite um desacoplamento de espaço e de tempo entre os remetentes e destinatários. O desacoplamento de espaço significa que o remetente não precisa conhecer o destinatário, conhecendo apenas o intermediário, o sistema de comunicação. Este desacoplamento facilita a replicação de mensagens e a substituição gradual de componentes (ex., ter duas versões de um microsserviço operando em paralelo por um tempo). O desacoplamento no tempo retira a limitação que o remetente e o destinatário estejam ativos ao mesmo tempo. As mensagens podem então ser armazenadas temporariamente no sistema de comunicação e entregues posteriormente aos destinatários. Esta abordagem ajuda a lidar com problemas de conectividade (como em Internet das Coisas e computação na borda) e com a recuperação após falhas, já que mensagens podem ser mantidas temporariamente.

Para comunicação direta, o estilo REST favorece a padronização e, portanto, a interoperabilidade. Isso é consequência do fato que o uso de um protocolo bem definido (HTTP, *HyperText Transfer Protocol*), com operações uniformes (POST, GET, PUT, DELETE), reusa o conhecimento dos desenvolvedores e são melhor suportados pelas ferramentas de desenvolvimento.

Para comunicação indireta, os dois estilos dominantes são filas de mensagens e barramentos de mensagens. O estilo de **fila mensagens** implementa uma fila (seja ela centralizada ou distribuída) que armazena as mensagens temporariamente, desacoplando remetentes e destinatários. Este estilo é mais popular quando a comunicação é de um para um. O estilo de **barramento de mensagens** é mais popular quando a comunicação é de um para muitos. Neste caso, um componente pode passar a mensagem para o sistema de comunicação e vários outros podem estar conectados no mesmo barramento, estando aptos a receber a mensagem. A lógica de roteamento pode variar, sendo o padrão publicar-assinar baseado em tópicos (do inglês, *topic-based publish-subscribe* [Euster et al. 2003]) o mais popular. O padrão publicar-assinar será o estilo usado no restante deste trabalho e será descrito em mais detalhes na próxima seção. É importante notar que mesmo quando a comunicação é um para um na maior parte do tempo, ter a possibilidade de conectar dinamicamente outros componentes e replicar as mensagens é útil para fins de teste (ex., alimentando uma nova versão de um componente), registro (ex., adição de um componente de *log* que armazena mensagens) ou tolerância a falhas (ex., duplicação das responsabilidades). Além disso, as ferramentas mais populares para barramentos de mensagens permitem o seu uso flexível como fila ou barramento.

Seja sobre um mecanismo de comunicação por fila, sobre um barramento ou sobre REST, JSON é um formato de troca bastante popular. JSON é um formato compacto, de padrão aberto independente, e que permite a troca de dados de forma simples entre os componentes. Além de também usufruir dos benefícios da

popularidade, como o conhecimento prévio dos desenvolvedores e a ampla disponibilidade de bibliotecas para sua manipulação, o padrão JSON é legível para humanos através de sua notação em texto plano e estrutura de pares chave-valor.

A popularidade de ferramentas para a implementação das abordagens usadas para comunicação entre microsserviços (REST e JSON) tem outra consequência desejável conhecida como programação poliglota. Uma vez que a comunicação entre os serviços é explícita e as bibliotecas para implementação dos canais (REST, filas, ou barramentos) e processamento das mensagens (JSON) estão disponíveis e bem documentadas nas linguagens populares, existe uma grande liberdade nas escolhas para a implementação de cada microsserviço. Assim, diferentes desenvolvedores podem implementar seus microsserviços usando as linguagens de programação mais familiares e as bibliotecas mais adequadas à realização das responsabilidades daquele microsserviço. Por este mesmo motivo, inovar é simples, já que uma vez que alguma ferramenta se torna disponível para realizar uma tarefa bem a reescrita daquele microsserviço na linguagem de programação recomendada para aquela nova biblioteca terá impacto limitado no resto da aplicação.

1.2.2. Barramentos de mensagens e Apache Kafka

Também conhecidos como sistemas baseados em eventos distribuídos, os sistemas publicar-assinar são caracterizados pelo baixo acoplamento das interações entre as diferentes entidades participantes, sendo este um requisito de sistemas de grande escala [Eugster et al., 2003]. Esse tipo de sistema é composto por publicadores, assinantes, e um serviço de eventos, também conhecido como barramento de mensagens ou, do inglês, *broker*.

Os publicadores divulgam eventos estruturados para o *broker*, e assinantes expressam interesse em eventos ou padrões de eventos específicos por meio de assinaturas. Em outras palavras, publicadores publicam informações para o *broker*, e através de uma etapa de filtragem baseado nas assinaturas recebidas, é feito o roteamento e entrega dessas mensagens para os respectivos assinantes. Esse paradigma para comunicação é bastante aplicado onde há disseminação e processamento de informações em larga escala e de forma contínua, como sistemas financeiros, sistemas de monitoramento de recursos, aplicações de IoT, entre outros.

O desacoplamento provido pelos serviços de eventos em conjunto com os publicadores e assinantes ocorre em três dimensões: espaço, tempo e sincronização [Eugster et al., 2003]. O desacoplamento espacial está relacionado com a ausência de comunicação direta entre os publicadores e assinantes. Mais especificamente, os publicadores sequer conhecem os endereços dos assinantes. O desacoplamento temporal está relacionado com a falta de necessidade das partes estarem interagindo ao mesmo tempo, uma vez que publicações podem ser feitas, e podem ser consumidas eventualmente. E, por fim, o desacoplamento de sincronização ocorre pois os consumidores são notificados de que existem novos eventos de maneira assíncrona, enquanto estão realizando atividades concorrentes.

Existem diferentes esquemas para assinaturas em sistemas publicar-assinar, onde os mais populares são o baseado em tópico (do inglês, *topic-based*) e o baseado em

conteúdo (do inglês, *content-based*) [Eugster et al., 2003]. Aqui usaremos o esquema baseado em tópico, onde os eventos publicados são associados a um tópico específico por seus respectivos publicadores, e os assinantes, por sua vez, assinam a tópicos. Na medida em que eventos vão sendo publicados, serão entregues para os assinantes dos tópicos ao qual estes pertencem. Existem diferentes implementações de serviços de eventos que suportam o esquema baseado em tópico, neste tutorial utilizaremos o Apache Kafka[2].

O Apache Kafka, aqui chamado simplesmente de Kafka, foi desenvolvido pelo LinkedIn[3] e posteriormente doado em 2011 para a Apache Software Foundation. O Kafka é um sistema publicar-assinar de código aberto bastante popular e escalável, usado em inúmeras empresas e sistemas de grande porte[4]. A sua implantação é flexível, permitindo que seja usado de forma portável em aplicações de pequeno porte ou em *clusters* para implantações escaláveis e tolerantes a falhas. Quando usado em modo de *cluster*, cada nó é tipicamente chamado de ***broker***. Além de funcionar como barramento de mensagens, integrandos produtores e consumidores assíncronos, o Kafka se tornou popular também como hub de mensagens em um sistema baseado em microsserviços.

No Apache Kafka, o **tópico** é a entidade para a qual as mensagens são publicadas e consumidas. Um tópico é dividido em **partições**. As partições dividem os dados de um tópico para permitir a escrita de vários produtores e leituras de vários consumidores no mesmo. Toda mensagem em uma partição possui um identificador chamado ***offset***, que é sequencial, monotonicamente crescente e denomina a posição de uma mensagem numa partição. Consequentemente, toda mensagem em um sistema Kafka pode ser identificada únicamente através da 3-upla *(Tópico, Partição, Offset)*.

Um tópico possui um fator de replicação, que determina a quantidade de réplicas, que será distribuída pelos *brokers* do *cluster*. Um *broker* pode ser eleito o líder de uma determinada partição, o que implica que todas as leituras e escritas em uma partição devem ser feitas exclusivamente a partir dele. É o líder que também coordena a atualização das réplicas com novas mensagens, e se um líder falha, uma réplica do *broker* se torna o novo líder. Uma vez que diferentes publicações para diferentes partições são gerenciadas por diferentes *brokers*, o sistema atinge escalabilidade.

Consumidores para um dado tópico podem ser independentes ou se organizarem na forma de grupos de consumidores (chamados *ConsumerGroups*). Quando independentes, todos recebem as mesmas mensagens. Quando em grupo, cada consumidor dentro do grupo lê de uma única partição, de maneira que o grupo como um todo consome todas as mensagens de um tópico.

É delegado a um dos *brokers* do *cluster* Kafka o papel chamado coordenador de grupo. Isso é feito para cada novo grupo de consumidores que se inscreve no sistema. Dentre suas responsabilidades, o coordenador de grupo detecta através do uso de pacotes de controle, se todos os consumidores de um grupo estão em funcionamento. Caso algum venha a parar de funcionar, ele é responsável por engatilhar o rebalanceamento através do líder do grupo.

[2] https://kafka.apache.org/
[3] https://www.linkedin.com
[4] https://kafka.apache.org/powered-by

É considerado líder do grupo o primeiro consumidor a juntar-se ao grupo. O líder recebe do coordenador de grupo uma lista de todos os consumidores do grupo, e é responsável por atribuir um subconjunto de partições a cada consumidor. Se existem mais consumidores do que partições, alguns deles ficarão em espera, mas se existem mais partições do que consumidores, alguns consumidores irão ler mensagens de mais de uma partição. Felizmente, a atuação do líder do grupo é feita de forma transparente, de acordo com a implementação do cliente ou biblioteca Kafka utilizada pela aplicação.

A Figura 1.1 apresenta algumas configurações de grupos de consumidores com diferentes balanceamentos. Nesta figura temos um tópico *xpto*, que possui replicação 2 e quatro partições. O nó de *id* 0 lidera as partições *P1* e *P4*, enquanto o nó de id 1 lidera as partições *P2* e *P3*. Como discutido acima, cada líder gerencia a entrega de mensagens aos seus consumidores.

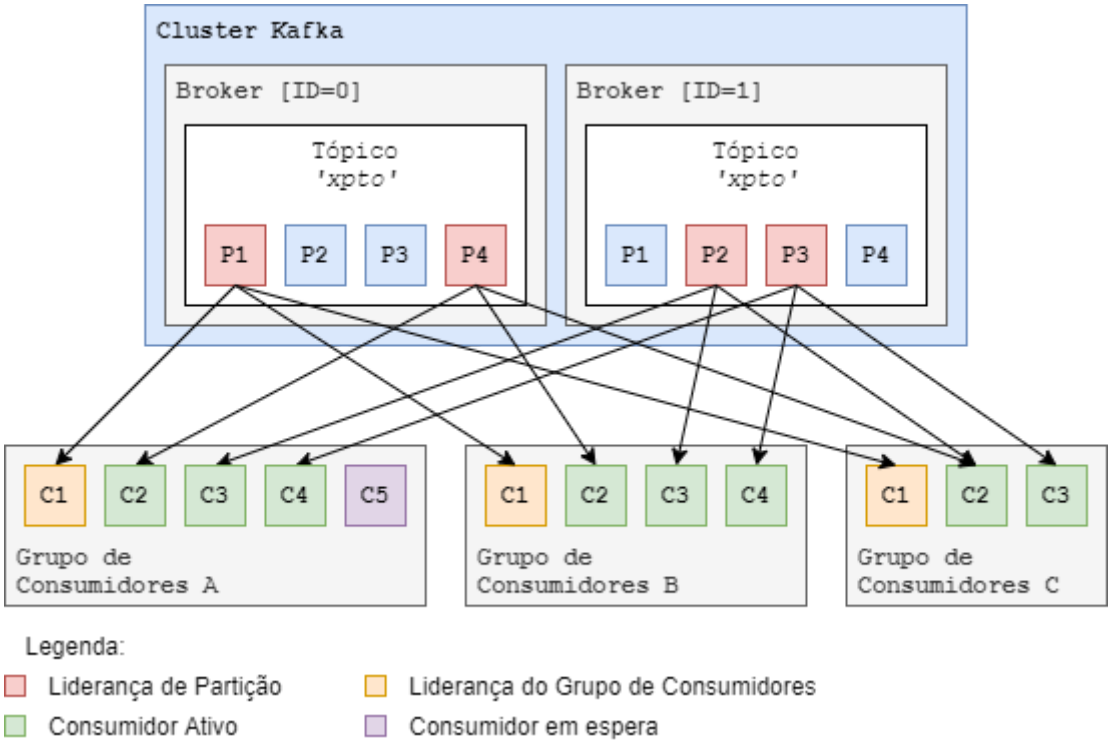

Figura 1.1. Balanceamento de grupos de consumidores

Na Seção 1.2.4 faremos uma implantação do Apache Kafka para nossa aplicação caso de uso. Embora o Apache Kafka tenha mecanismos de proteção para dados em trânsito e mecanismos para autenticação, ele não dispõe de mecanismos para proteção para dados em repouso e em processamento. Como discutido na Seção 1.4.3, estes níveis adicionais de proteção são importantes para o nosso modelo de ameaças, que considera que dados sensíveis podem estar em infraestruturas compartilhadas e gerenciadas por terceiros. Finalmente, a Seção 1.6 apresenta discussões sobre outras boas práticas e otimizações para o caso de uso apresentado, incluindo aspectos relacionados ao Kafka.

1.2.3. Kubernetes

A popularização de microsserviços e a crescente adoção de contêineres como principal forma de implantação e distribuição de aplicações nativas da nuvem impulsionaram o surgimento de aplicações para gerenciar grandes conjuntos de contêineres em produção. Estas aplicações, conhecidas como orquestradores de contêineres, atuam no topo da infraestrutura (seja esta composta por máquinas físicas ou máquinas virtuais), por vezes abstraindo-a complemente, o que diminui acoplamento e, por conseguinte, provê mais portabilidade para as aplicações.

Kubernetes é, atualmente, o orquestrador de contêineres padrão da indústria. Lançado oficialmente em 2014, o projeto foi desenvolvido pela Google e teve sua arquitetura fortemente influenciada pelo orquestrador de contêineres interno da empresa, Borg [Verma et al., 2015]. Após o lançamento, Google e Linux Foundation fundaram a CNCF, que passou a gerenciar o projeto Kubernetes, além de formular e disseminar as diretrizes do que seria a computação nativa da nuvem.

Um *cluster* Kubernetes é composto por dois tipos de nós: *master* e *worker*. Nós *master* são responsáveis por executar os componentes do painel de controle de Kubernetes, como o servidor de API (*api-server*) e o gerenciador de controladores (*controller manager*). É recomendado que nós *master* sejam usados exclusivamente para esse fim, e não executem aplicações de usuários. Nós *worker*, por sua vez, são dedicados a executar aplicações de usuários. É comum que *clusters* tenham apenas um *master*. Contudo, para *cluster* com requisitos de alta disponibilidade e tolerância a falha, é recomendável que se use ao menos três nós *master*. A persistência de todos os objetos e de todo o estado do *cluster* é efetuada no sistema de armazenamento distribuído chave-valor, *etcd*[5].

Um dos pontos-chave da arquitetura de Kubernetes é sua abordagem centrada em aplicação. Suas primitivas (*pods*, *deployments*, *daemonsets*, *services*, etc., discutidas abaixo) e poderosa API declarativa permitem que o usuário não precise se preocupar com detalhes da infraestrutura ou de escalonamento e alocação de recursos. Essa característica possibilita que Kubernetes gerencie infraestruturas muito grandes de forma eficiente e transparente pro usuário. De fato, atualmente, o orquestrador suporta oficialmente *clusters* de até 5000 nós[6]. Uma outra consequência dessa arquitetura é que a API de Kubernetes, por ser independente da infraestrutura, permite a portabilidade e interoperabilidade de aplicações em infraestruturas distintas (por exemplo, de provedores de nuvem diferentes).

Um conjunto de controladores assegura que a aplicação definida pelo usuário esteja sempre no seu estado desejado. Dessa forma, é possível delegar para o orquestrador, através de sua API, tarefas como gerenciamento de falhas, replicação, atualização e escalabilidade automática de aplicações. Além disso, ao longo dos anos, a API de Kubernetes fora estendida e alavancada por diversas outras aplicações, o que criou um ecossistema de computação nativa da nuvem extremamente rico e versátil. Uma infinidade de ferramentas de monitoramento, registro, controle de acesso, armazenamento, dentre outras, integra-se de forma rápida e fácil ao orquestrador,

[5] https://etcd.io/
[6] https://kubernetes.io/docs/setup/best-practices/cluster-large/

estendendo seu potencial e tornando a operação e provisionamento de infraestruturas e aplicações mais prático e robusto[7].

Das primitivas de Kubernetes, destacam-se *Pod*, *Deployment, Service*, *Ingress*, *DaemonSet* e *StatefulSet*, a seguir explicadas em maior detalhe.

Pod. O menor grão de uma aplicação, representam um ou mais contêineres que podem ser enxergados como um conjunto coeso, servindo a um propósito em comum. Por exemplo, um contêiner de banco de dados e um contêiner auxiliar que funciona como um *proxy* desse mesmo banco de dados. *Pods* têm seu próprio espaço de rede (um endereço IP e portas) e sistema de arquivos, que são compartilhados por todos os seus contêineres. *Pods* também são considerados efêmeros pelo escalonador de Kubernetes, que pode terminá-los e movê-los por alguma razão (liberar recursos em um nó sob pressão, por exemplo). Por esse motivo, *Pods* devem sempre ser criados com um controlador de replicação associado, o que é oferecido por outras primitivas, como *Deployments* e *StatefulSets*.

Deployment. Representa um *Pod* associado a um controlador de replicação, que garante o estado desejado definido. É possível definir número de réplicas desejadas e políticas de recuperação de falhas. Por exemplo, caso um nó da infraestrutura fique indisponível em razão de uma falha, deixando um *Deployment* com menos réplicas que o desejado, o controlador rapidamente criará tantas réplicas quantas forem necessárias para que o número desejado de réplicas ativas seja restabelecido.

Service. Permite que *Pods* sejam descobertos e acessados por outras entidades. Como *Pods* são efêmeros, seus endereços IP podem variar constantemente. Um *Service* é, na prática, um balanceador de carga de camada 4 (de acordo com o modelo OSI) para *Pods*. Através de seletores e etiquetas é possível selecionar quais *Pods* são expostos por determinado *Service*. *Services* têm um endereço IP interno fixo e uma entrada no servidor de nomes do *cluster* (DNS) que podem ser acessados por outras entidades. Para que *Services* sejam acessíveis de fora do *cluster*, existem dois tipos especiais de *Service*: *NodePort*, que expõe uma porta na rede dos nós, permitindo que o *Service* seja alcançável através do endereço IP de qualquer nó na porta escolhida, e *LoadBalancer*, que aciona o provedor da infraestrutura na criação de um balanceador de carga dinâmico com um endereço IP público.

Ingress. Atua como um balanceador de carga de camada 7 (de acordo com o modelo OSI) para *Services*, o que possibilita um conjunto maior de funcionalidades, já que um *Ingress* atua na camada de aplicação (suportando, por exemplo, endereços HTTP, como *www.example.com/app*), enquanto *Services* expõem apenas um endereço IP e uma porta. Através de um *Ingress* é possível particionar tráfego de forma mais complexa, levando em consideração nomes e *hosts* virtuais ou implantações canário, além de possibilitar o término de conexões seguras (SSL). Kubernetes não implementa a lógica de *Ingress*. Em vez disso, a API de *Ingress* precisa ser implementada por um controlador especial, chamado controlador de *Ingress*, que é responsável por implantar os balanceadores de carga e gerenciar objetos do tipo *Ingress* no *cluster*. Os

[7] https://landscape.cncf.io/

balanceadores de carga mais populares do mercado, como NGINX, HAProxy e Envoy, possuem seus próprios controladores de *Ingress* disponíveis.

DaemonSet. Representa um *Pod* associado a um controlador de replicação que garante a existência de exatamente uma réplica da aplicação em cada nó do *cluster*, sendo possível filtrar nós através de seletores e etiquetas. Útil para implantar aplicações como coletores de registro, agentes de armazenamento, de monitoramento ou de atestação, entre outros.

StatefulSet. Representa um *Pod* associado a um controlador de *StatefulSet,* que persiste a identidade de cada réplica. Dessa forma, é possível implantar aplicações que dependem de um estado, como um gerenciador de banco de dados. *Pods*, apesar de efêmeros, passam a ter uma identidade única e persistente que, combinada com estratégias de persistência de dados (provisionamento de volumes, por exemplo), permite a manutenção de estado mesmo em caso de falha.

Objetos da API de Kubernetes são declarados através de manifestos na linguagem YAML (acrônimo do inglês *Yet Another Markup Language*), como mostrado na Figura 1.2, que cria um *Deployment* e um *Service* para a aplicação NGINX. Os manifestos definem a versão da API do Kubernetes que irá processar a requisição (*apiVersion*), metadados (*spec.selector.matchLabels*, por exemplo, define uma etiqueta que permite que o *Service* encontre os *Pods* a serem expostos). No *Deployment* é possível, ainda, ver o número de réplicas desejadas (3), a imagem do contêiner a ser utilizada (*nginx:1.14.2*, a ser obtida de repositórios cadastros, como *hub.docker.com*), e a porta na qual a aplicação estará acessível (80). O *Service*, por sua vez, escuta na porta 80 (*port: 80*) e balanceia o tráfego para a porta 80 (*targetPort*) dos contêineres selecionados (através das etiquetas em *spec.selector*). Assim, é possível acessar o serviço através do nome *my-nginx:80*.

```
01 | apiVersion: apps/v1
02 | kind: Deployment
03 | metadata:
04 |  name: my-nginx
05 | spec:
06 |  selector:
07 |   matchLabels:
08 |      run: my-nginx
09 |  replicas: 3
10 |  template:
11 |    metadata:
12 |      labels:
13 |        run: my-nginx
14 |    spec:
15 |      containers:
16 |      - name: my-nginx
17 |        image: nginx:1.14.2
18 |        ports:
19 |        - containerPort: 80
20 | ---
21 | apiVersion: v1
22 | kind: Service
23 | metadata:
24 |   name: my-nginx
25 |  labels:
26 |    run: my-nginx
27 | spec:
28 |  ports:
29 |  - port: 80
30 |    targetPort: 80
31 |    protocol: TCP
```

```
32 |  selector:
33 |    run: my-nginx
```

Figura 1.2. Exemplo de manifesto Kubernetes para aplicação NGINX

Atualmente, a maior parte dos provedores de nuvem já oferecem serviços de Kubernetes gerenciado, que permitem a criação de *clusters* de forma simples e rápida. Para instalação em infraestrutura própria, há vários instaladores certificados pela CNCF, que cobrem casos de uso e infraestruturas diversas (*baremetal*, *clusters* de borda, *clusters* locais para teste e desenvolvimento). Um dos mais versáteis instaladores, MicroK8s[8], é provido pela Canonical, e permite a instalação rápida em máquinas ou *clusters* locais através do gerenciador de pacotes do Ubuntu. Os parceiros certificados deste projeto incluem distribuições, instaladores e ambientes de hospedagem[9,10].

1.3. Caso de uso: processamento de sensores de consumo de energia

O nosso caso de uso é uma versão simplificada do sistema LiteCampus[11] [Silva, Silva e Brito, 2020]. O LiteCampus é um sistema de processamento de dados de sensores, especializado na gerência de consumo de energia elétrica. Este sistema permite aos usuários, por exemplo, acompanhar o histórico de consumo, estimar contas, identificar consumo por equipamento, além de receber alertas para anomalias que possam danificar equipamentos ou implicar em multas na conta de energia.

O LiteCampus usa o modelo publicar-assinar, onde os diferentes componentes interagem através do barramento de mensagens Kafka, conforme ilustrado na Figura 1.3. Através do Kafka são disseminados dados potencialmente confidenciais e, portanto, estes dados devem ser protegidos. Os principais componentes figura são os seguintes.

- ***Smart meters***: Também conhecidos como medidores de energia inteligentes, são dispositivos computacionais capazes de monitorar o consumo em uma rede elétrica e transmiti-lo através de um canal de comunicação. Neste caso, os medidores estão instalados nos quadros elétricos e transmitem as informações relativas ao seu consumo para o LiteKafka Gateway, através do uso de uma conexão segura HTTPS (HTTP sobre TLS).
- ***LiteKafka Gateway***: Se comunica com os medidores usando protocolos padrão ou próprios para recuperar e publicar as informações de consumo. Ele é necessário pois os medidores, por serem dispositivos limitados, tipicamente não suportam protocolos complexos. As informações coletadas são transformadas em mensagens Kafka e publicadas em um tópico no barramento de mensagens.
- ***Cluster* Kafka**: Conjunto de brokers Kafka configurados de modo prover escalabilidade e disponibilidade.
- **Detector de Anomalias de Energia**: Componente que analisa os dados do sensores de energia e emite alertas em situações relevantes. Este componente é

[8] https://microk8s.io/
[9] https://kubernetes.io/partners/#conformance
[10] Atualmente, o suporte a Kubernetes com mecanismos de computação confidencial com Intel SGX é provido apenas pela Microsoft Azure [Microsoft, 2020]. A instalação em outros provedores é possível através de instalação do Kubernetes em máquinas físicas (*bare-metal*) com suporte a Intel SGX.
[11] https://litecampus.lsd.ufcg.edu.br/

assinante do tópico de medições, publicadas pelo LiteKafka Gateway, e publica os alertas em um tópico específico.

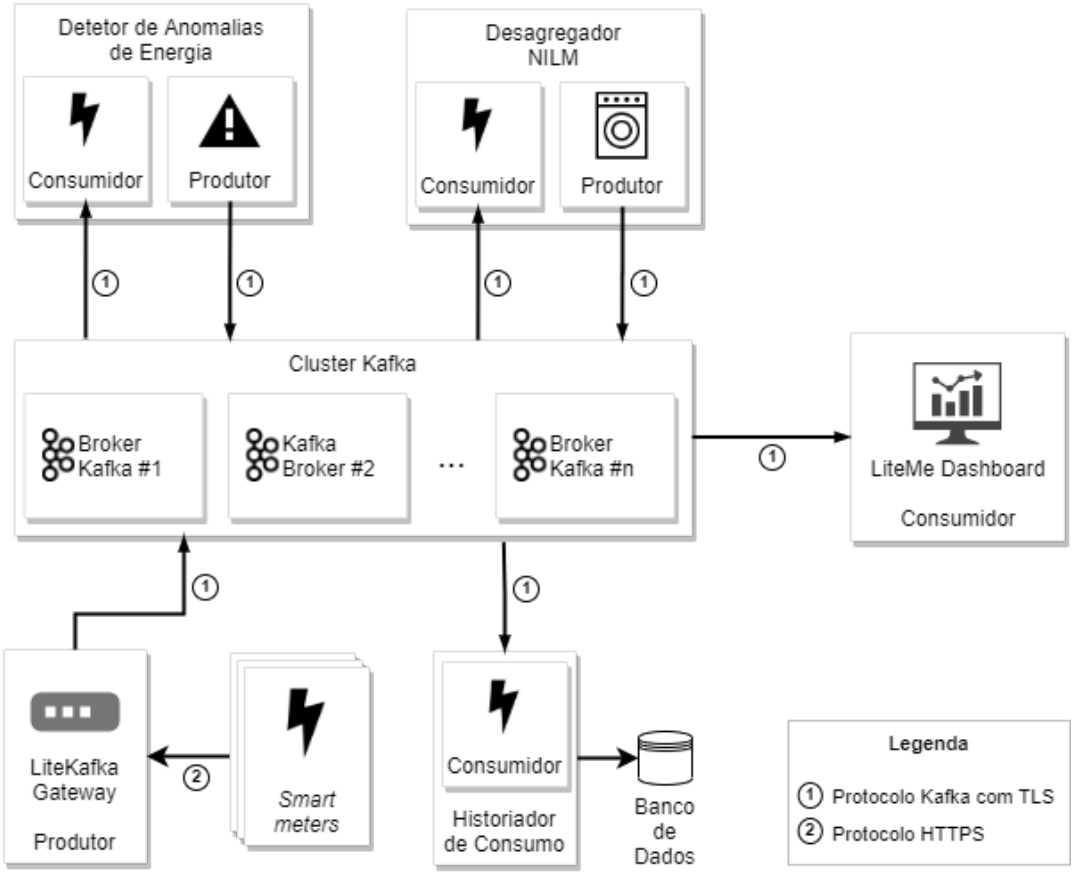

Figura 1.3. Arquitetura da aplicação LiteCampus

- **Desagregador NILM**: Possui a função de identificar, dado um perfil de consumo, quais equipamentos estão consumindo energia naquele instante através da aplicação de uma técnica conhecida por monitoramento não-intrusivo de carga (do inglês, *Non-Intrusive appliance Load Monitoring*, NILM [Armel et al., 2013]). Este componente é assinante do tópico de medições, publicadas pelo LiteKafka Gateway, e publica as decomposições em um tópico específico.
- **Historiador de Consumo**: Persiste medições para análises posteriores, como por exemplo, treinamento de modelos de detecção de anomalia ou de cargas. Este componente é assinante do tópico de medições, publicadas pelo LiteKafka Gateway e poderia armazenar os dados em banco de dados ou em sistemas de arquivos para processamento em *batch* (como HDFS). Sendo este componente seguro, poderia também anonimizar os dados sensíveis antes da exportação.
- **LiteMe Dashboard**: Interage com o usuário final, por exemplo, exibindo alertas. Podendo ser baseado em tecnologias abertas (como Grafana[12]), proprietárias (como PowerBI[13]), ou desenvolvidas para o ambiente de uso.

[12] https://grafana.com/
[13] https://powerbi.microsoft.com/pt-br/

O uso de componentes como o desagregador NILM para gerar alertas e recomendações gera um grande potencial de economia [Armel et al, 2013], mas também um risco de privacidade [Barbosa, Brito e Almeida, 2016]. Informações de quando e quais equipamentos foram usados revelam os hábitos detalhados das pessoas naquele ambiente ou dos processos industriais e, portanto, devem ser protegidos.

Para o caso de uso explorado neste trabalho, simplificamos a aplicação acima da seguinte forma: (1) os *Smart Meters* e LiteKafka Gateway serão combinados em um simulador de medições que produzirá dados diretamente no Kafka; (2) ao invés de um *cluster* Kafka, utilizaremos um único nó e não usaremos conexões TLS mutuamente autenticadas ou configurações de controle de acesso aos tópicos; no entanto, é importante notar que mesmo sem essas configurações de autenticação e autorização, os dados sensíveis estão criptografados com mecanismos sofisticados de compartilhamento de chaves, tornando inútil o acesso de pessoas não autorizadas aos dados no barramento; (3) os componentes de análise, detecção de anomalias, e desagregador NILM, serão substituídos por um componente que consome dados sensíveis e publica alertas simples, como de excesso de demanda ou sobretensão; (4) sendo fora do escopo deste tutorial e para economia de espaço, o dashboard será substituído por clientes de linha de comando do próprio Kafka, que mostrarão os alertas produzidos.

Finalmente, é importante adicionar que esta arquitetura de aplicação não se aplica apenas para dados de energia. Em uma aplicação de monitoramento de segurança, o sensores poderiam ser câmeras e os componentes de análise de dados poderiam realizar operações sensíveis como a contagem ou identificação de pessoas. Já em uma aplicação de trânsito, os dados de localização dos usuários, também sensíveis, seriam coletados pelos smartphones e seriam analisados por diferentes componentes que mediriam níveis de congestionamento, tempos de rotas, etc.

1.3.1. Instalação do Kafka

Nesta seção mostramos uma instalação simples do Kafka, que inclui a criação de certificados e a configuração básica do mesmo, assim como um teste rápido.

Para o nosso caso de uso, os passos para a criação de certificados incluem a criação de uma autoridade certificadora (AC), para que esta assine uma requisição de certificado para o nosso *broker*. Em um sistema fechado, como em uma empresa ou instituição, a AC seria gerada pelos operadores e confiadas por todas as aplicações. Já em um sistema aberto, público, a requisição de certificado seria gerada pelos operadores da aplicação mas assinada por uma AC pública[14]. Os passos para geração da entidade certificadora estão detalhados na Figura 1.4.

```
01 | $ openssl req -new -x509 -keyout ca-key -out ca-cert -days 365
02 | Generating a RSA private key
03 | ...+++++
04 | ..................+++++
05 | writing new private key to 'ca-key'
06 | Enter PEM pass phrase: senha-ca
07 | Verifying - Enter PEM pass phrase: senha-ca
08 | -----
09 | You are about to be asked to enter information that will be incorporated
10 | into your certificate request.
```

[14] https://estrutura.iti.gov.br/

```
 11 | What you are about to enter is what is called a Distinguished Name or a DN.
 12 | There are quite a few fields but you can leave some blank
 13 | For some fields there will be a default value,
 14 | If you enter '.', the field will be left blank.
 15 | -----
 16 | Country Name (2 letter code) [AU]:BR
 17 | State or Province Name (full name) [Some-State]:Paraiba
 18 | Locality Name (eg, city) []:Campina Grande
 19 | Organization Name (eg, company) [Internet Widgits Pty Ltd]:UFCG
 20 | Organizational Unit Name (eg, section) []:LSD
 21 | Common Name (e.g. server FQDN or YOUR name) []:kafka.lsd.ufcg.edu.br
 22 | Email Address []:sbseg2020@lsd.ufcg.edu.br
```

Figura 1.4. Criação da entidade certificadora

Na Figura 1.4, o comando gerará um certificado X.509 chamado *ca-cert* e uma chave privada chamada *ca-key* com validade de um ano. O uso de uma senha de proteção da chave privada evita que um eventual vazamento da mesma (arquivo *ca-key*) comprometa imediatamente a geração de certificados. As outras informações podem ser personalizadas para a aplicação em mãos. O arquivo *ca-cert* pode então ser compartilhado com outros sistemas (entre eles o Kafka), que devem confiar em certificados assinados pela recém-criada AC como veremos a seguir.

Como o Kafka é escrito em Java, a gerência dos certificados de uma aplicação e dos certificados que ela confia é baseado em *Keystores*[15]. Na nossa aplicação, como tipicamente é utilizado, teremos dois armazenamentos do tipo *Keystore*. Um *Keystore* para armazenar os certificados a serem confiados, chamado de *Truststore*, que poderia ser compartilhado por várias aplicações. E um *Keystore* para armazenar as chaves e certificados do Kafka em si. Cada um desses armazenamentos é um arquivo local. A criação do Truststore que confia na nossa AC recém-criada está detalhada na Figura 1.5

```
 01 | $ # Atualiza Truststore
 02 | $ keytool -keystore broker.truststore.jks -alias CARoot -importcert -file
ca-cert
 03 | Enter keystore password:  senha-truststore
 04 | Re-enter new password: senha-truststore
 05 | Owner: EMAILADDRESS=sbseg2020@lsd.ufcg.edu.br, CN=kafka.lsd.ufcg.edu.br,
OU=LSD,
      O=UFCG, L=Campina Grande, ST=Paraiba, C=BR
 06 | Issuer: EMAILADDRESS=sbseg2020@lsd.ufcg.edu.br,CN=kafka.lsd.ufcg.edu.br,
OU=LSD,
      O=UFCG, L=Campina Grande, ST=Paraiba, C=BR
 07 | Serial number: 3fa682094049f344bc026ec673ce3e73ad71c8f8
 08 | Valid from: Tue Sep 15 11:57:51 BRT 2020 until: Wed Sep 15 11:57:51 BRT 2021
… (detalhes do certificado omitidos)
 09 | Trust this certificate? [no]:  yes
 10 | Certificate was added to keystore
```

Figura 1.5. Ensinando o *broker* a confiar na AC recém-criada

Em seguida criamos o *Keystore* para as chaves privadas e certificados do próprio Kafka. A Figura 1.6 mostra dois comandos, um para criação de uma *Keystore* e de uma geração da chave privada já armazenada nela e outro para a exportação de uma solicitação de assinatura para um certificado associado a esta chave privada. Note que este *Keystore* é específico para cada aplicação, neste caso, nosso *broker* Kafka.

[15] https://docs.oracle.com/en/java/javase/15/security/general-security1.html

```
01 | $ # Cria uma Keystore para chaves próprias e gera uma chave privada
02 | $ keytool -keystore broker.keystore.jks -alias kafka -validity 365 -genkey
-keyalg
     RSA -storepass keystore-senha -dname
"cn=kafka.lsd.ufcg.edu.br,ou=LSD,o=UFCG,c=BR"
03 | $ # Agora exportar um certificado com base
04 | $ keytool -keystore broker.keystore.jks -alias kafka -certreq -file
     kafka-cert-file -storepass keystore-senha
```

Figura 1.6. Geração do certificado no *broker* e exportação para assinatura

De posse do certificado ainda não assinado do nosso *broker*, podemos levá-lo para a validação pela AC. No caso de uma aplicação interna, isso seria feito junto à instalação onde foi gerada a AC, como visto na Figura 1.7. No caso de uma aplicação pública, o arquivo *cert-file* da Figura 1.6 seria enviado para a AC pública, que poderia exigir outras comprovações que o requerente é quem diz ser.

```
01 | $ openssl x509 -req -CA ca-cert -CAkey ca-key -in kafka-cert-file -out
     kafka-cert-signed -days 365 -CAcreateserial -passin pass:senha-ca
```

Figura 1.7. Assinatura do certificado pela AC

Com o certificado pronto, importamos ele de volta na *Keystore* do *broker*, como visto na Figura 1.8. Antes do certificado assinado (arquivo *cert-signed*), importamos o certificado da própria AC (arquivo *ca-cert*).

```
01 | $ keytool -keystore broker.keystore.jks -alias CARoot -importcert -file ca-cert
     -storepass keystore-senha
02 | Owner: EMAILADDRESS=sbseg2020@lsd.ufcg.edu.br, CN=kafka.lsd.ufcg.edu.br,
OU=UFCG,
     O=LSD, L=Campina Grande, ST=Paraiba, C=BR
03 | Issuer: EMAILADDRESS=sbseg2020@lsd.ufcg.edu.br, CN=kafka.lsd.ufcg.edu.br,
OU=UFCG,
     O=LSD, L=Campina Grande, ST=Paraiba, C=BR
04 | Serial number: 7f9cb40754386489be49c353697046b5f77e86d8
05 | Valid from: Tue Sep 15 14:17:00 BRT 2020 until: Wed Sep 15 14:17:00 BRT 2021
06 | Certificate fingerprints:
     … (detalhes do certificado omitidos)
07 | Trust this certificate? [no]:  yes
08 | Certificate was added to keystore
09 | $ # Importando a resposta do certificado assinado
10 | $ keytool -keystore broker.keystore.jks -alias kafka -importcert -file
     kafka-cert-signed -storepass keystore-senha
11 | Certificate reply was installed in keystore
```

Figura 1.8. Inserindo o certificado assinado de volta no Keystore do *broker*

Com os certificados e os armazenamentos *Keystore* configurados, podemos partir para a configuração do Kafka em si. A instalação básica de um Kafka, com apenas um *broker* e sem configurações de autenticação e autorização, é um processo simples e bem documentado. A partir da página *Quickstart*[16] é possível baixar um arquivo compactado com a distribuição mais recente. Um resumo destes passos é mostrado na Figura 1.9.

```
01 | $ tar -xzf kafka_2.13-2.6.0.tgz
02 | $ cd kafka_2.13-2.6.0
03 | $ # Exige Java 8 ou mais recente
04 | $ # Zookeeper é usado para armazenar estado e coordenação entre os brokers
05 | $ bin/zookeeper-server-start.sh config/zookeeper.properties
     … (saída do Zookeeper omitida)
06 | # Em outro terminal, inicie o Kafka
07 | $ bin/kafka-server-start.sh config/server.properties
```

Figura 1.9. Instalação básica do Kafka

[16] https://kafka.apache.org/quickstart

Conforme ilustrado na Figura 1.9, primeiro iniciamos um serviço de coordenação, o Zookeeper[17], que pode ser usado com suas configurações padrão, mas deve ser acessível apenas pelas máquinas que executam os *brokers* Kafka. Em seguida podemos iniciar o teste do Kafka, com os comandos exibidos na Figura 1.10 a partir da máquina onde o Kafka foi instalado e usando a porta padrão (parâmetro *--bootstrap-server localhost:9092*). Criamos um novo tópico chamado *sbseg* e consultamos suas propriedades: ele tem apenas uma partição e não tem replicação. Em seguida, produzimos dois eventos, na forma de linhas de texto. Mais detalhes sobre o funcionamento do Kafka podem ser encontrados na página do *Quickstart*.

```
 01 | $ bin/kafka-topics.sh --create --topic sbseg --bootstrap-server localhost:9092
 02 | Created topic sbseg.
 03 | $ bin/kafka-topics.sh --describe --topic sbseg --bootstrap-server
localhost:9092
 04 | Topic: sbseg    PartitionCount: 1     ReplicationFactor: 1  Configs:
      segment.bytes=1073741824
 05 |          Topic: sbseg   Partition: 0   Leader: 0     Replicas: 0    Isr: 0
 06 | $ # Produzindo alguns eventos pela entrada padrão, parar com Control-C
 07 | $ bin/kafka-console-producer.sh --topic sbseg --bootstrap-server localhost:9092
 08 | >Evento em string número 1
 09 | >Evento em string número 2
```

Figura 1.10. Teste do Kafka (criação de tópico e produção de eventos)

Complementando o teste da Figura 1.10, podemos consumir os eventos gerados, usando o comando da Figura 1.11. Neste exemplo, o parâmetro *from-beginning* define que o consumidor que resgatar todo o histórico de mensagens disponível (e o período padrão de retenção é de 7 dias). Sem estes parâmetros, apenas as mensagens publicadas após a iniciação do consumidor seriam recebidas.

```
 01 | $  bin/kafka-console-consumer.sh --topic sbseg --from-beginning
--bootstrap-server
      localhost:9092
 02 | Evento em string número 1
 03 | Evento em string número 2
 04 | ^C
```

Figura 1.11. Teste do Kafka (consumo de eventos)

Note que todos estes comandos estão sendo executados a partir da máquina onde o Kafka foi instalado. Para execução de outra máquina, o servidor de contato inicial (*bootstrap-server*) deveria ser alterado para um nome igual ao especificado nos arquivos de configuração do Kafka, como visto a seguir, e que seja mapeável por DNS,.

```
 01 | zookeeper.connect=localhost:2181
 02 | security.protocol=SSL
 03 | security.inter.broker.protocol = SSL
 04 | ssl.enabled.protocols=TLSv1.2,TLSv1.1,TLSv1
 05 | listeners=SSL://kafka.lsd.ufcg.edu.br:9092
 06 | ssl.truststore.location=/home/andrey/sbseg/broker.truststore.jks
 07 | ssl.truststore.password=senha-truststore
 08 | ssl.keystore.location=/home/andrey/sbseg/broker.keystore.jks
 09 | ssl.keystore.password=keystore-senha
```

Figura 1.12. Configuração mínima do servidor para TLS (arquivo *server.tls.properties*)

Na Figura 1.12, a linha 1 especifica o endereço do Zookeeper levantado antes. Já as linhas 2-5 especificam o protocolo TLS e o endereço e porta onde o serviço espera

[17] https://zookeeper.apache.org/

conexões. As linhas 6-9 especificam os diretórios dos armazenamentos para os certificados que o Kafka confia (*truststore*) e os próprios que ele serve (*keystore*).

Para conectar com um servidor que serve conexões SSL/TLS, o cliente precisa confiar na AC que emitiu o certificado daquele servidor. A Figura 1.13 mostra a arquivo de configuração para o cliente de linha de comando do Kafka usado antes (Figura 1.11), agora configurado para usar SSL/TLS. Note que especificamos um armazenamento de certificados confiáveis (*truststore*), que é uma cópia do usado para o *broker*. Além disso, especificamos que a conexão deve ser SSL/TLS. Finalmente, a Figura 1.14 mostra o comando usado para o teste. É importante destacar que o nome do servidor na configuração da Figura 1.12 deve ser o mesmo do servidor na Figura 1.14, que deve ser o campo CN (*common name*) do certificado emitido para o *broker* (Figura 1.6) e, por fim, deve ser um nome mapeável pelo DNS (ou pelo arquivo *hosts* local na máquina).

```
01 | ssl.truststore.location=/home/andrey/kafka/broker.truststore.jks
02 | ssl.truststore.password=senha-truststore
03 | security.protocol=SSL
```

Figura 1.13. Teste do Kafka (consumo de eventos)

```
01 | $  bin/kafka-console-consumer.sh --bootstrap-server kafka.lsd.ufcg.edu.br:9092
     --topic sbseg --from-beginning --consumer.config client.properties
02 | Evento em string número 1
03 | Evento em string número 2
```

Figura 1.14. Teste do Kafka (consumo de eventos)

Embora esta configuração básica do Kafka seja um bom ponto de partida, existem várias configurações interessantes e importantes, como replicação, particionamento, tempo de retenção de mensagens, entre outros.

1.4. Computação confidencial

Nesta seção apresentamos conceitos de computação confidencial, incluindo a tecnologia de foco neste mini-curso, Intel SGX. Além disso, apresentamos dois conjuntos de ferramentas para desenvolvimento de aplicações confidenciais baseadas em Intel SGX.

1.4.1. Intel Software Guard Extensions

Intel *Software Guard Extensions* (SGX) é um conjunto de instruções adicionadas e mudanças no acesso à memória adicionados à arquitetura Intel x86 [Costan e Devadas, 2016]. Intel SGX é um ambiente de execução confiável assistido por *hardware* que permite a criação de regiões protegidas e isoladas de memória dentro do espaço de endereçamento de uma aplicação. Tais regiões protegidas são denominadas enclaves e têm seu controle de acesso reforçado pelo processador. Um processador com SGX habilitado verifica as decisões de mapeamento de memória do sistema operacional, garantindo que apenas instruções que pertencem ao código do enclave tenham acesso às páginas de memória protegidas. Além disso, o conteúdo da memória dedicada a um enclave é criptografada pelo processador.

Infelizmente, a área de memória dedicada para criação de enclaves é pequena, quando comparada a quantidade de memória principal disponível em computadores de propósito geral atuais. A região de memória dedicada para o funcionamento do Intel SGX é chamada de Memória Reservada do Processador (PRM, do Inglês *Processor*

Reserved Memory). O tamanho máximo mais comum para a PRM é de 128 MB, e apenas alguns processadores recentes possuem um tamanho máximo de PRM de 256 MB. Como é necessário manter outros metadados, a porção de memória realmente disponível para alocação de enclaves, a Cache de Páginas de Enclaves (EPC, do inglês *Enclave Page Cache*), tem um tamanho em torno de 93 MiB. Em máquinas virtuais, esses limites podem ser diferentes, por exemplo, no Microsoft Azure esses valores são de 28, 56, 112, ou 168 MB [Microsoft, 2020]. Se dados ou códigos carregados por enclaves excederem esse o tamanho da EPC, páginas de memória protegidas são desalojadas e enviadas para a memória regular, processo que aumenta a latência de acesso à memória em ordens de magnitude. Deve-se então tentar manter a memória de trabalho dos enclaves em execução em uma máquina dentro desses limites sob o risco de perdas consideráveis de desempenho [Arnautov, 2016].

Uma característica importante das aplicações SGX é que elas são sempre compostas por ao menos uma parte que não é protegida pelo ambiente de execução confiável, mas pode ter vários enclaves independentes. Como um enclave não consegue fazer chamadas de sistema ao sistema operacional, a porção não confiável da aplicação executa chamadas de entrada e saída, e também é responsável por iniciar os enclaves. De todo modo, os enclaves devem ser construídos de maneira a nunca passar dados sensíveis em texto plano para a parte não confiável da aplicação.

1.4.2. Atestação Remota

Além da proteção memória através de criptografia, que entrega confidencialidade e integridade, Intel SGX também implementa o suporte à atestação remota. O processo de atestação remota permite que uma aplicação desafiante ganhe confiança de que um enclave está executando realmente em uma máquina com Intel SGX habilitado [Costan e Devadas, 2016]. Através deste processo, a aplicação obtém propriedades de segurança importantes sobre a máquina, como se a máquina está com o *firmware* atualizado ou se a funcionalidade de *hyperthreading* está habilitada (o que facilitaria ataques de canal lateral). Ao final do processo de atestação, o desafiante pode verificar as identidades do enclaves: a identidade do enclave e a identidade do assinante.

A identidade do enclave, chamada de **MRENCLAVE**, é o resultado de uma operação de *hash*, utilizando o algoritmo SHA-256, envolvendo um registro de todas as operações realizadas no processo de criação do enclave. Dessa maneira, todo o conteúdo relacionados às páginas de memória de um enclave, incluindo código e *flags* de segurança das páginas, influi na identidade do enclave. Já a identidade do assinante, chamada de **MRSIGNER**, permite a identificação daquele que assina a aplicação SGX.

Atualmente, a Intel provê duas maneiras de realizar o processo de atestação. A primeira, usando o Serviço de Atestação da Intel (IAS, do inglês *Intel Attestation Service*), foca no serviço de verificação de enclaves provido pela própria Intel e utiliza um esquema de assinatura de grupo, que pode prover privacidade e é verificável somente pela Intel. Utilizando o esquema de assinatura EPID (do inglês *Enhanced Privacy ID*) é possível ainda escolher entre dois modos de uso do IAS: um modo que produz *quotes* vinculáveis que permite identificar se duas assinaturas foram geradas por uma mesma plataforma; e um modo que produz *quotes* não vinculáveis que não permite permite essa identificação. A segunda forma de atestação se dá através do uso das

Primitivas de Atestação de Datacenter (DCAP, do inglês *Datacenter Attestation Primitives*). Usar o DCAP é mais flexível e é baseada em assinaturas ECDSA (do inglês, *Elliptic Curve Digital Signature Algorithm*, ou Algoritmo de Assinatura Digital de Curvas Elípticas). Essas primitivas de atestação permitem a construção de serviços de atestação locais e têm como alvo aplicações em ambientes em que não se deseja delegar decisões de confiança para terceiros, como a Intel. Além disso, atestação via DCAP é mais indicada para ambientes onde a latência do acesso à internet precisa ser evitada.

A Figura 1.15 ilustra, em alto nível, o processo de atestação. No passo 1, a aplicação desafiante inicia o processo de atestação para verificar se a aplicação SGX está de fato executando em uma máquina com Intel SGX habilitado. O desafio gerado no passo 1 inclui um número aleatório, para garantir o atualidade da resposta da aplicação SGX. A parte não confiável da aplicação SGX serve de interface para os enclaves durante o processo. Contudo, criptografia e algoritmos de autenticação de mensagens são utilizados para garantir propriedades como integridade e confidencialidade quando necessário. No passo 2, a parte não confiável requisita que o enclave gere um relatório de atestação, passando o número aleatório recebido da aplicação desafiante. O enclave então gera um relatório e um manifesto, e os envia como resposta para a parte não confiável, no passo 3.

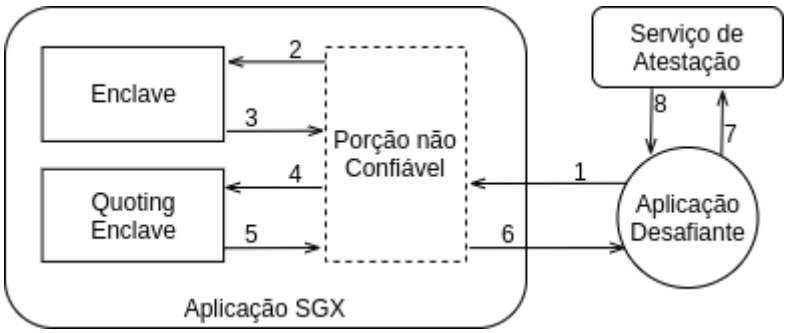

Figura 1.15. Processo de Atestação Remota

No passo 4 da Figura 1.15, a parte não confiável da aplicação entrega os artefatos gerados pelo enclave para um outro enclave, um enclave especial provido pela Intel chamado de *Quoting Enclave*, que tem acesso a chaves de assinatura que nunca são expostas para fora do processador. O relatório é assinado, juntamente com o manifesto, gerando uma estrutura chamada de ***quote***, que é encaminhada para a parte não confiável, no passo 5, que por sua vez, encaminha para a aplicação desafiante, no passo 6. De posse do *quote*, a aplicação desafiante agora pode conferir se deve confiar no conteúdo. Então, no passo 7, a aplicação desafiante verifica junto a um serviço de atestação a validade do *quote* apresentado. A resposta do serviço de atestação, passo 8, inclui um relatório de verificação de atestação (AVR, do inglês *Attestation Verification Report*) que contém informações sobre a segurança da plataforma onde o enclave desafiado diz estar executando. Após verificar que pode confiar no *quote* apresentado pela aplicação SGX, a aplicação desafiante pode verificar a identidade do enclave apresentada no *quote* junto aos seus valores de referência.

1.4.3. Modelo de ameaças

A implementação do ambiente de execução confiável baseado em Intel SGX assume um modelo de ameaça em que um atacante tem controle sobre a máquina com privilégios de superusuário. Assim, um atacante pode controlar toda a pilha de software executando na máquina, incluindo o sistema operacional. O atacante usa então esses poderes para extrair dados, chaves de criptografia ou modificar código das aplicações.

As motivações para tal modelo de ataque são variadas. Em primeiro lugar, a identidade dos operadores podem ser roubadas por um atacante, por exemplo, através de ataques elaborados e direcionados (*spear phishing*). Além disso, operadores ou seus provedores podem ser forçados a fornecer os dados (por exemplo, através de recursos legais). Mesmo se provedor e operadores forem confiáveis, as pilhas de software têm uma base de código muito grande e complexa. Considerando apenas porções do sistema que podem permitir o ganho de privilégios (como o núcleo do Linux em si, o gerente de nuvem, como o OpenStack, ou o virtualizador, como o KVM), ainda assim há dezenas de milhões de linhas de código. Finalmente, subestimar o modelo de ameaças é caro, o custo médio de um vazamento de dados está estimado em US$ 3,86 milhões, com tempo de detecção e contenção médio de 280 dias [IBM Security, 2020].

Assim, assumindo que a infraestrutura e a pilha de software não são confiáveis, nossa base de confiança é reduzida para o software da aplicação, o software de suporte do SGX (por exemplo, as bibliotecas que suportam atestação) e o hardware do SGX em si. Assim como no modelo típico, assumimos que o software do enclaves da aplicação e as bibliotecas de desenvolvimento estão livres de *bugs*. Além disso, assumimos que ataques de canal lateral estão fora do escopo do SGX. Esta consideração tem impactos que dependem de como o software foi desenvolvido. Para código desenvolvido diretamente com o kit de desenvolvimento da Intel (discutido a seguir), o total controle do que é código confiável exige que o desenvolvedor se responsabilize por mecanismos de redução de riscos. Já para código desenvolvido com apoio de um ambiente de execução (*runtime*) como o SCONE, mecanismos implementados no próprio ambiente de execução permitem que riscos sejam mitigados nas aplicações. Como um exemplo, Varys [Oleksenko et al., 2018] mitiga ataques de canal lateral através da detecção de taxas anormais de interrupção na execução do código do enclave e do uso de *threads* irmãs executando simultaneamente.

1.4.4. Preparação do ambiente

Para executar aplicações Intel SGX, é necessário que o ambiente possua o *driver* Intel SGX instalado. As demais dependências necessárias para a execução, como o SGX PSW, já estão embutidas nos contêineres que usaremos. Considerando que o ambiente utilize a distribuição Linux Ubuntu 16.04 ou superior, a instalação do *driver* pode ser feita como ilustrado na Figura 1.16.

```
01 | $ sudo apt-get install build-essential git linux-headers-$(uname -r)
02 | $ git clone https://github.com/intel/linux-sgx-driver.git
03 | $ cd linux-sgx-driver
04 | $ make && sudo make install
05 | $ sudo depmod -a
06 | $ sudo modprobe isgx
```

Figura 1.16 - Passos para instalação do driver SGX

Para verificar se o *driver* foi instalado corretamente, verifique a existência do arquivo de dispositivo */dev/isgx*. Mais informações sobre a instalação do *driver* podem ser encontradas no repositório do GitHub[18]. Uma opção para instalação é utilizando o *script* de instalação disponibilizado pela Scontain[19], criadora do SCONE. Além do *driver* básico, o *script* permite instalar um *driver* que possui extensões que permitem o monitoramento de recursos SGX da máquina.

1.4.5. Desenvolvendo com o Intel SGX SDK

A Intel provê um kit de desenvolvimento de software para que desenvolvedores possam criar aplicações que executam em enclaves SGX, o **Intel SGX SDK**. O SGX SDK permite criar enclaves utilizando as linguagens de programação C e C++. Junto com esse kit de desenvolvimento, a Intel disponibiliza também o SGX PSW (do inglês, *Platform SoftWare*). O PSW contém enclaves especiais como o já mencionado *Quoting Enclave* e o *Launch Enclave*, que permite a criação de novos enclaves. Portanto, para executar aplicações SGX é necessário ter o SGX PSW instalado (além do *driver* SGX), mas para desenvolver, também é necessário ter o SGX SDK.

O desenvolvimento utilizando o SGX SDK tem algumas particularidades. Uma aplicação SGX é dividida em duas porções. Uma porção é a parte confiável da aplicação, que pode ser composta de um ou mais enclaves, e é responsável pela computação sobre os dados sensíveis. A outra porção é a porção não confiável, que executa em nível de usuário comum e é responsável pelas operações que necessitam de chamadas ao sistema operacional, como comunicação com outras aplicações.

É no código da porção não confiável que reside o ponto de início de uma aplicação SGX. Após o início da aplicação, a função *sgx_create_enclave* do enclave deve ser chamada para criação de enclaves. A partir da criação de um enclave, a comunicação entre o enclave criado e a porção não confiável da aplicação deve ser especificada utilizando uma linguagem especial, chamada de linguagem de definição de enclave (EDL, do inglês *Enclave Definition Language*). Utilizando essa linguagem, o desenvolvedor pode definir quais chamadas um enclave pode realizar para a parte não confiável e quais chamadas a parte não confiável pode realizar para um enclave. Quando a chamada é de um enclave para a parte não confiável, ela é denominada ***ocall***, Já quando a chamada é realizada da parte não confiável para o enclave, ela é denominada ***ecall***. A Figura 1.17 ilustra um exemplo de um arquivo EDL que define a interface de comunicação da aplicação "Alô, Mundo" que estudaremos a seguir.

```
 01 | enclave {
 02 |     trusted{
 03 |         public void app_to_enclave([in, string] char *secretIn);
 04 |            public void enclave_to_app([out, size = len] char *secretOut, size_t
len);
 05 |     };
 06 |     untrusted{
 07 |         void print_debug([in, string] char *dbg_message);
 08 |     };
 09 |
 10 | };
```

Figura 1.17. Trecho de Código - EDL: aplicação Alô, mundo!

[18] https://github.com/intel/linux-sgx-driver
[19] https://github.com/scontain/SH

O desenvolvedor deve tomar o cuidado de não expor dados sensíveis através de *ecalls* e *ocalls*. As bibliotecas de criptografia disponíveis no SGX SDK podem ajudar na transferência de dados de maneira segura. Bibliotecas como o *mbedtls-compat-sgx*[20] permitem estabelecer comunicação segura entre enclaves e aplicações remotas.

A seguir criamos uma aplicação SGX simples para entender os principais componentes envolvidos. Como outras aplicações SGX SDK, ela é composta de uma parte não confiável, que chamaremos de *App* e um enclave, que chamaremos de *Enclave*[21]. O *App* é responsável pela criação do enclave e pela intermediação da comunicação de e para o enclave. O *Enclave* receberá os bytes enviados pelo *App* e os guardará em uma variável local, em memória protegida, para entregar de volta em uma consulta futura. Em maneira geral, para o desenvolvedor, os artefatos mais importantes do *Enclave* são o arquivo EDL, que contém a definição da interface entre o enclave e a parte não confiável da aplicação, e os arquivos C/C++ que contém o código responsável por implementar as *ecalls* definidas no EDL. A Figura 1.18 apresenta o código do *App*.

```
01 | #include "enclave_u.h"
02 | #include "sgx_urts.h"
03 | #include <string.h>
04 | #include <stdio.h>
05 |
06 | #define ENCLAVE_NAME "enclave.signed.so"
07 |
08 | void print_debug(char *dbg_message)
09 | {
10 |     printf("%s", dbg_message);
11 | }
12 |
13 | int main()
14 | {
15 |
16 |     sgx_status_t ret = SGX_SUCCESS;
17 |     sgx_launch_token_t launch_token;
18 |     int updated = 0;
19 |     sgx_enclave_id_t eid;
20 |     ret = sgx_create_enclave(
21 |                 ENCLAVE_NAME,//         const char *file_name,
22 |                 SGX_DEBUG_FLAG,//         const int debug,
23 |                 &launch_token,//         sgx_launch_token_t *launch_token,
24 |                 &updated,//         int *launch_token_updated,
25 |                 &eid,//         sgx_enclave_id_t *enclave_id,
26 |                 NULL//         sgx_misc_attribute_t *misc_attr
27 |     );
28 |
29 |     if (ret != SGX_SUCCESS)
30 |     {
31 |         printf("\nUnable to create enclave!\n");
32 |         return -1;
33 |     }
34 |     printf("\nSGX enclave successfully created!\n");
35 |
36 |     char *secretIn = "MyNewSecret";
37 |     ret = app_to_enclave(eid, secretIn);
38 |
39 |     if (ret != SGX_SUCCESS)
40 |     {
41 |         printf("\nUnable to pass secret to enclave!\n");
42 |         return -1;
43 |     }
44 |     printf("\nSuccessfully passed secret to enclave!\n");
45 |
46 |     char *secretOut = (char *) malloc (strlen(secretIn)+1);
```

[20] https://github.com/ffosilva/mbedtls-compat-sgx

[21] O código completo está disponível em https://git.lsd.ufcg.edu.br/lsd-sbseg-2020/alo-mundo-sgx-sdk

```
47 |     ret = enclave_to_app(eid, secretOut, strlen(secretIn)+1);
48 |
49 |     printf("\n%s\n", secretOut);
50 |
51 |     ret = sgx_destroy_enclave(eid);
52 |     if (ret != SGX_SUCCESS)
53 |     {
54 |         printf("\nUnable to destroy enclave!\n");
55 |         return -1;
56 |     }
57 |     printf("\nSGX enclave successfully destroyed!\n");
58 |     return ret;
59 |
60 | }
```

Figura 1.18. Trecho de Código - App: aplicação Alô, mundo!

Na linha 20, o *App* faz uma chamada à função *sgx_create_enclave* para inicializar o *Enclave*. Após a criação do nosso enclave, o *App* então cria uma variável chamada *secretIn* que aponta para o valor "*MyNewSecret*" que vai ser enviado para o *Enclave*, na linha 36. Na linha 37, o *App* invoca a nossa primeira *ecall*: *app_to_enclave*. Esta *ecall* está definida no EDL abaixo, e envia o valor apontado por *secretIn* para o *Enclave*. Em seguida, o *App* invoca nossa segunda *ecall*, camada *enclave_to_app*, para recuperar o valor guardado no enclave SGX. Depois trocar dados com o enclave da aplicação, o *App* faz uma chamada à função *sgx_destroy_enclave*, para destruir o *Enclave*, e então termina a execução.

O EDL que define a interface entre o *Enclave* e o *App* para esta aplicação é o ilustrado na Figura 1.17 e é dividido em duas seções: *trusted* e *untrusted*. Na seção *trusted* são definidas as chamadas ao enclave, *ecalls*. Nas linhas 4 e 5 são declaradas as duas funções do *Enclave* que o *App* utiliza. A declaração das *ecalls* é similar à declaração de funções em C/C++. No entanto, entre colchetes precisamos indicar, no caso de ponteiros, qual a direção do dado e também tamanho do dado em bytes. Em especial, se o tipo de ponteiro for *char* * e o último byte for *0x00*, podemos substituir a indicação de tamanho pelo identificador *string*. Na seção *untrusted* temos a definição de uma *ocall*, chamada de um enclave para a parte não confiável da aplicação, que utilizamos neste exemplo para imprimir o valor recebido dentro do *Enclave*.

Finalmente, a Figura 1.19 ilustra o código do *Enclave* que implementa as interfaces definidas no EDL. Apesar de escrito em C/C++, o código do enclave não pode dispor da biblioteca C padrão. Isso ocorre em função das limitações impostas pela propriedade de isolamento do ambiente de execução confiável.

```
01 | #include "enclave_t.h"
02 | #include "stdlib.h"
03 | #include "string.h"
04 | char *secret;
05 | void app_to_enclave(char *secretIn)
06 | {
07 |     secret = (char *) calloc(strlen(secretIn)+1,1);
08 |     memcpy(secret, secretIn, strlen(secretIn));
09 |     print_debug(secret);
10 | }
11 | void enclave_to_app(char *secretOut, size_t len)
12 | {
13 |     memcpy(secretOut, secret, len);
14 | }
```

Figura 1.19. Trecho de Código - Enclave: aplicação Alô, mundo!

A execução é realizada com ajuda de um script bash que constrói um contêiner e executa a aplicação. É necessário ter o Docker instalado na máquina e também o driver do SGX, que foi mencionado na seção "Preparação do ambiente."

1.4.6. Desenvolvendo com SCONE

Como discutido acima, o uso do Intel SGX SDK reduz severamente sua aplicabilidade para aplicações preexistentes, cujo custo de reescrita (no melhor caso, de alguns funções, caso a aplicação já seja escrita em C/C++, e no pior caso, da aplicação inteira, caso ela seja escrita em outra linguagem) pode ser proibitivo. Desta forma, ambientes de execução foram criados para permitir o uso de código não modificado.

Assim como sistemas como serviços alternativos como Anjuna[22] e Fortanix[23], SCONE[24] (Secure CONtainer Environment) foi criado para permitir que aplicações inteiras e não modificadas rodem dentro de enclaves Intel SGX. SCONE utiliza versões modificadas de bibliotecas do sistema como *musl-libc* ou *glibc*, e um compilador *gcc* especial, o que permite que o código-fonte de aplicações já existentes seja compilado para execução confidencial com a mínima necessidade de modificação por parte do desenvolvedor. SCONE suporta uma variedade de linguagens de programação, como C, C++, Java, Python, Go e JavaScript.

Além disso, SCONE provê uma plataforma para desenvolvimento e implantação de aplicações confidenciais com Intel SGX. Esta plataforma conta com ferramentas de atestação local e remota, geração, compartilhamento e entrega segura de segredos, além de criptografia transparente de arquivos e de tráfego de rede, o que retira do desenvolvedor a responsabilidade de implementar manualmente estas funcionalidades com o SGX SDK.

A seguir são apresentados alguns desses conceitos e funcionalidades, usando como base uma aplicação-exemplo simples escrita na linguagem Python.

1.4.6.1. Alô, mundo!

Para começar, considere um programa Python simples que exibe a mensagem "Alô, mundo!" na tela, *programa.py* (Figura 1.20).

```
01 | print("Alo, mundo!")
```

Figura 1.20. Trecho de Código - *programa.py*: Alô, mundo! na linguagem Python

Para executar essa aplicação simples dentro de um contêiner Docker, é necessária a criação de um arquivo de definição de imagem, ou *Dockerfile* (Figura 1.23).

```
01 | FROM python:3
02 | COPY programa.py .
03 | ENTRYPOINT [ "python3", "programa.py" ]
```

Figura 1.21. Trecho de Código - *Dockerfile*: aplicação Alô, mundo!

Para construir a imagem e executar o contêiner, basta executar os comandos descritos na Figura 1.22.

[22] https://www.anjuna.io/microsoft-azure
[23] https://fortanix.com/products/enclave-manager/
[24] https://sconedocs.github.io/

```
01 | $ docker build . -t sbseg-alo-mundo
02 | Sending build context to Docker daemon  3.072kB
03 | Step 1/3 : FROM python:3
04 |  ---> 28a4c88cdbbf
05 | Step 2/3 : COPY programa.py .
06 |  ---> 33ae2928e067
07 | Step 3/3 : ENTRYPOINT [ "python3", "programa.py" ]
08 |  ---> Running in 1744daef5b94
09 | Removing intermediate container 1744daef5b94
10 | ---> b050e55a0823
11 | Successfully built b050e55a0823
12 | Successfully tagged sbseg-alo-mundo:latest
13 |
14 | $ docker run -it --rm sbseg-alo-mundo
15 | Alo, mundo!
```

Figura 1.22. Criação e execução do contêiner Alô, mundo!

Para construir executar a mesma aplicação dentro de um enclave SGX com SCONE, basta utilizar um interpretador Python seguro. Por conveniência, SCONE oferece um conjunto de imagens pré-compiladas de aplicações populares, entre elas o interpretador Python. Dessa forma, para executarmos a mesma aplicação em um enclave, basta modificarmos a imagem-base no *Dockerfile* (instrução *FROM*, na linha 1). A Figura 1.23 mostra a *scone.Dockerfile* utilizada.

```
01 | FROM sconecuratedimages/public-apps:python-3.7.3-alpine3.10
02 | COPY programa.py .
03 | ENTRYPOINT [ “python3”, “programa.py” ]
```

Figura 1.23. Trecho de Código - *scone.Dockerfile*: aplicação Alô, mundo!

Com SCONE, a construção e execução da imagem é similar, como ilustrado na Figura 1.24. No entanto, agora a aplicação necessita acessar o *driver* de Intel SGX, o que pode ser feito pelo parâmetro *--device* do Docker (assume-se que o *driver* está instalado e disponibiliza uma interface em */dev/isgx*). O MRENCLAVE da aplicação pode ser obtido ao executá-la com a variável de ambiente *SCONE_HASH=1* definida.

```
01 | $ docker build . -t sbseg-alo-mundo-scone -f scone.Dockerfile
02 | Sending build context to Docker daemon  4.096kB
03 | Step 1/3 : FROM sconecuratedimages/public-apps:python-3.7.3-alpine3.10
04 |  ---> 7985d7286c75
05 | Step 2/3 : COPY programa.py .
06 |  ---> 801811ae27ea
07 | Step 3/3 : ENTRYPOINT [ "python3", "programa.py" ]
08 |  ---> Running in 350ad362fb0d
09 | Removing intermediate container 350ad362fb0d
10 |  ---> 2622e5882129
11 | Successfully built 2622e5882129
12 | Successfully tagged sbseg-alo-mundo-scone:latest
13 |
14 | $ docker run -it --rm --device /dev/isgx sbseg-alo-mundo-scone
15 | Alo, mundo!
16 |
17 | $ docker run -it --rm --device /dev/isgx -e SCONE_HASH=1 sbseg-alo-mundo-scone
18 | 41f0117a3c62966b48ef6e2388b5fe7ff719b1f48abbf417e855fff0546a8e0d
```

Figura 1.24. Criação, execução e obtenção de MRENCLAVE do Alô, mundo! no SCONE

Como a aplicação inteira é executada dentro de um enclave Intel SGX, SCONE gerencia a comunicação com o sistema operacional (que não é confiável, de acordo com seu modelo de ameaça) através de filas e *threads* especiais que gerenciam as chamadas de sistema requisitadas pela aplicação, numa abordagem assíncrona. Note que, devido à limitação na quantidade de memória protegida, ou EPC, disponível, é comum que as

aplicações necessitem paginar seu conteúdo para a memória principal, o que incorre em perda de desempenho, como discutido na Seção 1.4.1.

SCONE também oferece, como mencionado anteriormente, uma plataforma que entrega ao desenvolvedor mais controle sobre a execução segura de suas aplicações, através de mecanismos como atestação remota e entrega de segredos. Essa plataforma, bem como algumas de suas funcionalidades, são exploradas nas seções a seguir.

1.4.6.2. Atestação remota

Para entender como funciona a atestação remota no SCONE, é necessário apresentar dois componentes essenciais: o **CAS** (do inglês, *Configuration and Attestation Service*) e o **LAS** (do inglês, *Local Attestation Service*). O CAS é o responsável por atestar uma aplicação SCONE. Uma vez que uma aplicação é atestada, o CAS pode provisionar segredos e configurações de forma segura. O LAS, por sua vez, é o agente responsável pela atestação local da aplicação SGX, gerando um *quote* de atestação que é, então, enviado ao CAS. O *quote*, como mencionado na Seção 1.4.2, contém informações do enclave a ser atestado (como MRENCLAVE e MRSIGNER) e da plataforma (incluindo dados sobre funcionalidades que influenciam a atestação, como a versão do *firmware* e se o *hyperthreading* está habilitado). Se o *quote* for o esperado, o CAS então finaliza o processo de atestação, provisionando também segredos e configurações.

O CAS é o ponto central da arquitetura de atestação, sendo também o local de armazenamento de segredos e configurações. Um CAS pode servir múltiplas aplicações. Novas aplicações são registradas no CAS por meio de sua API, através de manifestos no formato YAML, onde a identidade de enclave esperada é descrita, e segredos e configurações são definidos. Esses manifestos são chamados de arquivos de sessão. O LAS, por sua vez, serve apenas para o nó em que ele está sendo executado, uma vez que se comunica diretamente com o *hardware* para efetuar a atestação local do enclave. Assim, é necessário executar um LAS por nó.

A Figura 1.25 ilustra como iniciar um CAS e um LAS localmente em contêineres Docker. O LAS expõe a porta 18766 para a atestação local de enclaves. O CAS, por sua vez, expõe duas portas: 8081, para submissão de sessões, e 18765, para atestação remota de enclaves. Note-se que o CAS também se atesta localmente, razão pela qual é definida a variável de ambiente *SCONE_LAS_ADDR*, cujo valor, *172.17.0.1*, refere-se à interface de rede padrão do Docker.

```
 01 | $ docker run -dt --rm --name las --device /dev/isgx -p 18766:18766
      sconecuratedimages/services:las
 02 | $ docker run -dt --rm --name cas --device /dev/isgx -p 18765:18765 -p 8081:8081
-e
      SCONE_LAS_ADDR=172.17.0.1 sconecuratedimages/services:cas
```

Figura 1.25. Iniciando CAS e LAS localmente via Docker

```
01 | name: sessao-exemplo
02 | version: 0.3
03 | services:
04 |    - name: alo-mundo
05 |                                                                    mrenclaves:
[41f0117a3c62966b48ef6e2388b5fe7ff719b1f48abbf417e855fff0546a8e0d]
06 |      command: python3 programa.py
07 |      pwd: /
08 |      environment:
09 |           SCONE_MODE: "hw"
```

```
10 |           VARIAVEL_SECRETA: "conteudosecreto"
11 | security:
12 |   attestation:
13 |     tolerate: [debug-mode, hyperthreading, outdated-tcb]
14 |     ignore_advisories: "*"
```

Figura 1.26. Trecho de Código - *sessao.yml*: exemplo de arquivo de sessão SCONE

A Figura 1.26 mostra um exemplo simples de arquivo de sessão SCONE. A sessão *sessao-exemplo* (linha 1) é criada e define um único *service* (linhas 4 a 10). Em SCONE, um *service* equivale a uma instância de aplicação. Mais especificamente, a aplicação *python3* (linha 6) está sendo registrada, e seu MRENCLAVE esperado é *41f0117a3c62966b48ef6e2388b5fe7ff719b1f48abbf417e855fff0546a8e0d* (linha 5). Note que o MRENCLAVE, ou identidade de enclave, define a aplicação. Alterações no código resultam em um MRENCLAVE completamente diferente. Caso uma aplicação possua um MRENCLAVE diferente do esperado, o processo de atestação é abortado e a aplicação é finalizada sem receber nenhum segredo ou configuração do CAS. Argumentos da aplicação (linha 6) e variáveis de ambiente (linhas 8 a 10) também são protegidos, apenas sendo entregues após o processo de atestação ser completado com sucesso. Por padrão, o CAS inicia em modo de produção, e não tolera vulnerabilidades no *hardware* onde o enclave está sendo executado. Estas vulnerabilidades, que são descritas no *quote* do processo de atestação, englobam *firmwares* defasados, a presença de *hyperthreading*, enclaves em modo *debug* (que podem ter seu conteúdo inspecionado), entre outros. Para fins de teste e desenvolvimento, é possível relaxar os requisitos do CAS através do campo *security* (linhas 11 a 14). Nela é possível definir não só quais vulnerabilidades são toleradas (no campo *tolerate*), como quais recomendação de atualização de *firmware* da Intel (ou *Intel Update Advisories)* podem ser ignoradas. A lista completa de vulnerabilidades de *hardware* para atestação pode ser encontrada na documentação do SCONE.

No caso de aplicações compiladas, como as escritas em linguagens como C, C++ e Go, o MRENCLAVE identifica o arquivo binário executável da aplicação em si. Em linguagens interpretadas, o MRENCLAVE identifica o interpretador. No exemplo da Figura 1.28, portanto, o MRENCLAVE apenas diz se o interpretador foi ou não modificado, sem oferecer garantias sobre o código sendo interpretado. Para contornar essa limitação, SCONE permite a atestação e criptografia de código-fonte (e arquivos em geral) através de uma funcionalidade chamada de FSPF (do inglês, *FileSystem Protection File*). A próxima seção aborda FSPF e outras funcionalidades do SCONE, como geração e entrega de segredos e criptografia transparente.

O CAS oferece uma API para criação e gerenciamento de sessões, disponível na porta *8081*. A Figura 1.27 mostra como obter certificados para contactar um CAS localizado num contêiner Docker local, enquanto a Figura 1.28 mostra como enviar uma nova sessão.

```
01 | $ cat > clientcertreq.conf <<EOF
02 | [req]
03 | [req]
04 | distinguished_name = req_distinguished_name
05 | x509_extensions = v3_req
06 | prompt = no
07 |
08 | [req_distinguished_name]
```

```
09 | C = EU
10 | ST = Germany
11 | L = Dresden
12 | O = Scontain
13 | OU = CLI
14 | CN = cli.scontain.com
15 | [ v3_req ]
16 |
17 | # Extensions to add to a certificate request
18 | basicConstraints = CA:FALSE
19 | keyUsage = nonRepudiation, digitalSignature, keyEncipherment
20 | subjectAltName = @alt_names
21 |
22 | [alt_names]
23 | DNS.1 = cli.scontain.com
24 | DNS.2 = scone-cli.scontain.com
23 | EOF
24 |
25 | $ openssl req -newkey rsa:4096 -days 365 -nodes -x509 -out client.pem -keyout
     client-key.pem -config clientcertreq.conf
```

Figura 1.27. Obtendo certificados para CAS via *openssl*

```
01 | $ export SCONE_CAS_ADDR=172.17.0.1
02 | $ curl -v -k -s --cert client.pem  --key client-key.pem  --data-binary
@sessao.yml -X POST https://$SCONE_CAS_ADDR:8081/session
```

Figura 1.28. Criando uma nova sessão em um CAS local. Para contactar outro CAS, basta mudar o valor de *SCONE_CAS_ADDR*. O arquivo de sessão é *sessao.yml*

Para executar uma aplicação SCONE com atestação remota, é necessário definir as variáveis de ambiente *SCONE_CAS_ADDR*, *SCONE_LAS_ADDR* e *SCONE_CONFIG_ID*, que apontam, respectivamente, para o endereço do CAS, o endereço do LAS, e o nome da sessão e do *service*, no formato *SESSÃO/SERVIÇO*. A aplicação da Figura 1.26, por exemplo, seria referenciada com *SCONE_CONFIG_ID=sessao-exemplo/alo-mundo*.

1.4.6.3. Segredos

Uma vez atestada, a aplicação pode receber segredos e configurações do CAS de forma segura. Segredos são definidos no arquivo de sessão do SCONE, no campo *secrets*, que permite definir segredos de vários tipos, como texto, binário e até certificados e chaves privadas. O desenvolvedor tem a opção de não especificar o conteúdo dos segredos, o que fará com que o CAS gere segredos de conteúdo aleatório. Uma vez criados, segredos podem ser referenciados pelas aplicações definidas no arquivo de sessão (por exemplo, podem ser injetados no ambiente ou em arquivos), e mesmo exportados para outros arquivos de sessão, inclusive para outro CAS.

```
01 | name: sessao-exemplo-segredos
02 | version: 0.3
03 |
04 | services:
05 |    - name: alo-mundo
06 |                                                                    mrenclaves:
[41f0117a3c62966b48ef6e2388b5fe7ff719b1f48abbf417e855fff0546a8e0d]
07 |      command: python3 programa.py
08 |      pwd: /
09 |      image_name: alo-mundo
10 |      environment:
11 |           SCONE_MODE: hw
12 |           UM_SEGREDO: $$SCONE::segredo1$$
13 | images:
14 |    - name: alo-mundo
```

```
15 |     injection_files:
16 |          - path: /etc/segredo.txt
17 |            content: $$SCONE::segredo2$$
18 | secrets:
19 |   - name: segredo1
20 |     kind: ascii
21 |     size: 16
22 |   - name: segredo2
23 |     kind: ascii
24 |     value: "isto eh um segredo!!!"
25 | security:
26 |   attestation:
27 |     tolerate: [debug-mode, hyperthreading, outdated-tcb]
28 |     ignore_advisories: "*"
```

Figura 1.29. Trecho de Código - *sessao-segredos.yml*: exemplo de arquivo de sessão SCONE com segredos

Na Figura 1.29, o arquivo de sessão do *Alô, mundo!* é estendido para incluir dois segredos do tipo *ascii*, ou seja, texto. O segredo *segredo1* (linhas 19 a 21), não possui seu conteúdo definido, o que fará com que o CAS gere um texto aleatório de 16 bytes. O segredo *segredo2* (linhas 22 a 24), por sua vez, tem seu conteúdo definido: "isto é um segredo!!!". Estes segredos são então injetados no ambiente (linha 12) e no sistema de arquivos (linhas 15 a 17) da aplicação. A notação *$$SCONE::NOME$$* serve para referenciar segredos e seu conteúdo ao longo do arquivo de sessão, onde *NOME* é o nome do segredo. A aplicação *Alô, mundo!* pode ser modificada para exibir também o conteúdo desses dois segredos (Figura 1.30).

```
01 | import os
02 | print("Alo, mundo!")
03 | print("UM_SEGREDO: %s" % os.environ.get("UM_SEGREDO"))
04 | arquivo = open("/etc/segredo.txt", "r")
05 | print(arquivo.read())
06 | arquivo.close()
```

Figura 1.30. Trecho de Código - programa.py: Alô, mundo! com segredos

A construção da aplicação não sofre alterações, e a mesma *scone.Dockerfile* pode ser utilizada. A execução da aplicação sem as variáveis de atestação do SCONE ocasionará um erro, já que nem a variável de ambiente *UM_SEGREDO*, nem o arquivo */etc/segredo.txt* existirão a menos que a aplicação seja atestada (Figura 1.31).

```
01 | $ docker build . -t sbseg-alo-mundo-scone-segredos -f scone.Dockerfile
02 | Sending build context to Docker daemon  4.096kB
03 | Step 1/3 : FROM sconecuratedimages/public-apps:python-3.7.3-alpine3.10
04 |  ---> 179f05bee7c7
05 | Step 2/3 : COPY programa.py .
06 |  ---> 9129296afc6d
07 | Step 3/3 : ENTRYPOINT [ "python3", "programa.py" ]
08 |  ---> Running in 72a81ea76a51
09 | Removing intermediate container 72a81ea76a51
10 |  ---> dd7f07ee4214
11 | Successfully built dd7f07ee4214
12 | Successfully tagged sbseg-alo-mundo-scone-segredos:latest
13 |
14 | $ docker run -it --rm --device /dev/isgx sbseg-alo-mundo-scone-segredos
15 | Alo, mundo!
16 | UM_SEGREDO: None
17 | Traceback (most recent call last):
18 |   File "programa.py", line 4, in <module>
19 |     arquivo = open("/etc/segredo.txt", "r")
20 | FileNotFoundError: [Errno 2] No such file or directory: '/etc/segredo.txt'
21 |
22 | $ docker run -it --rm --device /dev/isgx \
23 |          -e SCONE_CAS_ADDR=$SCONE_CAS_ADDR \
24 |          -e SCONE_LAS_ADDR=$SCONE_LAS_ADDR \
25 |          -e SCONE_CONFIG_ID=sessao-exemplo-segredos/alo-mundo \
```

```
26 |          sbseg-alo-mundo-scone-segredos
27 | Alo, mundo!
28 | UM_SEGREDO: ^/Z/!Cm1D&Q84BP'
29 | isto eh um segredo!!!
```

Figura 1.31. Criação, execução e obtenção de MRENCLAVE do *Alô, mundo!* no SCONE

1.4.6.4. FSPF e volumes

Outra funcionalidade oferecida pelo SCONE é atestação e criptografia do sistema de arquivos, através de SCONE FSPF (*FileSystem Protection File*) e volumes. FSPF permite ao desenvolvedor criar regiões do sistema de arquivos que podem ser autenticadas e criptografadas. Uma das vantagens é que o gerenciamento de chaves e o processo de criptografia são feitos pelo *runtime* do SCONE, com o auxílio do CAS, o que torna a criptografia transparente para as aplicações. Por exemplo, quando uma aplicação SCONE tenta ler um arquivo numa região protegida por FSPF, o *runtime* intercepta a chamada de sistema de leitura do arquivo e, caso a aplicação tenha sido atestada, recebe as chaves do CAS que permitem a descriptografia do arquivo automaticamente dentro do enclave. Dessa forma, aplicações não precisam ser modificadas para lidar com criptografia. Essa funcionalidade é particularmente útil no caso de aplicações escritas em linguagens de programação interpretadas (Python, JavaScript), já que o código-fonte (e bibliotecas) pode ser criptografado, sendo lido apenas pelo interpretador autorizado.

Para começar a definir regiões protegidas por FSPF, é necessário criar o arquivo FSPF em si. Para isso, utiliza-se a interface de linha de comando do SCONE (*scone-cli*). Após a criação do arquivo FSPF, é possível definir regiões, que podem ser autenticadas (*authenticated*) ou criptografadas (*encrypted*). Regiões autenticadas têm a integridade do conteúdo verificada, o que permite detectar modificações não-autorizadas. Contudo, o conteúdo dos arquivos pode ser lido por aplicações fora do enclave. Regiões criptografadas têm seu conteúdo criptografado, então aplicações fora do enclaves não podem acessar seu conteúdo. Regiões criptografadas são também autenticadas.

Para incluir o FSPF no processo de atestação remota feito pelo CAS é necessário criptografar o arquivo de FSPF em si, também através da *scone-cli*, que então gera uma chave (*key*) e um código de autenticação de mensagem (*tag* ou MAC, do inglês *Message Authentication Code*), essenciais para acessar o arquivo de FSPF e, por consequência, as regiões e arquivos. Por fim, *key* e *tag* são adicionados no arquivo de sessão, permitindo que regiões e arquivos sejam acessados apenas por aplicações atestadas. Caso a região tenha persistência ativada (*--kernel* na criação da região), escritas efetuadas nessas regiões são automaticamente persistidas pelo CAS, e uma nova *tag* é gerada para o novo estado do sistema de arquivos. Caso a persistência esteja desativada (*--ephemeral*), modificações não serão persistidas. A criação de FSPF através da *scone-cli* pode ser usando um cliente instalado localmente na máquina ou através de uma imagem Docker.

```
01 | scone fspf create /fspf/volume.fspf
02 | scone fspf addr /fspf/volume.fspf / --not-protected --kernel /
03 | scone fspf addr /fspf/volume.fspf /app --encrypted --kernel /app
```

```
04 | scone fspf addf /fspf/volume.fspf /app /native-files /app
05 | scone fspf encrypt /fspf/volume.fspf > /native-files/keytag
```

Figura 1.32. Trecho de Codigo - *fspf.sh*: Criação de FSPF e regiões criptografadas

O arquivo da Figura 1.32 mostra uma sequência de comandos executados em um contêiner de *scone-cli* para a criação de um arquivo FSPF em */fspf/volume.fspf* (linha 1). A criação de regiões *(scone fspf addr*) e adição de arquivos (*scone fspf addf*), ilustradas nos comandos seguintes, sempre se referem a um arquivo de FSPF, que é o primeiro argumento desses comandos. Ao FSPF são adicionadas duas regiões: */* (linha 2) e */app* (linha 3). A região */*, a raiz do sistema de arquivos, não é protegida (*--not-protected*) por FSPF. A região */app*, por sua vez, é criptografada (*--encrypted*), ou seja, todos os arquivos adicionados a */app* serão criptografados automaticamente. Ambas as regiões são persistidas pelo SCONE (*--kernel*). A linha 4 mostra a adição de arquivos à região */app*. Os arquivos originais estão localizados em */native-files* e os arquivos criptografados resultantes serão adicionados ao diretório */app* do sistema de arquivos. Por fim, o arquivo de FSPF é criptografado, e a chave e *tag* necessárias para acessar o FSPF são escritas no arquivo */native-files/keytag*, podendo ser depois adicionadas a um arquivo de sessão SCONE.

A Figura 1.33 ilustra como executar o arquivo acima e, assim, criar o FSPF, através da imagem Docker de *scone-cli*. Os arquivos a serem criptografados estão no diretório *native-files* local, e são injetados no contêiner de *scone-cli* via volumes Docker (*-v native-files:/native-files*). Os arquivos criptografados serão salvos no diretório *encrypted-files* local (*-v encrypted-files:/encrypted-files*), criado antes de executar o contêiner. Por fim, o arquivo de FSPF, *volume.fspf* será persistido no diretório local (*-v $PWD:/fspf-file*). Os comandos de criação estão no arquivo *fspf.sh*, que será executado pelo contêiner de *scone-cli*. O arquivo *programa.py*, da aplicação *Alô, mundo!* será usado como exemplo.

```
01 | $ mkdir fspf native-files encrypted-files
02 | $ cp programa.py native-files/
03 | $ chmod +x fspf.sh
04 | $ cp fspf.sh fspf/
05 |
06 | $ docker run -it --rm --device /dev/isgx \
07 |          -v $PWD/fspf:/fspf \
08 |          -v $PWD/native-files:/native-files \
09 |          -v $PWD/encrypted-files:/app \
10 |          sconecuratedimages/sconecli:alpine3.7-scone4.2.1 \
11 |          bash -c /fspf/fspf.sh
12 |
13 | $ cat encrypted-files/programa.py
14 | ��ڻ�zoA!�^�
```

Figura 1.33. Usando a imagem Docker de *scone-cli* para criação de FSPF com região criptografada

Uma vez criado o FSPF, basta verificar a chave e *tag* em */native-files/keytag* e incluí-las no arquivo de sessão, através dos parâmetros *fspf_path*, *fspf_key* e *fspf_tag* (substituir *$SCONE_FSPF_KEY* e *$SCONE_FSPF_TAG*). O arquivo de sessão da Figura 1.34 ilustra uma sessão que considera FSPF para o *service alo-mundo*.

```
01 | name: sessao-exemplo-fspf
02 | version: 0.3
03 | services:
04 |   - name: alo-mundo
```

```
05           |                                                                mrenclaves:
[41f0117a3c62966b48ef6e2388b5fe7ff719b1f48abbf417e855fff0546a8e0d]
06 |     command: python3 /app/programa.py
07 |     image_name: alo-mundo
08 |     pwd: /
09 |     environment:
10 |       SCONE_MODE: hw
11 |       UM_SEGREDO: $$SCONE::segredo1$$
12 |     fspf_path: /fspf/volume.fspf
13 |     fspf_key: $SCONE_FSPF_KEY
14 |     fspf_tag: $SCONE_FSPF_TAG
15 | images:
16 |   - name: alo-mundo
17 |     injection_files:
18 |     - path: /etc/segredo.txt
19 |       content: $$SCONE::segredo2$$
20 | secrets:
21 |    - name: segredo1
22 |      kind: ascii
23 |      size: 16
24 |    - name: segredo2
25 |      kind: ascii
26 |      value: "isso eh um segredo"
27 | security:
28 |   attestation:
29 |     tolerate: [debug-mode, hyperthreading, outdated-tcb]
30 |     ignore_advisories: "*"
```

Figura 1.34. Trecho de Código - *sessao-fspf.yml*: exemplo de arquivo de sessão SCONE com FSPF

Como ilustrado na Figura 1.35, é necessário enviar o novo arquivo de sessão para o CAS, e reescrever a definição de imagem Docker, *scone.Dockerfile*, para considerar o novo arquivo Python criptografado, bem como o arquivo FSPF.

```
01 | FROM sconecuratedimages/public-apps:python-3.7.3-alpine3.10
02 | COPY encrypted-files/programa.py /app/programa.py
03 | COPY volume.fspf /fspf-file/volume.fspf
04 | ENTRYPOINT [ “python3”, “/app/programa.py” ]
```

Figura 1.35. Trecho de Código - scone.Dockerfile: aplicação Alô, mundo! com FSPF e código criptografado

A Figura 1.36 ilustra a construção e execução da nova imagem. Note que agora *SCONE_CONFIG_ID=sessao-exemplo-fspf/alo-mundo* e a tentativa de execução não atestada resulta em erro pois o interpretador Python não consegue compreender o arquivo *programa.py* criptografado. Uma vez atestado, o *runtime* do SCONE descriptografa o arquivo transparentemente, e a aplicação executa da forma esperada.

```
 01 | $ docker build . -t sbseg-alo-mundo-scone-fspf -f scone.Dockerfile
 02 | Sending build context to Docker daemon  23.55kB
 03 | Step 1/4 : FROM sconecuratedimages/public-apps:python-3.7.3-alpine3.10
 04 |  ---> 179f05bee7c7
 05 | Step 2/4 : COPY encrypted-files/programa.py /app/programa.py
 06 |  ---> e9af77581bbd
 07 | Step 3/4 : COPY fspf/volume.fspf /fspf/volume.fspf
 08 |  ---> 80b9d856016e
 09 | Step 4/4 : ENTRYPOINT [ "python3", "/app/programa.py" ]
 10 |  ---> Running in 709b8c8a65ec
 11 | Removing intermediate container 709b8c8a65ec
 12 |  ---> a24945984a80
 13 | Successfully built a24945984a80
 14 | Successfully tagged sbseg-alo-mundo-scone-fspf:latest
 15 |
 16 | $ docker run -it --rm --device /dev/isgx sbseg-alo-mundo-scone-fspf
 17 | File "/app/programa.py", line 1
 18 | SyntaxError: Non-UTF-8 code starting with '\xc1' in file /app/programa.py on
line
```

```
      1, but no encoding declared; see http://python.org/dev/peps/pep-0263/ for
details
 19 |
 20 | $ docker run -it --rm --device /dev/isgx \
 21 |          -e SCONE_CAS_ADDR=$SCONE_CAS_ADDR \
 22 |          -e SCONE_LAS_ADDR=$SCONE_LAS_ADDR \
 23 |          -e SCONE_CONFIG_ID=sessao-exemplo-fspf/alo-mundo \
 24 |          sbseg-alo-mundo-scone-fspf
 25 | Alo, mundo!
 26 | UM_SEGREDO: Nrj05qjgqHjDlYJd
 27 | isto é um segredo!!!
```

Figura 1.36. Construção e execução da aplicação Alô, mundo! com código criptografado por SCONE FSPF

Volumes[25] por sua vez, permitem definir regiões criptografadas do sistema de arquivos que podem ser montadas no sistema de arquivos de aplicações SCONE. Um volume (*volume*, no arquivo de sessão) pode ser compartilhado entre várias aplicações, ou mesmo importado por aplicações definidas em sessões diferentes. Neste caso, é possível controlar que tipo de modificações uma outra sessão pode realizar no sistema de arquivos. Volumes têm uma única região criptografada, e também podem ter uma chave e uma *tag* de FSPF definida pelo usuário. Caso uma chave e *tag* não sejam definidas, o volume é inicializado vazio, e o CAS gerencia as novas chave e *tag* de forma transparente. Para utilizar volumes, é necessário associá-los a uma imagem (*image*, no arquivo de sessão), que pode então ser referenciada pelas aplicações.

1.5. Computação Confidencial nativa da Nuvem com Intel SGX

Nesta seção, discutimos como combinar computação confidencial com computação nativa da nuvem para nosso caso de uso. Para a implementação dos componentes, detalharemos a criação de um produtor baseado no SGX SDK e de um processador (consumidor e produtor) baseado no SCONE. Outros exemplos, como um consumidor SDK, além de produtores e consumidores SCONE, estão disponíveis no repositório do mini-curso [LSD, 2020].

1.5.1. Construindo clientes Kafka confidenciais utilizando SGX SDK

Conforme descrito na Seção 1.4, o desenvolvimento de aplicações que utilizam SGX SDK deve ser feito utilizando a linguagem C/C++, particionadas em uma parte confiável e uma parte não confiável. Além disso, uma vez que o enclave não consegue executar chamadas de sistema o desenvolvedor deve gerenciar a execução de atividades de persistência e comunicação que circularam pela porção não confiável, de modo que não sejam vazadas informações potencialmente sensíveis.

Também consequência da impossibilidade da execução de chamadas de sistema pela parte confiável, é a impossibilidade de ligação de várias bibliotecas populares de C/C++ contra enclaves, como ASIO, Boost, librdkafka, RapidJSON, entre outras. Para utilizar tais bibliotecas, uma alternativa é ligá-las contra a parte insegura e utilizá-las através de *ocalls*, porém, não há garantias em relação à segurança das informações que são transacionadas através de *ocalls*.

Neste exemplo utilizaremos a biblioteca *librdkafka*[26] para implementação de um cliente produtor Kafka. O gerenciamento de chaves criptográficas bem como os

[25] https://sconedocs.github.io/CAS_session_lang_0_3
[26] https://github.com/edenhill/librdkafka

processos de criptografia e descriptografia de mensagens são feitos dentro do enclave, de forma que nenhuma informação confidencial é vazada em texto plano. Uma alternativa à esta biblioteca é a implementação do suporte ao protocolo do Kafka dentro do enclave, que é uma tarefa complexa. Outra biblioteca utilizada é a JSMN, para fazer a análise de strings JSON. Esta, por sua vez, é ligada diretamente contra o enclave, uma vez que ela não depende de chamadas de sistema.

Para a garantir segurança dos dados, a gerência do processo de atestação é feita pelo enclave, assim como a descriptografia dos dados simulados e a nova criptografia com a chave do Kafka. A porção insegura recebe então apenas dados opacos que ela encapsula e envia para a rede. Por simplicidade, utilizamos somente uma chave de criptografia e um tópico, mas a solução poderia ser facilmente estendida para sistemas mais complexos (considerando também os pontos a serem discutidos na Seção 1.6).

1.5.1.1. Gerenciador de Segredos

O gerenciador de segredos é o componente responsável por armazenar e entregar chaves criptográficas para os demais componentes da aplicação. Escrito aqui em Python, assumimos que este componente executa em uma máquina de confiança do usuário.

A entrega de segredos é realizada mediante atestação remota, ou seja, as aplicações são submetidas ao processo de atestação remota para verificação de suas identidades, conforme discutido na Seção 1.4.2, para que então os segredos sejam entregues. No exemplo desenvolvido aqui, o gerenciador de segredos identifica as aplicações através dos seus respectivos valores de MRENCLAVE e para cada um estão associados segredos diferentes. Esses segredos são carregados através de um arquivo de configuração que utiliza o formato JSON, conforme exposto na Figura 1.37.

```
01 | {
02 |  "app-list": [
03 |   {
04 |    "app-name": "consumer",
05 |    "mrenclave": "164f40a42ab6908cde5183e9f6e662a408d0c31a684afedf81b2506b62b32979",
06 |    "secrets": {
07 |     "KAFKA_SECRET": "4145533132384b65792d313643686172"
08 |    }
09 |   },
10 |   {
11 |    "app-name": "producer",
12 |    "mrenclave": "0c0ccf1e24287cfa50d6a85f7652afbfd482761c7d89c11124540a528a690984",
13 |    "secrets": {
14 |     "KAFKA_SECRET": "4145533132384b65792d313643686172",
15 |     "PAYLOAD_SECRET": "b588fbd73e704331df95764ed85faf78"
16 |    }
17 |   }
18 |  ]
19 | }
```

Figura 1.37 - Exemplo de arquivo de configuração com segredos

Na Figura 1.37 podemos identificar duas entradas de aplicações em *app-list*: *consumer* e *producer*. Para cada entrada nós temos um nome (*app-name*), utilizado apenas para identificar a aplicação; um MRENCLAVE (*mrenclave*) que é utilizado para checagem da identidade da aplicação; e por fim, os segredos (*secrets*), que são as chaves criptográficas utilizadas para criptografar e descriptografar os conteúdos das mensagens dentro dos enclaves.

A comunicação entre o sistema gerenciador de segredos e as aplicações utilizam *socket* em sua comunicação, protegida através do uso de primitivas criptográficas como CMAC (do inglês, *Cipher-Based Message Authentication Code*) para autenticação, e criptografia utilizando chaves simétricas derivadas a partir de um processo de troca de chaves de Diffie–Hellman.

Durante a troca de mensagens, verificações são realizadas para identificar possíveis adulterações que possam ser realizadas por atacantes no processo de atestação remota. A Figura 1.38 apresenta o processo de verificação do *quote*, recebido pelo sistema gerenciador de segredos, ao longo do processo de atestação.

```
085 |     gid = quote[0x4:0x8]
086 |     if gid != acontext.get_sp_gid():
087 |         print("FAIL: GID on quote is different
                        of initial GID. Aborting...")
088 |         return False
089 |
090 |     derived_key_smk = acontext.get_derived_key("SMK")
091 |     calc_mac = aes128_cmac(derived_key_smk, g_a +
                                          ps_sec_prop_desc + quote)
092 |
093 |     if calc_mac != mac:
094 |         print("FAIL: MAC on MSG3 is different of
                      computed locally. Aborting...")
095 |         return False
096 |
097 |     quote_report_data = quote[QUOTE_REPORT_DATA_OFFSET:
098 |                               QUOTE_REPORT_DATA_OFFSET +
                                  QUOTE_REPORT_DATA_LENGTH]
099 |
100 |     derived_key_vk = acontext.get_derived_key("VK")
101 |     calc_report_data = sha256_hash(acontext.get_sp_public_numbers()['x']+
                                        acontext.get_sp_public_numbers()['y']+
102 |                                     acontext.get_public_numbers()['x']+
                                        acontext.get_public_numbers()['y']+
103 |                                     derived_key_vk, False)+
                                        binascii.unhexlify(32 * b'00')
104 |
105 |     if quote_report_data != calc_report_data:
106 |         print("FAIL: [#%d] Quote report data on MSG3 is different of the
                     computed locally. Aborting...", acontext.get_id())
107 |         return False
```

Figura 1.38 - Processo de verificação do *quote* recebido da aplicação

Na Figura 1.38, campos do *quote* são verificados com o objetivo de verificar sua integridade, uma vez que a computação utilizada para calcular o CMAC e *hash* desses campos utilizam chaves derivadas do processo de troca de chaves de Diffie-Hellman. Ao final do processo de atestação, os segredos são criptografados utilizando uma das chaves simétricas derivadas, e enviados para a aplicação cliente, conforme visto na função *send_secrets()* na Figura 1.39[27].

```
056 | def send_secrets(client_socket, address, acontext, mrenclave):
057 |     global secrets
058 |
059 |     derived_key_sk = acontext.get_derived_key("SK")
060 |
061 |     msg_type = struct.pack('<B', MSG_TYPE_SECRETS)
062 |     payload = aes_encrypt(json.dumps(secrets[mrenclave]), derived_key_sk)
063 |     payload_len = struct.pack('<I', len(payload))
064 |
```

[27] A implementação completa do sistema de gerenciamento de segredos está disponível em: https://git.lsd.ufcg.edu.br/lsd-sbseg-2020/kafka-sample-sgxsdk/tree/master/attestor

```
065 |     msg_secrets = msg_type + payload_len + payload
066 |
067 |     try:
068 |         client_socket.send(msg_secrets)
069 |         print("INFO: [#%d] Encrypted secrets sent!" % acontext.get_id())
070 |     except:
071 |         print("FAIL: [#%d] Unable to send secrets!" % acontext.get_id())
```

Figura 1.39 - Código de criptografia e envio de segredos

1.5.1.2. Produtor Kafka

Para implementar o produtor Kafka utilizando o SGX SDK, foi utilizada para a comunicação com o Apache Kafka a biblioteca *librdkafka*. Essa aplicação simula um gateway Kafka para medições de energia produzidos por um *smart meter*. Essas medições se encontram criptografadas no arquivo *smartmeter-data.enc*, e a chave para descriptografar as informações é recebida do sistema gerenciador de segredos durante o processo de atestação.

```
095 | static int initialize_kafka_producer() {
096 |     char errstr[512];
097 |
098 |     run = 1;
099 |     conf = rd_kafka_conf_new();
100 |
101 |     if (rd_kafka_conf_set(conf, "bootstrap.servers", broker,
102 |                           errstr, sizeof(errstr)) != RD_KAFKA_CONF_OK) {
103 |         fprintf(stderr, "%s\n", errstr);
104 |         rd_kafka_conf_destroy(conf);
105 |         return 0;
106 |     }
107 |
108 |     rd_kafka_conf_set_dr_msg_cb(conf, dr_msg_cb);
109 |
110 |     rk = rd_kafka_new(RD_KAFKA_PRODUCER, conf, errstr, sizeof(errstr));
111 |     if (!rk) {
112 |         fprintf(stderr,
113 |                 "%% Failed to create new producer: %s\n", errstr);
114 |         return 0;
115 |     }
116 |
117 |     signal(SIGINT, stop);
118 |
119 |     return 1;
120 | }
```

Figura 1.40 - Código de inicialização do produtor Kafka

A Figura 1.40 apresenta o código utilizado para inicializar um produtor Kafka utilizando a *librdkafka*. Esse produtor é inicializado na parte não confiável da aplicação, consequentemente informações relacionadas aos metadados das mensagens e detalhes sobre o cluster podem ser obtidos por alguém com acesso privilegiado ao sistema. A recuperação das chaves criptográficas pelo lado seguro é disparada a partir de uma chamada de *ecall* para a parte confiável, como ilustrado na Figura 1.41 e as chaves recuperadas nunca deixam o enclave.

```
283 |         fprintf(stderr, "INFO: Getting secrets from attestation service
                    '%s:%d'...\n", attestation_host, attestation_port);
284 |         if (SGX_SUCCESS != ecall_load_keys_from_server(global_eid,
                &ecall_ret, attestation_port, attestation_host,
                strlen(attestation_host) + 1) || ecall_ret < 0) {
285 |             fprintf(stderr, "FAIL: Unable to get keys from attestation
                                service! Exiting...\n");
286 |
287 |             sgx_destroy_enclave(global_eid);
288 |
```

```
289 |             return EXIT_FAILURE;
```

Figura 1.41 - Chamada de *ecall* para obtenção das chaves a partir do gerenciador de segredos

A função *ecall_load_keys_from_server()*, por sua vez, executa a função *do_remote_attestation()* dentro do enclave, que realiza o processo de atestação remota, e consequentemente obtenção das chaves criptográficas.

```
320 | int do_remote_attestation(int ra_port, char *ra_host,
                                uint8_t p_payload_key[SGX_AESGCM_KEY_SIZE],
                                uint8_t p_kafka_key[SGX_AESGCM_KEY_SIZE]) {
321 |     int sock_fd, ret, ra_ret = 0;
322 |     ssize_t recv_ret;
323 |     sgx_status_t ocall_ret;
324 |
325 |     sgx_ra_context_t ra_ctx;
326 |     sgx_ec256_public_t pubkey;
327 |
328 |     uint8_t recv_msg_type;
329 |
330 |     ocall_ret = ocall_create_connection(&sock_fd, ra_port, ra_host);
331 |     if (ocall_ret != SGX_SUCCESS) {
332 |         print_string("ENCL: FAIL: Unable to call
                           'ocall_create_connection'...\n");
333 |
334 |         return RA_ERROR;
335 |     }
336 |
337 |     if (sock_fd < 0) {
338 |         print_string("ENCL: FAIL: Cannot connect to host: [%s:%d]\n", ra_host,
                            ra_port);
339 |
340 |         return RA_ERROR;
341 |     }
342 |
343 |     if (send_init_ra(sock_fd) < 0) {
344 |         print_string("ENCL: FAIL: Unable to send INIT_RA message.
Exiting...\n");
345 |
346 |         ocall_close_connection(&ret, sock_fd);
347 |         return RA_ERROR;
348 |     }
349 |
350 |     do {
351 |         ocall_ret = ocall_recv(&recv_ret, sock_fd,
                                   (char *)&recv_msg_type,
                                   sizeof(uint8_t));
352 |
353 |         if (SGX_SUCCESS != ocall_ret || recv_ret != sizeof(uint8_t)) {
354 |             print_string("ENCL: FAIL: Unable to receive message.
Exiting...\n");
355 |             ra_ret = RA_ERROR;
356 |             break;
357 |         }
358 |
359 |         switch (recv_msg_type) {
360 |             case MSG_TYPE_MSG0:
361 |                 ra_ret = handle_ra_msg0(sock_fd, &ra_ctx, &pubkey);
362 |                 break;
363 |
364 |             case MSG_TYPE_MSG2:
365 |                 ra_ret = handle_ra_msg2(sock_fd, &ra_ctx);
366 |                 break;
367 |
368 |             case MSG_TYPE_SECRETS:
369 |                 ra_ret = handle_ra_secrets(sock_fd, &ra_ctx,
                                                p_payload_key, p_kafka_key);
370 |                 break;
371 |         }
372 |     } while (ra_ret == RA_CONTINUE);
373 |
```

```
374 |     ocall_close_connection(&ret, sock_fd);
375 |
376 |     return ra_ret;
377 | }
```

Figura 1.42 - Função que realiza o processo de atestação remota

No código apresentado na Figura 1.42 podemos observar que o código realiza o processo de atestação remota através do uso de funções que processam as mensagens recebidas de acordo com o seu tipo (*MSG0*, *MSG2* e *MSG_SECRETS*). As mensagens do tipo *MSG0* e *MSG2* representam as mensagens trocadas no processo de atestação remota, conforme apresentado na Seção 1.4.2, enquanto a *MSG_SECRETS* é o tipo de mensagem que transporta os segredos, e só é enviada pelo gerenciador de segredos em caso de sucesso no processo de atestação.

Após o processo de atestação remota, a aplicação recebe duas chaves: *PAYLOAD_SECRET* e *KAFKA_SECRET*. A chave *PAYLOAD_SECRET* é utilizada para descriptografar dentro do enclave a informação lida do arquivo de medições criptografadas, enquanto a chave *KAFKA_SECRET* é utilizada para criptografar as mensagens que vão ser enviadas para o tópico no barramento de dados.

Para cada medição criptografada, é executada a *ecall ecall_reencrypt_message()* pela parte não confiável, que recebe o conteúdo criptografado e, dentro do enclave, o descriptografa com a chave *PAYLOAD_SECRET* e criptografa utilizando a chave *KAFKA_SECRET*, fazendo a publicação no Kafka utilizando a *ocall ocall_produce_message()*[28].

1.5.1.3. Execução e testes com Kubernetes

Para executar a aplicações *producer* com Kubernetes precisamos construir as imagens, utilizando os *Dockerfiles* dentro dos diretórios correspondentes a cada aplicação. Para facilitar o processo de criação das imagens, disponibilizamos dois *scripts* chamados *build-images-hw.sh* e *build-images-sim.sh*[29]. Quando executado, o primeiro *script* gerará imagens prontas para executar as aplicações utilizando SGX, já o segundo gerará imagens que executam em modo simulado, dispensando o uso do *driver* SGX, mas sem nenhuma garantia de proteção de memória. Com as images *Docker* construídas, podemos criar os arquivos YAML que descrevem como será a implantação de cada aplicação no Kubernetes, para cada aplicação.

Primeiramente, é importante notar que, o attester tipicamente executaria fora da nuvem, em uma máquina de controle e confiança do usuário que implanta a aplicação. Além disso, em um modo de testes, ele poderia ser configurado para não consultar a Intel para validação (definindo a variável de ambiente *NO_IAS*). Caso contrário, para validar os quotes com o IAS, é preciso realizar a inscrição no serviço de atestação da Intel e obter um SPID e uma *subscription key*. Isso pode ser feito através da página do

[28] A implementação completa do produtor Kafka está disponível em https://git.lsd.ufcg.edu.br/lsd-sbseg-2020/kafka-sample-sgxsdk/tree/master/producer
[29] https://git.lsd.ufcg.edu.br/lsd-sbseg-2020/kafka-sample-sgxsdk

serviço de atestação[30], seguindo as instruções de cadastro. Além disso, há uma documentação da API disponível[31].

A Figura 1.43 mostra o arquivo YAML para a implantação do *producer*.

```
01 | apiVersion: apps/v1
02 | kind: Deployment
03 | metadata:
04 |   name: producer-sdk
05 | spec:
06 |   selector:
07 |     matchLabels:
08 |       run: producer-sdk
09 |   replicas: 1
10 |   template:
11 |     metadata:
12 |       labels:
13 |         run: producer-sdk
14 |     spec:
15 |       containers:
16 |       - name: producer-sdk
17 |         args: ["./producer", "10.11.19.115:9092", "sbsegtopic",
                    "./input/smartmeter-data.enc"]
18 |         image: lsd-sbseg-2020/producer:hw
19 |         imagePullPolicy: Always
20 |         env:
21 |           - name: "ATTESTATION_SERVICE"
22 |             value: "attestor:10719"
23 |         volumeMounts:
24 |         - mountPath: /dev/isgx
25 |           name: dev-isgx
26 |         securityContext:
27 |           privileged: true
28 |       volumes:
29 |       - name: dev-isgx
30 |         hostPath:
31 |           path: /dev/isgx
```

Figura 1.43 - Implantação do *producer* no Kubernetes: arquivo YAML de implantação

O exemplo mostra a implantação do *producer* com SGX habilitado. Na linha 17, temos os argumentos que são o endereço do Kafka, o nome do tópico onde serão inseridas as mensagens produzidas e o arquivo que contém o conteúdo que será transformado em mensagens Kafka. Além desses argumentos, a variável de ambiente *ATTESTATION_SERVICE* indica o endereço do *attester*, onde o *producer* buscará pelas chaves de criptografia necessárias para ler os dados criptografados, e para produzir as mensagens. As chaves são entregues após o processo de atestação bem sucedido.

De posse dos arquivos de implementação das aplicações e dos segredos do *attestor*, podemos realizar a implantação de fato, usando o comando *kubectl* do Kubernetes (para a versão MicroK8s, a mais simples de ser instalada, o comando seria *microk8s kubectl apply -f producer-deployment.yaml*). Depois da execução dos comandos de implantação, podemos verificar o estado das aplicações que acabamos de implantar com o comando *kubectl get pods*.

1.5.2. Construindo clientes Kafka confidenciais em Python com SCONE

Como discutido anteriormente, com aplicações confidenciais construídas com SCONE as porções inseguras são geridas pelo SCONE e o código do usuário é todo inserido na parte segura. Desta maneira, nessa seção mostraremos a implementação de uma

[30] https://api.portal.trustedservices.intel.com/EPID-attestation
[31] https://api.trustedservices.intel.com/documents/sgx-attestation-api-spec.pdf

aplicação simples que se comunica com um servidor Apache Kafka, consumindo e produzindo eventos, e que tira proveito da flexibilidade oferecida pelos contêineres seguros SCONE para se tornarem confidenciais.

1.5.2.1. Construindo um Processador de dados Kafka em Python

Apresentamos agora um código para processar os dados consumidos de um tópico e tomar uma decisão com base no conteúdo desses dados[32]. No exemplo, o processador será responsável por detectar se existem anomalias nas medições coletadas por sensores inteligentes de energia e retroalimentar o Kafka em um tópico chamado *alarms* com o tempo em que a anomalia ocorreu.

```
01 | import sys, getopt, os
02 | from json import loads
03 | from time import sleep
04 | from kafka import KafkaConsumer, KafkaProducer
05 | from kafka.errors import KafkaError
06 | from cryptography.hazmat.backends import default_backend
07 | from cryptography.hazmat.primitives.ciphers import (
08 |     Cipher, algorithms, modes
09 | )
10 |
11 | KEY = os.getenv('KAFKA_KEY').encode('ascii')
12 | NONCE_BYTE_SIZE = 12
```

Figura 1.44. Trecho de Código - Carregamento de módulos (Processador)

O processador possui características tanto de consumidor quanto de produtor, assim como mostrado na Figura 1.44. A função *decrypt* (Figura 1.45) recebe como argumentos a chave para decifragem, os dados de autenticação de mensagem, o *nonce*, o texto cifrado e a *tag*, todas essas variáveis são necessárias no algoritmo AES-GCM. Como resultado da função temos os dados originais de energia que foram enviados ao servidor (linha 22). A função *consume*, descrita na Figura 1.46, se conecta com o servidor Kafka informado no parâmetro *server*, a conexão com o servidor passa por uma etapa de autenticação SSL. Observe ainda que o consumidor é configurado sem *enable_auto_commit*, ou seja, receber mensagens através desse consumidor não reflete na marcação mantida no Zookeeper. Assim, toda vez que esse consumidor é iniciado ele começa a consumir mensagens a partir do mesmo ponto. Esse consumo se dá de acordo com o valor escolhido no parâmetro *auto_offset_reset* que definimos como *latest*, ou seja, o evento mais recente.

```
14 | def decrypt(key, associated_data, nonce, ciphertext, tag):
15 |     decryptor = Cipher(
16 |         algorithms.AES(key),
17 |         modes.GCM(nonce, tag),
18 |     ).decryptor()
19 |
20 |    if associated_data is not None:
20 |         decryptor.authenticate_additional_data(associated_data)
21 |
22 |     return decryptor.update(ciphertext) + decryptor.finalize()
```

Figura 1.45. Trecho de Código - Função para decifrar

```
24 | def consume(server, topic):
25 |     consumer = KafkaConsumer(topic,
26 |                              security_protocol='SSL',
```

[32] Código completo do processador disponível em: https://git.lsd.ufcg.edu.br/lsd-sbseg-2020/kafka-sample-scone/blob/master/src/feedback.py

```
27 |                              ssl_check_hostname=False,
28 |                              ssl_cafile='/certs/CARoot.pem',
29 |                              bootstrap_servers=[server],
30 |                              auto_offset_reset='latest',
31 |                              enable_auto_commit=False)
32 |     for message in consumer:
33 |         encoded_message = message.value
34 |
35 |          nonce = encoded_message[:12]
36 |          tag = encoded_message[12:28]
37 |          cipher = encoded_message[28:]
38 |
39 |          text = decrypt(KEY, None, nonce, cipher, tag).decode('ascii')
40 |          return text
```

Figura 1.46. Trecho de Código - Função de consumo com retorno

Para produção de avisos de alerta, criamos a função *produce_warning* (Figura 1.47), que é similar à função *produce* utilizada no produtor, mas que recebe como parâmetro, além do endereço do servidor Kafka, a variável *message*, que representa o conteúdo da mensagem a ser publicada no tópico *alarm*, linhas 42 e 45.

```
42 | def produce_warning(server, message):
43 |     producer = KafkaProducer(bootstrap_servers=[server],
                                 security_protocol='SSL',
                                 ssl_check_hostname=False,
                                 ssl_cafile='/certs/CARoot.pem')
44 |
45 |     future = producer.send("alarm", value=b'[alarm] - ' +
message.encode('ascii'))
46 |     try:
47 |         record_metadata = future.get(timeout=5)
48 |     except KafkaError as e:
49 |         print(e)
50 |         sys.exit(1)
```

Figura 1.47. Trecho de Código - Função de produção de alertas

A lógica de detecção de anomalias nos dados de energia fica então no código executado como módulo principal (Figura 1.48). Note que entre as linhas 69 e 75 temos um laço, que consome uma métrica e a codifica como JSON (linha 70), e faz uma avaliação da coluna *V*, que representa a tensão da rede. Definimos para este exemplo que uma anomalia ocorre quando o valor de *V* é inferior a 210. A linha 71 é responsável por verificar se uma anomalia ocorreu e, caso ocorra, a função *produce_warning* é chamada e o tempo em que a métrica foi coletada, coluna *timestamp*, é enviado como conteúdo da mensagem de alarme.

```
52 | if __name__ == "__main__":
53 |     argv = sys.argv[1:]
54 |
55 |     try:
56 |         opts, args = getopt.getopt(argv,"hs:t:",["server=","topic="])
57 |     except getopt.GetoptError:
58 |         print('feedback.py -s <server:port> -t topic')
59 |         sys.exit(2)
60 |     for opt, arg in opts:
61 |         if opt == '-h':
62 |             print('feedback.py -s <server:port> -t topic')
63 |             sys.exit()
64 |         elif opt in ("-s", "--server"):
65 |             server = arg
66 |         elif opt in ("-t", "--topic"):
67 |             topic = arg
68 |
69 |     while True:
70 |         metric = loads(consume(server, topic))
```

```
71 |         if float(metric['V']) <= 210.00:
72 |             produce_warning(server, str(metric['timestamp']))
73 |             print("alarm sent")
74 |
75 |         sleep(5)
```

Figura 1.48. Trecho de Código - Checagem de alertas

1.5.2.2. Adicionando confiabilidade às soluções utilizando SCONE

As aplicações Python apresentadas nas seções anteriores ainda não possuem as garantias de confiabilidade que ambientes de execução confiável proporcionam. Para que possamos executá-las dentro de enclaves protegidos Intel SGX, podemos utilizar contêineres que contenham uma versão protegida do interpretador Python (ou seja, o interpretador foi compilado com o próprio SCONE).

Como base de construção, utilizamos a imagem curada que contém um interpretador Python-3.7 seguro[33], que está disponível para acesso público. Esse interpretador utiliza da versão modificada da *musl-libc* para executar de forma transparente dentro do enclave SGX. Como detalhado no *Dockerfile* representado na Figura 1.49, adicionamos a essa imagem base as bibliotecas *libffi* e *openssl*, que são necessárias para funções de criptografia (linha 2). Copiamos então o código-fonte da aplicação para uma pasta no contêiner chamada */app* e também o arquivo *requirements.txt*, que descreve quais módulos Python a aplicação utiliza (linha 4). Na linha 6 os módulos são instalados através da ferramenta *pip*. A imagem desse contêiner pode então ser gerada com o comando: *docker build . -t sbseg:latest*

```
01 | FROM sconecuratedimages/kubernetes:python-3.7.3-alpine3.10-scone4.2
02 | RUN apk update && apk add gcc g++ libffi-dev openssl-dev
03 | COPY ./feedback.py /app
04 | COPY requirements.txt /
05 | WORKDIR /app
06 | RUN pip install -r /requirements.txt
07 | ENTRYPOINT python3
```

Figura 1.47. Trecho de Código - Arquivo *Dockerfile*

Além da imagem Python segura, precisamos garantir a integridade (ou até a confidencialidade do código). Como discutimos na Seção 1.4.6, Python é uma linguagem interpretada, sendo assim é o interpretador que está sendo verificado quando há uma execução. Ou seja, isso significa que o *hash* ou MRENCLAVE das aplicações é o do próprio interpretador, e ele então lê o código depois de carregado. Assim, as aplicações Python estão sujeitas a ataques contra sua integridade, mesmo executando dentro de enclaves protegidos. Para remediar este problema precisamos proteger o sistema de arquivos da imagem, que contém o código-fonte e as dependências. Como visto na Seção 1.4.6, isso pode ser feito de forma transparente. Usamos então o mesmo script da Figura 1.32, que cria, utilizando FSPF, uma região criptografada no diretório */app*, onde o código-fonte estará armazenado.

Utilizando do script *fspf.sh* e se beneficiando da solução de *multi-stage building* do Docker, cria-se então um novo contêiner, que consiste no encapsulamento do criado na seção anterior e seu Dockerfile é mostrado na Figura 1.48. Usando como base a imagem *sbseg:latest* executamos o *fspf.sh* (linha 7), assim criando o diretório protegido

[33] *Tag* no Docker Hub: sconecuratedimages/kubernetes:python-3.7.3-alpine3.10-scone4.2

/app na imagem de apelido *fspf*, dessa imagem copiamos o arquivo *fspf.pb* e o conteúdo de */app* para um diretório chamado *app* para a imagem final (linhas 9 e 10) e definimos o valor da variável *SCONE_CONFIG_ID*, que será utilizada pelo SCONE na fase de atestação remota.

```
01 | FROM sbseg:latest as fspf
02 | COPY fspf.sh /
03 | RUN mkdir /app
04 | WORKDIR /
05 | RUN SCONE_NO_FS_SHIELD=1 SCONE_MODE=sim /fspf.sh
06 | FROM sbseg:latest
07 | COPY --from=fspf /fspf.pb /
08 | COPY --from=fspf /app /app
09 | ENV SCONE_CONFIG_ID=python_kafka/python_kafka_producer
```

Figura 1.48. Trecho de Código - Arquivo *Dockerfile.fspf*

A imagem desse contêiner pode então ser gerada com o comando: *docker build . -t sbseg:fspf -f Dockerfile.fspf* e no arquivo *keytag* estarão os valores da chave e *tag* do arquivo *fspf.pb*, necessários para decifrá-lo.

O último passo antes de podermos executar o contêiner *sbseg:fspf* é submeter o arquivo de sessão da nossa aplicação para o CAS. A Figura 1.49 mostra o arquivo de sessão que é utilizado para registrar o serviço Kafka criado na seção anterior. Note que na linha 4 é definido o nome do serviço. O CAS é responsável por guardar os segredos das aplicações, no caso descrito os segredos são a chave de cifragem, definida na linha 10 como variável de ambiente e o certificado para autenticação do servidor Kafka contido nas linhas 17-39 como arquivo injetado. Note também que a chave e a *tag* do arquivo *fspf.pb* obtidas durante a criação do FSPF são informadas nas linhas 11-13.

```
 01 | name: python_kafka
 02 | version: "0.3"
 03 | services:
 04 |   - name: python_kafka_feedback
 05 |     image_name: python_kafka
 06 |     command: python3 -u feedback.py -s 127.0.0.1:9092 -t sbsegtopic
 07 |     mrenclaves:
[67b8017f7083435cb614b87c8daa14303f741a10a2a0bbf5dfabec777cf629b9]
 08 |     pwd: /src
 09 |     environment:
 10 |       KAFKA_KEY: "AES128Key-16Char"
 11 |     fspf_path: /fspf.pb
 12 |     fspf_key: {{key}}
 13 |     fspf_tag: {{tag}}
 14 | images:
 15 |   - name: python_kafka
 16 |     injection_files:
 17 |        - path: /certs/CARoot.pem
 18 |          content: |
 19 |             -----BEGIN CERTIFICATE-----
 20 |             MIIDazCCAlOgAwIBAgIUIj/cO1QmJhmzVDiGI07lMEErwXEwDQYJKoZIhvcNAQEL
 .. |             ...
 38 |             3jlCpqlCl/KzUVYzh5wH
 39 |             -----END CERTIFICATE-----
 40 | security:
 41 |   attestation:
 42 |     tolerate: [debug-mode, hyperthreading, outdated-tcb]
 43 |     ignore_advisories: "*"
```

Figura 1.49. Trecho de Código - Arquivo *session.yml*

1.5.2.5. Execução e testes com Kubernetes

Podemos testar o consumo dos dados publicados usando também o consumidor simples provido pelo próprio Kafka, dessa forma é possível ver que os dados que foram publicados estão cifrados. Para esse teste simples executaremos um *Pod* consumidor em um *cluster* Kubernetes. O arquivo mostrado na Figura 1.50 mostra um exemplo de manifesto para criação de um *Pod* consumidor. A imagem utilizada é definida na linha 10, e nas linhas 11 e 12 definimos o comando de execução do consumidor e seus parâmetros: endereço do servidor Kafka e tópico. As linhas 13-23 contém as variáveis de ambiente SCONE.

```
01 | apiVersion: v1
02 | kind: Pod
03 | metadata:
04 |   name: consume-demo
05 |   labels:
06 |     purpose: demonstrate-consumer
07 | spec:
08 |   containers:
09 |   - name: console-consumer
10 |     image: lucasmc/sbseg:fspf
11 |     command: ["python3"]
  12  |          args:  ["/src/kafkaconsumer.py",  "-s",  "10.30.0.51:9092",  "-t",
"sbsegtopic"]
13 |     env:
14 |       - name: "SCONE_CAS_ADDR"
15 |         value: "4-2-1.scone-cas.cf"
16 |       - name: "SCONE_LAS_ADDR"
17 |         value: "10.30.0.51"
18 |       - name: "SCONE_VERSION"
19 |         value: "1"
20 |       - name: "SCONE_LOG"
21 |         value: "7"
22 |       - name: "SCONE_CONFIG_ID"
23 |         value: "kafka_python/python_kafka_consumer"
24 |     volumeMounts:
25 |       - mountPath: /dev/isgx
26 |         name: dev-isgx
27 |     securityContext:
28 |       privileged: true
29 |   volumes:
30 |   - name: dev-isgx
31 |     hostPath:
32 |       path: /dev/isgx
33 |   restartPolicy: OnFailure
```

Figura 1.50. Trecho de Código - Arquivo *consumer.yaml*

Este manifesto pode então ser executado no Kubernetes (no caso do MicroK8s: *microk8s kubectl apply -f consumer.yaml)* e para observar os valores de saída basta olhar o registro de saída do *Pod* em questão (no caso do MicroK8s: *microk8s kubectl logs -f console-consumer*). O registro de saída do *Pod* mostrará as mensagens criptografadas que foram enviadas ao Kafka.

1.6. Boas práticas e direções promissoras

Esta seção complementa o material apresentado com sugestões de material complementar e direções de pesquisa e de refinamento dos exemplos expostos aqui.

1.6.1. Implantação de *software* nativo da nuvem

O extenso ecossistema de computação nativa da nuvem oferece uma série de soluções e ferramentas que podem ser facilmente integradas a novas aplicações nativas da nuvem,

promovendo a portabilidade e o desacoplamento. Helm[34], por exemplo, é um instalador de aplicações nativas da nuvem. Atuando de forma similar à de um gerenciador de pacotes de sistema operacional, *Helm* permite implantar aplicações em *clusters* Kubernetes de forma rápida. As aplicações são definidas em artefatos chamados *charts*, que são agrupados em repositórios. Para um operador de nuvem, basta adicionar determinado repositório e utilizar a interface de linha de comando do *Helm* para instalar, atualizar (com versionamento e possibilidade de recuperação em caso de falha) e gerenciar aplicações nativas da nuvem. É possível personalizar determinados parâmetros através de um arquivo YAML, o que facilita o gerenciamento de diferentes cenários de implantação (por exemplo, cenários como produção, desenvolvimento e teste).

Helm provê um repositório canônico de *charts* de aplicações populares. SCONE também oferece um repositório com versões seguras de algumas aplicações populares, dentre elas MariaDB, Kafka e PySpark, que podem ser instaladas em *clusters* Kubernetes que suportam computação confidencial (como Azure Kubernetes Services [Microsoft, 2020]). A utilização de serviços Kubernetes é uma alternativa mais prática em relação à configuração de máquinas virtuais, informações sobre a configuração de clusters podem ser encontrados nos apêndices disponíveis em [LSD, 2020].

1.6.2. Abordagens complementares de segurança

No caso de uso desenvolvido ao longo do capítulo, a utilização de mecanismos de computação confidencial permite que o desenvolvedor não precise confiar na integridade da infraestrutura (ex., *firmware*, sistema operacional, hipervisor). Isso é possível uma vez que apenas o seu código conseguirá acesso às chaves de criptografia dos dados e que uma vez obtidas, essas chaves não podem ser inspecionados pelo software privilegiado na infraestrutura. No entanto, a possibilidade de que algum componente de segurança apresente vulnerabilidades não deve ser descartada e é conhecido como segurança em camadas: caso uma das camadas se mostre mais frágil que o esperado, as outras ainda oferecerão proteção.

No caso em mãos, existem algumas alternativas que podem ser consideradas: (1) assumir um modelo de zero confiança (do inglês, *zero trust*); (2) atestação do carregamento do sistema operacional onde os enclaves estão executando; (3) configuração de mecanismos adicionais de controle de acesso.

A adoção de um modelo de zero confiança tem dois aspectos principais. Em primeiro lugar, pode-se assumir zero confiança na rede. Esta abordagem contrasta com o modelo de perímetro de segurança, onde as máquinas na rede local são confiáveis e as máquinas não confiáveis estão separadas das primeiras por um *firewall*. A confiança zero na rede então assume que a comunicação entre quaisquer dois componentes deve ser autenticada e criptografada. Para o nosso caso de uso, isso poderia ser implementado usando certificados X.509 para o estabelecimento de canais TLS mutuamente autenticados, através dos quais o cliente verifica o servidor, como fizemos nas seções anteriores, mas também o servidor autentica o cliente. Este modelo é suportado pela maioria de sistemas maduros, a exemplo do Apache Kafka e MariaDB, está disponível

[34] https://helm.sh/

em bibliotecas das linguagens de programação populares e é suportado por ferramentas de gerência de aplicações nativas da nuvem através de padrões como SPIFFE[35].

Além de não confiar na rede, podemos assumir também que humanos não devem ser confiados com segredos. Esta estratégia foi exemplificada na Seção 1.4, onde um segredo, que poderia ser uma chave criptográfica, foi escolhida internamente pelo sistema (através do SCONE CAS) e entregue diretamente ao código da aplicação depois da atestação. Sem nunca ter saído dos enclaves, o segredo nunca poderia ter sido vazado por um humano. Tal estratégia poderia ser usada também para geração de certificados de autenticação de servidores e clientes X.509 para conexões TLS, isso permite que código seja aprovado e assinado por um comitê (de modo que nenhum indivíduo possa gerar código que exporte os segredos) e depois de implantado nunca exponha os dados sensíveis do sistema em execução.

A atestação do sistema operacional pode ser feita a partir do uso de componentes como o TPM (do inglês, *Trusted Platform Module*) [Arthur e Challener, 2015]. Enquanto o carregamento de um enclave registra o carregamento de uma aplicação, o TPM pode ser usado para registrar o carregamento do sistema operacional, assim como serviços e seus arquivos de configuração. Embora o TPM considere uma base de confiança muito maior (todo o ambiente da máquina), e por isso seja menos confiável, ele serve como camada adicional de segurança, por exemplo, provendo evidências que uma versão atualizada do sistema operacional foi usado e que serviços desnecessários não foram carregados.

Finalmente, o Apache Kafka poderia ser configurado para limitar quais tópicos cada cliente pode acessar. Isto é feito usando listas de controle de acesso (do inglês, Access Control List - ACL[36]). O controle de acesso pode então ser baseado nas identidades embutidas em certificados do cliente, como vistos acima. Assim, como estes certificados podem ser específico do cliente, mesmo que um deles seja comprometido, o domínio de impacto da falha é limitado.

1.5.4. Alternativas para computação confidencial, privacidade e leis de proteção de dados

Computação confidencial pode ser alcançada através de recursos de hardware, como o Intel SGX discutido aqui, ou de técnicas algorítmicas, como computação homomórfica [Gentry, 2009] (HE, do inglês, *Homomorphic Encryption*) e computação multi-parte [Cramer, Damgard, e Nielsen, 2015] (MPC, do inglês, *Multi-Party Computation*).

HE e MPC são técnicas que realizam processamento sobre dados numéricos criptografados. O dado só é entregue a entidades não-confiáveis após ser criptografado. Embora não possam visualizar os dados originais, essas entidades conseguem realizar operações aritméticas sobre os dados criptografados. Com HE, a cifragem transforma cada valor numérico em um polinômio e o valor correto do dado pode ser decifrado por quem tiver a chave de entrada correta para a função polinomial. A desvantagem é que simples operações aritméticas se tornam custosas operações polinomiais [Fan e Vercauteren, 2012]. A técnica de MPC é semelhante à HE, pois também opera sobre

[35] https://spiffe.io/
[36] https://kafka.apache.org/documentation/#security

dados criptografados, porém, é ainda mais complexa pois as operações são realizadas de forma distribuída e síncrona, o que adiciona latência de rede ao processo [Orlandi, 2011]. Uma consequência positiva que se aplica à HE e MPC é que o fato de adições e multiplicações serem suficientes para replicar as operações lógicas AND, OR, XOR e NOT, permite que HE e MPC possam ser generalizadas para realizar computações arbitrárias, embora esta generalização seja um processo complexo [Gentry, 2010].

A HE pode ser categorizada em três tipos: computação parcialmente homomórfica (PHE, do Inglês *Partially Homomorphic Encryption*), computação um tanto homomórfica (SWHE, do Inglês *Somewhat Homomorphic Encryption*), e computação completamente homomórfica (FHE, do Inglês *Fully Homomorphic Encryption*). O PHE permite a realização apenas de um conjunto limitado de operações sobre dados criptografados, como por exemplo, apenas adições ou apenas multiplicações, como é o caso do algoritmo de criptografia RSA que é homomorficamente multiplicativo mas não é homomorficamente aditivo. No SWHE, adições e um número limitado de multiplicações são permitidas sobre um mesmo conjunto de dados criptografados. No FHE, adições e multiplicações podem ser realizadas de forma indiscriminada, mas em compensação a quantidade de operações adicionais requeridas para reduzir o impacto do ruído gerado durante a adição e multiplicação dos polinômios tornam a computação (quase) impraticável [Naehrig, Lauter, e Vaikuntanathan, 2011]. Para se ter uma ideia desse custo, Hayward e Chiang (2015) sorteou 20 números inteiros de 8 bits e computou a soma, produto vetorial, e variância, sobre esses números, que custaram, respectivamente, 34.9 segundos, 952.17 segundos, e 2496.62 segundos [Hayward e Chiang, 2015], operações que durariam apenas algumas dezenas de ciclos em um enclave.

Uma vantagem clara de enclaves sobre HE e MPC é o custo computacional. Quando se trata da técnica de HE, existe um custo-benefício entre eficiência do processamento confidencial (PHE e SWHE) e a capacidade de executar uma quantidade arbitrária de operações de adição e multiplicação (FHE). PHE e SWHE são mais eficientes em tempo, porém mais limitadas em termos de capacidades de processamento confidencial, pois cada valor criptografado acumula ruído à medida que é adicionado ou multiplicado, até um dado ponto em que finalmente o ruído torna o resultado criptografado indecifrável [Naehrig, Lauter, e Vaikuntanathan, 2011]. O FHE, por outro lado, possui alta capacidade de processamento confidencial, sendo generalizável para computar muito além de operações aritméticas, mas são pouco viáveis em termos de tempo computacional para a grande maioria dos problemas [Hayward e Chiang, 2015]. Outra vantagem dos enclaves SGX é a capacidade de atestação de código, já que mesmo com a confidencialidade garantida, um código baseado em HE e MPC poderia ser adulterado para realizar computações diferentes da esperada.

Enquanto computação confidencial tem como diferencial a proteção de dados em uso, estando associada a mecanismos de comunicação e armazenamento seguros, privacidade tem como objetivo garantir a capacidade de agir seletivamente sobre quais de suas informações são compartilhadas. Computação confidencial pode então ser considerada uma ferramenta auxiliar na tarefa de garantir privacidade, já que sem mecanismos de confidencialidade, não haveria privacidade.

Por outro lado, existem estratégias alternativas que operam sobre os dados com o objetivo de reduzir informação, preservando a privacidade nos dados, mas podendo deteriorar a qualidade do dado original - anonimização [Zouinina et al., 2020] e privacidade diferencial [Nguyen, Kim e Kim, 2013]. Em termos de privacidade, anonimização pode simplesmente diminuir a informação disponível, eliminar os nomes de pessoas ou substituindo valores numéricos exatos por intervalos. Além de poder reduzir a utilidade dos dados, o uso de anonimização é frequentemente criticado pois pode deixar informações suficientes para que a anonimização seja desfeita, como no notório caso da Netflix [Narayanan e Shmatikov, 2008], onde o processo de anonimização foi desfeito para alguns usuários contidos no banco de dados.

Além dos aspectos de privacidade, outro conceito relacionado é a rastreabilidade. Computação confidencial por si só também pode não ser suficiente para garantir que indivíduos tenham controle sobre seus dados: quem os está usando, de que forma, e que como pode revogar o uso. Estes são alguns poderes exigidos nas leis mais recentes de proteção de dados, como a Lei Geral de Proteção de Dados (LGPD, lei nº 13.709, com vigência a partir de setembro de 2020).

Nesse sentido, empresas que lidam com dados de terceiros precisam se adequar aos requisitos da LGPD. As principais mudanças envolvem uma remodelagem da governança e gestão de processos, buscando promover transparência e facilidade de acesso às informações de utilização dos dados pessoais. Embora esse movimento por parte das empresas tenha se tornado evidente, existe uma distinção entre política e mecanismo. Mesmo que a lei estipule políticas, é possível que uma determinada empresa que em algum momento fez uso inadequado de dados pessoais altere informações emitidas em relatórios de uso de dados de clientes.

O processamento confidencial posto em prática por enclaves provêem duas interessantes propriedades que podem elevar o grau de auditabilidade e transparência de empresas que desejam estar aderentes à LGPD: (1) proteção de dados em memória e (2) atestação de um ambiente de execução. Enquanto a proteção de dados em uso na memória, discutida anteriormente, protege os dados de cópia por operadores e invasores, os procedimentos de atestação garantem que bibliotecas e binários em um ambiente de execução não foram adulterados. Protegidas contra alterações, as aplicações se comportarão da forma esperada, o que pode ser usado para implementar mecanismos de criação de relatórios na utilização de dados. Assim, os sistemas de controle e os relatórios de acesso serão confiáveis. Tal abordagem pode ser complementada pela adição de informações desses relatórios em um registro público imutável, como por exemplo, uma *blockchain*.

Uma implementação das capacidades mencionadas acima pode ser encontrada na plataforma DNAT [Nascimento et al., 2020] (*Data and Application Tracking*). Esta plataforma emprega computação confidencial via TEEs para impedir o acesso de agentes indevidos a dados sensíveis, bem como prover o rastreamento do uso dos dados pertencentes a uma entidade, e conceder a esta entidade o poder de revogação do uso de seus dados. No DNAT, registros públicos imutáveis são empregados para guardar o histórico de uso de um determinado dado assim como a aquisição de direito de uso e possíveis revogações. O uso de um dado é intermediado pela própria plataforma, a qual

emprega os registros públicos imutáveis e o Intel SGX para garantir que só agentes devidamente autorizados façam uso dos dados.

1.6. Considerações finais

Este trabalho apresentou a construção de uma aplicação prova-de-conceito para a disseminação e processamento de dados de sensores de forma confidencial. O uso de um barramento de mensagens Kafka e de um ambiente de execução SCONE facilitam a adaptação da aplicação para outros fins.

Alguns trechos de código foram omitidos neste trabalho. Os códigos completos podem ser obtidos no repositório Gitlab do minicurso [LSD, 2020]. Além do código, o repositório apresenta exemplos complementares e instruções para a criação de máquinas virtuais com suporte a SGX no provedor Microsoft Azure. Além da Azure, usuários do sistema RNP poderão contratar recursos de nuvem com suporte a SGX a partir do portal NasNuvens[37]. De forma semelhante, tais usuários poderão ter acesso ao sistema LiteCampus de processamento de dados de sensores com suporte a processamento confidencial, sistema que serviu de referência para o caso de uso discutido aqui.

1.7. Agradecimentos

Este trabalho foi apoiado pelo Laboratório de Sistemas Distribuídos (LSD) da Universidade Federal de Campina Grande, pelo GT LiteCampus da Rede Nacional de Pesquisa (RNP) e pela Smartiks Ltda. Agradecemos a José Benardi Nunes e Eduardo Falcão pelas sugestões e ao time da Scontain pelo apoio na utilização do SCONE.

Referências

Armel, K. C., Gupta, A., Shrimali, G., Albert, A. (2013) “Is disaggregation the holy grail of energy efficiency? The case of electricity”, Energy Policy.

Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., Fetzer, C. (2016) “SCONE: Secure Linux Containers with Intel SGX”. Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation.

Arthur, W., Challener, D. A practical guide to TPM 2.0: Using the new trusted platform module in the new age of security. Springer Nature, 2015.

Barbosa, P., Brito, A., Almeida, H. (2016) “A Technique to provide differential privacy for appliance usage in smart metering”, Information Sciences.

Cramer, R., Damgard, I. B., Nielsen, J. B., Secure multiparty computation. Cambridge University Press (2015).

Costan, V., Devadas, S. (2016) “Intel SGX Explained”. IACR Cryptol. ePrint Arch., v. 2016, n. 86.

Eugster, P. T., Felber, P., Guerraoui, R., Kermarrec, A.-M. (2003) “The many faces of publish/subscribe”. ACM computing surveys (Volume 35).

[37] https://www.nasnuvens.rnp.br/cms/

Fan. J., Vercauteren, F., “Somewhat practical fully homomorphic encryption”, IACR Cryptology ePrint Archive, vol. 2012, p. 144, 2012.

Gentry, C. A fully homomorphic encryption scheme. Tese de doutorado, Stanford University Stanford (2009).

Gentry, C. (2010) Computing arbitrary functions of encrypted data. Communication of the ACM 53, 3 (2010).

Hayward, R., Chiang, C. (2015) Parallelizing fully homomorphic encryption for a cloud environment, Journal of Applied Research and Technology, Volume 13, Issue 2.

IBM Security. Cost of a Data Breach Report 2020. (Online, acessado em 15/09/2020) https://www.ibm.com/security/data-breach

LSD. Processamento confidencial de dados de sensores na nuvem (Repositório). Laboratório de Sistemas Distribuídos, Universidade Federal de Campina Grande. (Online, acessado em 25/09/2020). https://git.lsd.ufcg.edu.br/lsd-sbseg-2020

Microsoft. Azure Confidential Computing Documentation. (Online, acessado em 24/09/2020). https://docs.microsoft.com/en-us/azure/confidential-computing/

Naehrig, M., Lauter, K., Vaikuntanathan, V. (2011) “Can homomorphic encryption be practical?”, Proceedings of the 3rd ACM Workshop on Cloud Computing Security.

Nguyen, H., Kim, J., Kim, Y. (2013) “Differential privacy in practice”, Journal of Computing Science and Engineering.

Narayanan, A., Shmatikov, V. (2008) “Robust De-anonymization of Large Sparse Datasets”, Proceedings of the IEEE Symposium on Security and Privacy.

Nascimento, J. R., Nunes, J. B. S., Falcão, E. L., Sampaio, L., Brito, A. (2020) “On the tracking of sensitive data and confidential executions”, Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems.

Oleksenko, O., Trach, B., Krahn, R., Martin, A., Silberstein, M., Fetzer, C. (2018) “Varys: Protecting SGX enclaves from practical side-channel attacks”, Proceedings of the 2018 USENIX Annual Technical Conference.

Orlandi, C., “Is multiparty computation any good in practice?”, Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, 2011.

Silva, F., Silva, M., Brito, A. (2020) “KafkaProxy: data-at-rest encryption and confidentiality support for Kafka cluster”, Anais do 20o Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais.

Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., Wilkes, J. (2015), “Large-scale cluster management at Google with Borg”, Proceedings of the ACM European Conference on Computer Systems.

Zouinina S., Bennani Y., Rogovschi N., Lyhyaoui A. (2020) “A Two-Levels Data Anonymization Approach”, Artificial Intelligence Applications and Innovations. Springer.

