Monitoramento Seguro de Métricas de Nível de Serviço em Nuvem

João Pedro Curvelo[1], **Carlos Alberto Maziero**[1]

[1]Programa de Pós-Graduação em Informática – Universidade Federal do Paraná (UFPR)
Curitiba – PR – Brazil

joao.pedro1905@gmail.com, maziero@inf.ufpr.br

Abstract. *Cloud computing is widely adopted by companies due to the scalability, high availability, and flexibility it offers, but cost management and data security are notable concerns. Service Level Agreements (SLAs) combine the topics of cost management and data confidentiality. However, monitoring the level of service offered is a task performed by the cloud provider itself, and the customer has no guarantees regarding the integrity of the collected data. This work proposes a cloud monitoring system for quality of service metrics that ensures the collected data has not been tampered with during monitoring. For this purpose, a threat model is proposed, as well as a monitoring architecture that relies on a hardware-based trusted execution environment. A proof-of-concept prototype of such architecture was built, to assess its viability for ensuring confidential SLA monitoring in the cloud. The results of the current study show that, despite the overhead imposed by the trusted environment on system performance, it is still viable in what was proposed.*

Resumo. *A computação em nuvem é amplamente adotada pelas empresas devido à escalabilidade, disponibilidade e flexibilidade que oferece, porém, a gestão de custos e a segurança dos dados são preocupações relevantes nesse contexto. Os Acordos de Nível de Serviço (SLA - Service Level Agreement) unem o tema de gestão de custos e confidencialidade de dados. O monitoramento do nível do serviço oferecido, contudo, é realizado pelo próprio provedor de nuvem, e ao cliente não há garantia da integridade dos dados coletados. Este trabalho propõe um sistema de monitoramento em nuvem de métricas de qualidade de serviço que garante que os dados coletados não foram manipulados durante o monitoramento. Para tal, é sugerido um modelo de ameaças, uma arquitetura de monitoramento que usa tecnologias de hardware para execução segura. Foi construído um protótipo da arquitetura proposta, para estimar sua viabilidade para garantir o monitoramento íntegro e confidencial de métricas de SLA na nuvem. Os resultados deste estudo mostram que, a despeito da sobrecarga imposta pelo ambiente de execução segura ao desempenho do sistema, ele ainda é viável no que foi proposto.*

1. Introdução

Acordos de nível de serviço (SLA – *Service Level Agreement*) são a convergência entre gestão de custos com infraestrutura remota e preocupação com confidencialidade de dados. São métricas de qualidade de serviço acordadas entre provedor e cliente, e quando não

mantidas, incorrem em desconto no valor a ser pago pelo contratante. As condições especificadas em um SLA firmado têm impacto direto no custo final dos serviços ofertados em nuvem. O provedor que garante um nível de serviço é também o responsável por coletar, armazenar e transmitir as métricas contratadas, portanto hospedar um sistema em nuvem para resolver tal problema não garantiria confidencialidade dos dados recolhidos. O acompanhamento seguro de tais métricas e a atestação da integridade dos dados coletados ainda são desafios comuns a sistemas de monitoramento de SLA [Badshah et al. 2023; De Carvalho et al. 2017; Alhamad et al. 2011].

O presente estudo propõe a implementação de um sistema de monitoramento em nuvem de métricas de utilização de processamento e memória que garanta confidencialidade e integridade dos dados recolhidos. Em detalhes, se espera mapear qual a superfície de ataque da aplicação e qual sua base computacional segura. Mapear, ainda, as permissões de uma entidade maliciosa, quais os possíveis pontos de falha ou pontos de risco do monitor e sugerir contramedidas responsáveis por mitigar possíveis ameaças à segurança.

Para tal, é proposta uma arquitetura que faz uso de ambientes de execução segura baseados em *hardware* (TEEs - *Trusted Execution Environments*) como Intel SGX (*Software Guard Extensions*) [Costan and Devadas 2016] e de contêineres seguros como SCONE [Brito et al. 2020] para garantir a confidencialidade e integridade no armazenamento, manipulação e comunicação dos dados coletados por um agente de monitoramento em nuvem. O ambiente de execução segura Intel SGX expande o conjunto de instruções da arquitetura de processadores x86-64 para criar um ambiente de memória seguro denominado *enclave*. O SGX propõe manter a confidencialidade e integridade de dados e aplicações durante a execução do código contido no enclave [Jain et al. 2016]. O ambiente SCONE, por sua vez, usa virtualização em nível de sistema operacional e SGX para construir contêineres seguros e pode ser integrado ao Docker [Arnautov et al. 2016].

Finalmente, este trabalho busca validar a hipótese de que um sistema de monitoramento que use SGX é seguro frente às ameaças mapeadas. Ao elaborar uma análise de complexidade específica à implementação da arquitetura proposta, e também ao confrontar o tempo de execução da aplicação com e sem os contêineres seguros, o estudo busca dar alguma dimensão do potencial de escalabilidade do ambiente desenvolvido e o impacto do uso do SCONE em alcançar o objetivo proposto.

Este texto está estruturado como segue: a Seção 2 apresenta os principais trabalhos correlatos identificados na literatura; a Seção 3 define os principais componentes da arquitetura proposta e suas interações; a Seção 3.1 define o modelo de ameaças considerado no detalhamento da arquitetura; a Seção 4 especifica as interfaces e interações entre os componentes; a Seção 5 detalha os aspectos de implementação do protótipo construído e as dificuldades encontradas; a Seção 6 avalia o desempenho e escalabilidade da arquitetura e o impacto dos contêineres seguros; por fim, a Seção 7 conclui o texto.

2. Trabalhos Relacionados

Esta seção analisa os principais trabalhos encontrados na literatura relacionados à monitoração segura de serviços em nuvem. Os temas abordados em cada trabalho estão sintetizados na Tabela 1; as próximas subseções discutem mais detalhadamente cada um desses trabalhos.

Estudo	Monitoramento	Intel SGX	SLA ou QoS	Computação em nuvem
[Nguyen and Ganapathy 2017]	✓	✓	✓	✓
[Zhan et al. 2021]	✓			✓
[Loch et al. 2021]	✓		✓	✓
[Jamous et al. 2020]	✓	✓	✓	✓
[Park et al. 2024]	✓	✓	✓	✓
[Dong et al. 2023]	✓	✓	✓	✓

Tabela 1. Temas dos trabalhos relacionados.

2.1. Sistemas de Monitoramento

Buscando atenuar os riscos de adulteração de dados, [Zhan et al. 2021] sugerem uma abordagem de monitoramento de contêineres a partir de um contêiner externo à aplicação. Essa entidade externa é transparente a todas as aplicações que monitora e é capaz de obter informações do ambiente de execução de seus alvos. O estudo mostra um sistema efetivo para mitigar ataques vindos dos recursos que hospedam as aplicações, mas apresenta sobrecarga pequena de desempenho.

Em [Nguyen and Ganapathy 2017], propõe-se um sistema chamado *EnGarde* que permite a análise estática do código de um enclave no momento de provisionamento do recurso, sem comprometer a confidencialidade oferecida pelo Intel SGX. O objetivo do trabalho é que, por meio desse sistema, duas partes (cliente e provedor, por exemplo) possam garantir que o código de um enclave respeite diretrizes impostas em um contrato de nível de serviço sem a quebra de confidencialidade. Como resultado, os autores demonstram com sucesso se o código de um enclave obedece às políticas impostas pelo SLA sem perda de desempenho.

Visando a confidencialidade de sistemas distribuídos em nuvem usando Intel SGX, [Park et al. 2024] propõem o *Cloister*, um modelo de TEE direcionado especificamente para a execução de aplicações *serverless* confiáveis. Além da introdução de técnicas voltadas para acelerar a implantação de aplicações seguras em nuvem, o trabalho também sugere soluções para a coleta confidencial de métricas do sistema e interações seguras entre enclave e sistema operacional.

Em [Dong et al. 2023], é apresentado o *T-Counter*, um *framework* baseado em Intel SGX que permite que aplicações mensurem seu próprio consumo de recursos de processamento de maneira sigilosa e íntegra. A proposta introduz três componentes seguros responsáveis por contar as instruções utilizadas pelo processador e defende-se contra provedores maliciosos de serviços em nuvem. O texto contempla ainda dois algoritmos que coordenam a execução dos componentes de contagem e avalia que o sistema apresentado é efetivo em mensurar a utilização de recursos de processamento confidencialmente.

2.2. Acompanhamento de SLA com Blockchain

[Loch et al. 2021] discutem um protocolo de *blockchain* para selecionar provedores de nuvem confiáveis e auditar contratos de SLA. Na etapa de auditoria, uma das atribuições do protocolo é garantir que métricas de nível de serviço disponibilizadas pelos provedores são confiáveis e, para tal, utiliza um módulo distribuído de monitoramento de SLA que recolhe

e analisa periodicamente métricas de QoS contratadas. Essas métricas são armazenadas localmente e enviadas a outros nós da rede em um processo de votação, visando chegar a um consenso sobre a quebra ou manutenção do contrato de SLA.

[Jamous et al. 2020] apresentam uma arquitetura de aplicação que usa contratos inteligentes e Intel SGX visando automatização e confidencialidade no processo de acordo, cumprimento e indenização de SLAs. Nesse contexto, os autores propõem que uma aplicação executada dentro de um enclave monitore e envie métricas de QoS a um algoritmo de decisão armazenado na *blockchain*, que, por sua vez, dispara automações com base nas métricas de QoS recebidas. A implementação do SGX é abstraída pelo *Town Crier*, uma ferramenta de execução confidencial adotada pelos autores, e a *blockchain* utilizada para armazenar os contratos inteligentes é a *Ethereum*.

2.3. Considerações

Um ponto de convergência dos trabalhos relacionados é a preocupação com confidencialidade e integridade de dados de monitoramento. Contudo, apenas [Park et al. 2024] e [Dong et al. 2023] utilizam majoritariamente o Intel SGX em nuvem para garantir o monitoramento seguro de métricas de SLA. [Nguyen and Ganapathy 2017] não monitoram dados de QoS; no lugar disso, aliam o uso de SGX em nuvem e conceitos de coleta de informações de SLA para desenvolverem uma solução de análise estática. Buscam, assim, garantir que uma política acordada em contrato de nível de serviço esteja sendo respeitada pelo cliente.

Os demais autores, por outro lado, desenvolvem soluções para assegurar que dados de monitoramento coletados pelo cliente não foram manipulados indevidamente. [Zhan et al. 2021], por exemplo, não abordam SGX em sua solução, mas debatem o tema de segurança no contexto de coleta de informações e apresentam uma arquitetura para monitoramento confidencial de recursos em nuvem. [Loch et al. 2021] também discorrem sobre integridade de dados recebidos e sugerem uma solução distribuída para coletar métricas de QoS e auditar contratos de nível de serviço.

Por fim, [Jamous et al. 2020] se aproximam do proposto no atual estudo ao utilizar o *Software Guard Extensions* na detecção de quebras de contrato de SLA. O trabalho coloca em pauta questões de confidencialidade de dados e cumprimento de contrato de nível de serviço. Apesar disso, as soluções apresentadas dão muito mais enfoque à automação do processo de manutenção do SLA e adotam a *blockchain* e os contratos inteligentes como tecnologias centrais às arquiteturas apresentadas.

Os artigos selecionados ressaltam a relevância de monitoramento confidencial e acompanhamento de SLA, destacam desafios e soluções de implementação de sistemas de monitoramento em nuvem e tangenciam, de maneira complementar, a proposta do atual trabalho. Constatou-se uma oportunidade de pesquisa acerca da utilização da arquitetura SGX para monitoramento confidencial e seguro de métricas de qualidade de serviço.

3. Arquitetura Proposta

Sistemas de monitoramento em nuvem (CMS – *Cloud Monitoring Systems*) são usualmente baseados no padrão *agente-gerente*, no qual pequenos agentes executam junto aos serviços monitorados, coletando dados sobre sua operação. Esses dados são então enviados a um gerente remoto, responsável por processar, armazenar e usar a informação coletada. O

modelo de interação entre agentes e gerente pode ser *push*, em que cada agente decide quando enviar dados ao gerente, *pull*, em que o gerente é responsável por solicitar os dados aos agentes, ou *híbrido*, quando combina caraterísticas de ambos.

A arquitetura proposta neste estudo foi definida com base no desenho sugerido em [Alhamazani et al. 2014]. Naquele trabalho, os autores desenvolvem uma topologia agente-gerente que visa monitorar múltiplas camadas da nuvem simultaneamente (partindo de infraestrutura até software) e de maneira granular a cada aplicação. Embora versátil, aquela plataforma assume que o sistema operacional é confiável e não adota quaisquer medidas para garantir a segurança dos dados observados.

A Figura 1 traz uma visão geral da arquitetura proposta. Nela, uma aplicação qualquer (denotada por "Aplicação 1") é executada em um contêiner e faz uso de recursos computacionais da máquina subjacente para executar suas tarefas. Em paralelo, um agente de monitoramento coleta dados sobre o funcionamento do contêiner e os armazena temporariamente até que esses dados sejam recolhidos pelo gerente. Este, por sua vez, faz requisições esporádicas a todos os agentes pré configurados e persiste as métricas coletadas em um banco de dados centralizado, onde podem ser consultadas. A interação é baseada em um modelo *push-pull* híbrido [Birje and Bulla 2019], buscando facilitar a configuração de novos agentes e manter a capacidade de escala do sistema.

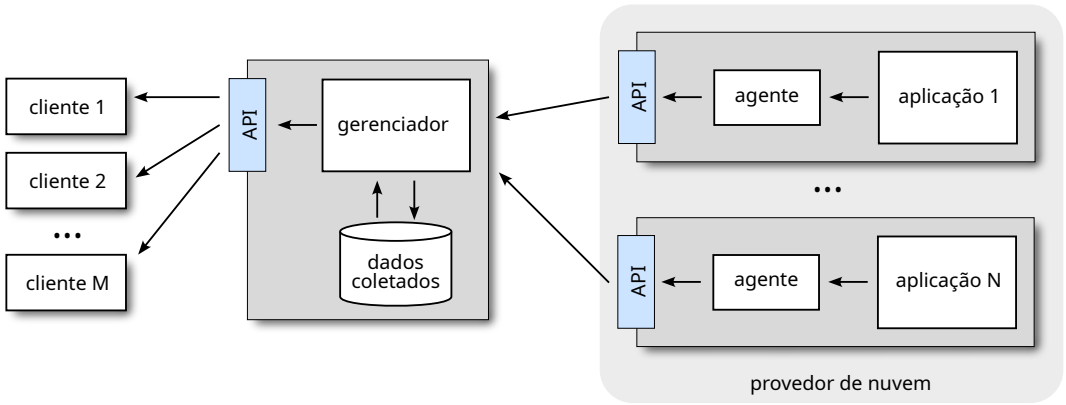

Figura 1. Arquitetura proposta do sistema.

3.1. Modelo de Ameaças

A topologia do sistema proposto assume que uma entidade maliciosa tenha permissão de super usuário na máquina que hospeda a aplicação em nuvem, e também que tenha acesso físico aos componentes dessa máquina. Isso posto, conclui-se que todos os recursos e aplicações provisionados em nuvem fazem parte da superfície de ataque do sistema de monitoramento e devem ser considerados não confiáveis. Considera-se, todavia, que o trecho do sistema responsável por gerenciar os agentes e armazenar dados é inerentemente seguro e está sob controle do usuário. Assim, o modelo de riscos engloba apenas a infraestrutura provisionada para hospedar a aplicação a ser monitorada e desconsidera o gerenciador do sistema de monitoramento.

Para garantir que as métricas coletadas pelos agentes sejam íntegras e que as informações processadas e armazenadas sejam confidenciais, é crítico que os requisitos levantados a seguir sejam implementados:

1. Durante a execução do agente, dados em processamento não podem ser acessados pela entidade maliciosa, estejam esses presentes nos registradores do processador, em memória ou em disco. Em conjunto, o binário do agente (ou código escrito, no caso de linguagens interpretadas), as variáveis de ambiente por ele usadas ou valores por ele movimentados também devem estar fora de alcance, e métricas de sistema referentes à utilização de memória e capacidade de processamento devem ser consideradas confiáveis.
2. No momento de provisionamento do contêiner que hospeda a aplicação monitorada e o agente de monitoramento, parâmetros de configuração bem como o código do agente não podem ser transmitidos ou observados pelo provedor. Falhar em atender esse preceito resulta na possibilidade de o atacante adulterar a configuração do contêiner ou o executável do agente que nele reside.
3. Caso interceptados, dados que transitam para o gerenciador devem estar criptografados e inacessíveis. É exigido, ademais, que apenas o gerenciador seja capaz de requisitar as informações coletadas pelos agentes.

3.2. Tecnologias e escolhas de arquitetura

Para satisfazer o requisito "1"apresentado no modelo de ameaças, decidiu-se usar o mecanismo de contêiner seguro do SCONE para isolar o agente dos demais processos da máquina. Além de reduzir o custo de desenvolvimento do Intel SGX padrão, essa tecnologia busca também minimizar a base computacional confiável *(Trusted Computing Base* ou TCB) que reside dentro de um enclave [Arnautov et al. 2016].

Com Intel SGX nativo, quaisquer interações com recursos externos ao enclave exigem que chamadas de saída e entrada sejam executadas. Isso implica, por exemplo, na necessidade de executar diversas trocas de contexto sequencialmente para completar tarefas triviais como acesso a arquivos externos ou conexões de rede. O SCONE remove essa sobrecarga de ações descrita ao substituir a biblioteca padrão da linguagem C por uma versão própria, ligada estaticamente aos binários de dentro do enclave. Chamadas de sistema continuam a ser executadas fora do enclave, mas a biblioteca padrão modificada não as faz diretamente. Em vez disso, ela copia parâmetros dos métodos executados para fora do ambiente seguro e resultados para dentro. Além disso, argumentos persistidos fora do enclave podem ser protegidos por criptografia e, para o desenvolvedor, a utilização da biblioteca padrão permanece idêntica [Arnautov et al. 2016].

Além disso, a ferramenta dispõe do Serviço de Atestação e Configuração (CAS – *Configuration and Attestation Service*) e do Serviço de Atestação Local (LAS – *Local Attestation Service*), responsáveis por injetar com segurança quaisquer argumentos de configuração do contêiner na inicialização, de forma a ocultar esses parâmetros do provedor de nuvem que hospeda a aplicação [Brito et al. 2020], o que garante o requisito "2"do modelo de ameaças.

Finalmente, os chamados *shields* do SCONE possibilitam a criptografia e autenticação para acesso de arquivos e também a cifragem de canal de comunicação via TLS [Arnautov et al. 2016]. Sua utilização assegura assim o cumprimento do requisito "3"do modelo de ameaças, uma vez que implementam autorização ao acesso de certificados usados pelo protocolo TLS, bem como o uso do próprio protocolo na transmissão privativa de dados. No gerenciador são configuradas as rotas e chaves de API que restringem a ele a permissão de requisitar dados aos agentes.

Além das decisões pautadas pelos requisitos de segurança, definiu-se por colocar o agente de monitoramento dentro da máquina virtual em que a aplicação é executada. Apesar de essa ser uma decisão que facilita a adulteração dos dados processados pelo agente, ela expande o leque de métricas passíveis a serem monitoradas. Um agente completamente isolado, por exemplo, não teria acesso ao uso de recursos da máquina ou informações de processos que nela são executados. Restaria, enfim, um monitoramento de disponibilidade ou outros SLAs de mais alto nível.

4. Especificação das APIs

Foram implementadas duas interfaces de aplicação distintas. A primeira, de agente de monitoramento, expõe as métricas mais recentes coletadas por um monitor específico. A interface do gerenciador, por sua vez, centraliza os dados obtidos na comunicação com as rotas de agentes e persistidos em memória primária e os apresenta ao cliente.

4.1. Agente de monitoramento

A depender da estratégia adotada, o processo de coleta de dados para monitoramento pode resultar em um alto consumo de recursos de rede, persistência de dados indesejados, dificuldades para provisionar novos agentes ou problemas de sincronização entre processos [Tharunya et al. 2016]. Assim, optou-se pela implementação de uma API específica para comunicação entre *n* agentes de monitoramento e um agregador.

A abordagem, similar à adotada por [Alhamazani et al. 2014], demanda um arquivo de configuração no gerenciador que contém as informações relevantes para o uso das interfaces disponibilizadas pelos agentes. Essas, por sua vez, implementam uma única rota que pode ser utilizada pelo gerenciador para obter as métricas de porcentagem de uso de recursos de processamento em um intervalo de tempo, e também as informações de uso de memória primária no instante exato da requisição.

Para expor ao gerenciador os dados recolhidos de uma nova aplicação *cloud*, a API descrita deve ser provisionada na máquina que hospeda tal aplicação e, portanto, reside dentro de um contêiner seguro. Essa interface é acessível pelo protocolo HTTPS em uma porta pré configurada; a adoção do protocolo TLS garante que dados transmitidos são criptografados, atendendo ao item ”3”do modelo de ameaças. Visando acesso exclusivo por parte do gerenciador às APIs de agentes monitores, faz-se necessária a transmissão de uma chave de API pelo *header* `X-Api-Key`, essa também previamente configurável.

A interface de aplicação de monitoramento dispõe de apenas uma rota `/metrics` acessível pelo método HTTP `GET`, usada para recuperar os dados referentes ao agente que tornou pública aquela API. Ademais, é obrigatório que o parâmetro de busca `?quantity=` seja especificado, posto que determina o número de métricas retornadas, ordenadas pela mais recente.

Como exposto na Seção 5.2, as informações apresentadas pelo caminho descrito são derivadas de um código perene executado no sistema a ser monitorado. É conferido à métrica um identificador único universal (UUID) [Leach et al. 2005] com fim de singularizar o registro, o horário de coleta da informação no formato de data ISO 8601 para permitir ordenação temporal dos dados e, enfim, um identificador único a cada sistema monitorado pelo agente, configurado em momento anterior.

Nas Tabelas 2 e 3 são especificados a rota da API de agentes de monitoramento e seus possíveis retornos. Na Tabela 4 é definido e exemplificado o modelo de métricas adotado (essa API retorna uma lista de *n* métricas ordenadas da mais recente à mais antiga).

Método HTTP	**Rota**	**Função**
`GET`	`/metrics?quantity=`*n*	Retornar *n* métricas mais recentes do agente

Tabela 2. Métodos da API dos agentes de monitoramento e respectivas funções.

Código HTTP	**Status**	**Significado na API**
200	*OK*	Métricas retornadas com sucesso
204	*No Content*	Não há métricas a retornar
400	*Bad Request*	Parâmetro de busca `quantity` deve ser maior que 0
401	*Unauthorized*	Chave `X-Api-Key` inválida
500	*Internal Server Error*	Problema desconhecido

Tabela 3. Possíveis códigos de retorno da API dos agentes de monitoramento.

Chave	**Tipo**	**Exemplo**	**Significado**
`id`	string	`b57426fb-1502-41fc-af2e-3d8cafbf3b92`	UUID único
`agentId`	string	`0`	Id pré configurado do agente a quem pertencem as métricas
`cpu`	string	`1.80`	Porcentagem de tempo em que o processador não ficou ocioso no intervalo analisado
`memory`	string	`2.00`	Uso de memória, em Gigabytes, no momento da requisição
`datetime`	string	`2024-12-09T23:42:22.981Z`	Horário da coleta de dados no formato ISO 8601

Tabela 4. Modelagem da entidade de métrica exemplificada.

4.2. Gerenciador

O gerenciador consome as APIs de agentes à procura de informações acerca dos sistemas monitorados. O processo ocorre de maneira concorrente e intervalada com sobreposição de dados coletados (método descrito na Seção 5.3). Uma vez que as métricas são agregadas e persistidas, faz-se indispensável o acesso a essas métricas por parte do cliente para, por exemplo, exibir em uma aplicação de visualização, alimentar modelos preditivos, alertas, dentre outras finalidades. Esse é, precisamente, o papel da API do gerenciador.

Diferentemente das interfaces de agente, o gerenciador é hospedado em um ambiente separado do restante da aplicação e, portanto, considerado seguro. Dessa forma, não é necessário que execute dentro de um contêiner seguro. Bem como as demais APIs implementadas, a rota do gerenciador também é acessível pelo protocolo *HTTPS*, com enfoque à implementação de TLS visando criptografia na transmissão de informações.

A porta de acesso da interface e a chave de API necessária para o acesso são definidas na etapa de configuração e provisionamento do nó agregador. Em específico, é

usado o método `GET` na rota `/manager/metrics` para recolher o agregado de métricas coletadas. As Tabelas 5 e 6 detalham a rota da API de gerenciador e listam seus possíveis códigos de retorno, respectivamente. A API do gerenciador retorna a lista de todas as métricas (Tabela 4) coletadas e persistidas até o momento.

Método HTTP	**Rota**	**Função**
`GET`	`/manager/metrics`	Retornar métricas persistidas de todos os agentes

Tabela 5. Métodos da API do gerenciador e respectivas funções.

Código HTTP	**Status**	**Significado na API**
200	*OK*	Métricas retornadas com sucesso
401	*Unauthorized*	Chave `X-Api-Key` inválida
500	*Internal Server Error*	Problema desconhecido

Tabela 6. Possíveis códigos de retorno da API do gerenciador.

5. Implementação

Nesta seção, especificam-se alguns detalhes do uso do SCONE e da implementação do agente e do gerenciador de monitoramento. Menciona-se também as limitações do sistema criado e são abordadas algumas dificuldades encontradas no desenvolvimento do projeto.

5.1. Utilização do SCONE

O uso do SCONE garantiu que o código executado em nuvem fosse criptografado e, portanto, inacessível ao provedor. Dados que trafegam nos registradores do processador e informações de execução presentes em memória também tornam-se confidenciais, e a adoção da ferramenta contribuiu ainda para uma *Trusted Computing Base* concisa [Arnautov et al. 2016].

Em termos práticos e gerais, o processo de construção do agente de monitoramento foi caracterizado pela execução das seis etapas enumeradas abaixo e, para cada nova aplicação a ser monitorada, deve-se realizar o *build* na máquina da aplicação e assim provisionar os contêineres seguros do agente naquela máquina:

1. identificar o diretório do *device* do Intel SGX.
2. limpar artefatos resultantes de execuções anteriores.
3. instalar dependências e extrair o executável modificado do Python de uma imagem do SCONE.
4. gerar o *File System Protection File* (FSPF) que define quais regiões do sistema de arquivos serão protegidas pelo *shield* do SCONE.
5. gerar chaves de criptografia para uso específico dos agentes de monitoramento.
6. construção da imagem Docker e execução dos dois contêineres seguros (do agente e de sua API).

Inicialmente, deve-se identificar onde está localizado o *device* do SGX a depender da versão do *driver* da Intel instalado no sistema. Para tal, usou-se o código disponível na própria documentação do SCONE [Scontain 2024a]. Depois, é criada na máquina hospedeira um caminho destinado aos artefatos resultantes do processo de *build*.

Em seguida, é provisionado um contêiner para instalação das bibliotecas utilizadas pelo projeto. Para tal, cria-se um ambiente virtual com o *Python venv* sobre uma instalação Python padrão fornecida pela imagem utilizada no contêiner. O *Python venv* é usado para isolar as dependências de um projeto em uma máquina virtual, e o uso dessa tecnologia está relacionado ao GCC não ter sido encontrado inicialmente, mas traz consigo a vantagem de padronizar o ambiente de instalação das dependências, simplificando assim a verificação de integridade dos arquivos ali presentes. Em seguida, um segundo contêiner é provisionado com a imagem do Python modificada pelo SCONE e esse executável é copiado para o diretório de artefatos. Dessa maneira ficam disponíveis, nos artefatos, todas as bibliotecas necessárias e o executável Python compatível com SCONE.

A estrutura de arquivos do projeto é, então, copiada para a pasta de artefatos para ser cifrada pelo *shield* do SCONE. Isso é feito por meio do FSPF, usado para determinar as regiões do *filesystem* a serem criptografadas. Esse processo resulta em uma chave que pode ser fornecida ao SCONE no momento de provisionamento da aplicação [Scontain 2024b]. Em outra etapa é criada pela biblioteca *Libnacl* outra chave criptográfica usada pelo agente monitor para criptografar o arquivo de métricas (como descrito na Seção 5.2).

Um *Dockerfile* é usado para copiar os artefatos gerados a uma nova imagem *Docker* e dois contêineres seguros (do agente e de sua API) são provisionados usando essa mesma imagem. São fornecidas às aplicações as chaves criptográficas mencionadas e variáveis de configuração do agente de monitoramento. Também é realizado o mapeamento entre porta da máquina hospedeira e do contêiner para comunicação da API e são montados os diretórios onde é escrito o arquivo de métricas e de arquivos relevantes para obtenção dessas métricas.

Antes da última etapa, era esperado o uso dos serviços CAS e LAS do ambiente SCONE para mitigar o risco de vazamento das variáveis de configuração do sistema ao provedor *cloud*, como descrito na proposta. Entretanto, por problemas durante a implementação (especificados em 5.5), a ferramenta não foi integrada ao projeto.

5.2. Implementação do agente de monitoramento

A organização do sistema conjuga controladores (que orquestram a execução da aplicação em geral e são o ponto de entrada do código executado) e modelos (instanciados nos controladores, mantém em si variáveis de estado da aplicação e contêm implementações específicas de cada domínio). Em simultâneo, são executados em dois processos distintos o controlador do agente de monitoramento e o controlador de sua API. Como o segundo já foi amplamente descrito na Seção 4.1, aqui são apresentados os detalhes do primeiro.

O código do controlador do agente extrai parâmetros de configuração de variáveis de ambiente e checa se as configurações esperadas foram definidas e atendem às restrições impostas (por exemplo, o diretório em que será salvo o arquivo de métricas coletadas precisa existir). Em seguida, um objeto da classe `agente` é instanciado e inicializado; ele, por sua vez, periodicamente coleta e persiste as medições realizadas.

Uma vez iniciado, o modelo do agente de monitoramento executa um laço contínuo que aguarda s segundos e, então, executa as medições de uso de memória RAM e percentual de utilização do CPU desde a última mensuração. A implementação inicial (sem SCONE) utilizava *threads* para evitar que a execução do sistema ficasse presa nos intervalos da medição. Entretanto, essa abordagem aparentou ser incompatível com SGX.

Para se obter as informações apresentadas, foram usados os pseudo-arquivos `/proc/meminfo` e `/proc/stat` específicos dos sistemas baseados em UNIX, mas a fonte desses dados pode ser facilmente ajustada para outros sistemas. São sugeridos na literatura métodos alternativos para medição de estatísticas em nuvem [Liu 2011]. Todavia, essas abordagens fogem do escopo do proposto aqui.

As informações coletadas são então anexadas a um arquivo de métricas criado pelo monitor e compartilhado com a máquina hospedeira e com o processo da API do agente. Apesar de a abordagem usar memória em disco e não memória volátil, essa decisão se justifica por ser a mais simples no contexto de comunicação entre processos do SCONE, enquanto atende aos requisitos de segurança da proposta. As métricas são criptografadas antes de serem escritas e traduzidas quando a API as consulta.

Como uma nova chave de cifragem de métricas é sempre gerada na criação da imagem do agente, reiniciar a execução do contêiner do controlador de monitoramento implica na necessidade de recriar o arquivo de métricas. Tem-se como pressuposto, pois, que esse arquivo é temporário e assume um papel similar ao de um *buffer* até que os dados ali contidos sejam persistidos pelo gerenciador.

5.3. Implementação do gerenciador

Bem como no caso do agente de monitoramento, o funcionamento do gerenciador depende da execução de dois processos: um para coletar e persistir os dados dos agentes periodicamente e outro para expor esses dados, via API, a um cliente. Como o código do gerenciador é executado em um ambiente seguro e, por pressuposto, não necessita do SCONE, o processo de criação da imagem do ambiente e gerenciamento dos contêineres se aproxima do convencional. Assim, foi suficiente usar um arquivo *Dockerfile* e os comandos de *Docker Compose* nas tarefas descritas. Usou-se o *Compose* também para provisionar uma instância do *Redis*, um banco de dados em memória de estruturas chave-valor, por meio da imagem disponibilizada pelo próprio *Docker* [Docker 2024].

A lógica implementada no gerenciador também faz checagens e validações acerca de variáveis de ambientes recebidas e, tal qual o agente de monitoramento, instancia um modelo específico para armazenar o estado da execução. O modelo do gerenciador, entretanto, utiliza *threads* de execução em intervalos definidos para percorrer todas as rotas de agentes disponíveis e persistir, então, os dados coletados em um banco de dados em memória não relacional.

A lista de domínios de agentes a serem verificados é configurada antes da execução do gerenciador. São configurados também o número de métricas lidas de cada agente por consulta e o intervalo entre essas varreduras. Como o retorno da API de agentes é ordenado pela data de coleta dos dados, buscar n registros de um agente se traduz em buscar os n registros mais recentes daquele agente. Assim, nas configurações padrões do gerenciador, existe sobreposição entre as informações retornadas em uma varredura e as informações retornadas na próxima. Essa estratégia garante que uma falha pontual em uma consulta não resultará na perda de dados, já que parte das informações perdidas estarão presentes na próxima consulta, ou estiveram presentes em consultas anteriores.

O banco de dados é então acessado por uma interface específica no momento do armazenamento de métricas. Por padrão, novas conexões são automaticamente gerenciadas por um *pool* de conexões, e informações de rede necessárias para conexão também figuram

no arquivo de configurações do gerenciador. Usando seus identificadores, métricas já encontradas na base são sobrescritas e métricas não encontradas são inseridas (operação denominada de *upsert*). Por isso, a sobreposição descrita no parágrafo anterior não incorre em duplicidade dos dados armazenados. Enfim, quando requisitada, a API do gerenciador consulta a instância provisionada do *Redis* e retorna ao cliente o que foi solicitado.

5.4. Limitações

Com a descrição dos componentes do sistema, ficam aparentes algumas limitações da implementação do projeto. Apesar da compatibilidade exclusiva dos agentes com sistemas Unix (dado que a coleta de métricas é feita através do acesso aos arquivos `/proc/meminfo` e `/proc/stat` exclusivos de distribuições Unix), essa não é listada como limitação. Isso porque seria possível executar o monitor em outros sistemas operacionais, com os necessários ajustes no código.

Com relação à configurações de ambiente é válido lembrar que a execução do agente de monitoramento depende da compatibilidade do CPU com Intel SGX e, por conseguinte, a aplicação a ser monitorada precisa ser executada em uma máquina que suporte essa tecnologia - mesmo que não faça uso direto da mesma.

Já na esfera de segurança, o trabalho limita-se ao proposto. Detalha-se, entretanto, alguns ataques de negação de serviço específicos à implementação proposta. Um superusuário poderia, por exemplo, apagar ou corromper o arquivo de métricas de um agente ou, também, quaisquer arquivos protegidos pelo *shield* do SCONE. Partindo desse prisma, o mesmo agente malicioso poderia inclusive interromper o acesso da máquina do agente à rede. Pela vastidão do tema de mitigação de ataques DoS e pelo fato do próprio SCONE desconsiderar essa vulnerabilidade em seu modelo de ameaça [Arnautov et al. 2016], tratar esses ataques foge da alçada deste estudo.

5.5. Falha na Atestação do CAS

Durante a utilização do CAS, a ferramenta retornou uma mensagem cujo conteúdo indicava falha na verificação do SCONE *Quote*, posto que o *nonce* do registro esperado do *SCONE Quoting Enclave* nunca havia sido enviado e, por isso, o registro não poderia ser verificado. Paralelamente, o CAS retornava também um erro relacionado às versões do CAS, embora todas as versões descritas coincidissem.

Pelo apresentado, suspeita-se que o problema pode ser decorrência da interação entre as diferentes versões de *drivers* e kernel do ambiente utilizado com versões das imagens disponíveis pelo SCONE, carência de alguma configuração no ambiente de desenvolvimento ou problema na utilização do CAS. Suspeita-se, ainda, que os erros podem ter sido ocasionados por problema nas imagens ou no CAS público disponibilizados pelo projeto SCONE. Pelo exposto, a integração do CAS e LAS ao CMS desenvolvido fica atribuída a trabalhos futuros, dado que a utilização dessas tecnologias já foi explorada em [Brito et al. 2020] e [Solis 2022].

6. Avaliação

Visando avaliar o desempenho do sistema desenvolvido e discutir os resultados obtidos, apresenta-se nesta seção a análise de complexidade espacial e temporal dos principais métodos implementados e a sobrecarga imposta ao tempo de execução de código devido ao uso do SCONE.

6.1. Desempenho e complexidade

Um dos métodos centrais para o funcionamento do agente monitor é o responsável por coletar e persistir métricas (`__collect_and_persist_data`). Essa função, subdividida em três outras, consulta os arquivos do sistema que registram o uso de processamento e memória para, então, calcular e anexar os dados ao arquivo de registros coletados.

Por conta da rotina de coleta independer da quantia de métricas já armazenadas, o tempo de execução desse método é constante para qualquer quantidade de registros previamente coletados. O custo de armazenamento das métricas coletadas (desconsiderando metadados do arquivo) em função do tempo de execução do processo pode ser descrito por $f(t) = \lfloor (t/i) \rfloor s$, onde t representa o tempo decorrido do início da execução do agente, i o intervalo de tempo entre coletas e s o espaço em disco ocupado por cada registro.

Em paralelo, a rotina de consolidação dos dados de diferentes agentes no gerenciador do sistema apresenta complexidade temporal de $O(hn)$, onde h é o número de *hosts* (ou agentes) a serem monitorados e n é o número de métricas recuperadas de cada agente. A complexidade espacial desse método, em consequência, é de $O(h + n)$ ou $O(max(h, n))$, dado que faz-se necessário armazenar em memória a lista de agentes e de métricas do agente visitado para percorrê-los.

Em ambas as APIs (monitor e gerenciador), tanto a complexidade de tempo quanto a de espaço são de $O(n)$, em que n é o número de métricas retornadas pela aplicação. Essa complexidade advém do custo de percorrer todas as métricas para realizar uma operação criptográfica e de armazenar esses registros em memória para retorná-los ao cliente.

6.2. Sobrecarga do SCONE

O desenvolvimento do projeto contemplou a criação de módulos separados, integração dos módulos em uma aplicação funcional e adoção do SCONE e suas tecnologias, respectivamente. Assim sendo, tornou-se possível executar o sistema também sem SGX ao ignorar as mudanças realizadas na etapa de adoção do SCONE. Dessa maneira, foram executados processos do agente de monitoramento com e sem a camada do SCONE visando comparar a sobrecarga no tempo de execução introduzida no sistema por essa tecnologia.

Para realizar a mensuração, foi incorporado ao projeto um *decorator* cuja tarefa é gravar em um arquivo o tempo necessário para executar determinada função. Na linguagem Python, *decorators* são comumente utilizados para modificar a funcionalidade de um método sem alterá-lo diretamente, sendo ideais para essa tarefa.

Inicialmente, executou-se (sem SCONE) um agente de monitoramento por tempo o suficiente para que 1.200 registros fossem coletados e persistidos no arquivo de métricas. Em seguida, replicou-se o mesmo para um agente de monitoramento, porém esse executando em um contêiner seguro. A seguir, foram confrontadas a média dos tempos de execução e respectivos desvios-padrão das amostras das principais funções de coleta de ambos os cenários para, enfim, obter-se a sobrecarga imposta ao tempo de execução do código pelo uso do SCONE (Tabela 7).

Os dados obtidos indicam que a adoção do SCONE proporcionou uma sobrecarga significativa ao tempo de execução das funções especificadas. No pior cenário, observou-se que o tempo de execução de um método em um contêiner seguro ultrapassou em mais que 10 vezes o tempo de execução desse método fora do contêiner SCONE (um aumento de

Método	Tempo médio sem SCONE	Tempo médio com SCONE	Sobrecarga do SCONE
`__update_cpu_info`	$0,30 \pm 0,05$ ms	$3,05 \pm 1,10$ ms	910%
`__calculate_cpu_usage_pct`	$0,00 \pm 0,00$ ms	$0,02 \pm 0,05$ ms	-
`__get_memory_info`	$0,33 \pm 0,08$ ms	$1,61 \pm 0,36$ ms	386%
`__collect_and_persist_data`	$1,82 \pm 0,73$ ms	$9,65 \pm 1,81$ ms	430%

Tabela 7. Sobrecarga no tempo de execução de métodos imposta pelo SCONE.

910%). Ao comparar os resultados do presente trabalho com os obtidos por [Solis 2022] - no qual o uso do SCONE introduziu, no pior cenário, um *overhead* temporal de 480% -, sugere-se que o acesso a arquivos em disco possa intensificar a degradação de desempenho decorrente do uso do SCONE, uma vez que a maior parte dos métodos aqui analisados consultam informações no sistema de arquivos da máquina *host*.

Apesar dos resultados obtidos, o aumento do tempo de execução do código do agente de monitoramento não invalida ou compromete a viabilidade de um sistema de monitoramento confidencial em nuvem. Isso porque, nessa proposta, as medições acontecem em intervalos muito maiores do que o tempo necessário para coletar as métricas. Contanto que o processo de coleta tome menos tempo que o intervalo entre coletas (esse, por padrão, de 10 segundos), o sistema implementado é viável.

7. Considerações Finais

No presente trabalho propôs-se uma abordagem para o desenvolvimento de um CMS de métricas de SLA confidencial. Em detalhe, foi sugerida uma arquitetura de monitoramento *push-pull* híbrida baseada em [Alhamazani et al. 2014], cujo principal propósito era mitigar as vulnerabilidades mapeadas no modelo de ameaças desenvolvido, visando a confidencialidade e integridade dos dados coletados pelo sistema.

Como resultado, implementou-se o sistema proposto utilizando contêineres seguros - implementação essa que engloba a coleta de métricas, comunicação entre agente e gerenciador, consolidação dos dados no gerenciador e as respectivas APIs responsáveis por tornar as métricas acessíveis ao cliente. As APIs desenvolvidas foram especificadas no atual trabalho e o código desenvolvido foi versionado em um repositório público no GitHub [Curvelo 2025], possibilitando assim que futuros trabalhos explorem as ideias aqui apresentadas.

Avaliou-se por fim o desempenho dos principais métodos desenvolvidos na aplicação em termos de complexidade temporal e espacial, e também foram realizadas comparações entre o tempo médio de execução dessas rotinas nos contêineres seguros e fora deles. Assim, constatou-se que o impacto do SCONE no desempenho da coleta de métricas é considerável, mas pelo exposto na Seção 6.2, não invalida a viabilidade do sistema criado.

Em decorrência do empecilho apresentado na seção 5.5, um ponto de partida proposto a trabalhos futuros seria a integração do sistema já desenvolvido ao CAS e LAS levando em conta o exposto nesse estudo, e também a implementação de [Brito et al. 2020]. Ainda na esfera de segurança e com fim de evitar vazamento de chaves, o trabalho abre margem para a utilização do protocolo de autenticação mútua (mTLS) na comunicação de

APIs no lugar das chaves de API adotadas na presente proposta [Siriwardena 2014].

A atestação da autenticidade dos dados no momento de coleta (como discutido em [Robin and Irvine 2000] e [Lucas et al. 2017]) também fugiu do escopo do estudo e é um tema que poderia ser explorado adiante. Como citado no trabalho, a utilização de enclaves e os contêineres seguros do SCONE também não garantem a mitigação de ataques de negação de serviço ou de canal lateral. Assim, existe espaço para adaptar ao sistema abordagens semelhantes a [Oleksenko et al. 2018] ou [Zhao et al. 2020], por exemplo.

Referências

[Alhamad et al. 2011] Alhamad, M., Dillon, T., and Chang, E. (2011). A survey on SLA and performance measurement in cloud computing. In *Confederated Intl Conferences: CoopIS, DOA-SVI, and ODBASE*, pages 469–477. Springer.

[Alhamazani et al. 2014] Alhamazani, K., Ranjan, R., Mitra, K., Jayaraman, P., Huang, Z., Wang, L., and Rabhi, F. (2014). Clams: Cross-layer multi-cloud application monitoring-as-a-service framework. In *Intl Conference on Services Computing*, pages 283–290. IEEE.

[Arnautov et al. 2016] Arnautov, S., Trach, B., Gregor, F., Knauth, et al. (2016). SCONE: Secure linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 689–703.

[Badshah et al. 2023] Badshah, A., Jalal, A., Farooq, U., Rehman, G.-U., Band, S. S., and Iwendi, C. (2023). Service level agreement monitoring as a service: an independent monitoring service for service level agreements in clouds. *Big Data*, 11(5):339–354.

[Birje and Bulla 2019] Birje, M. N. and Bulla, C. (2019). Cloud monitoring system: basics, phases and challenges. *Intl Journal of Recent Technology Engineering (IJRTE)*, 8(3):4732–4746.

[Brito et al. 2020] Brito, A., Souza, C., Silva, F., Cavalcante, L., and Silva, M. (2020). Processamento confidencial de dados de sensores na nuvem. In *Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg)*.

[Costan and Devadas 2016] Costan, V. and Devadas, S. (2016). Intel SGX explained. *Cryptology ePrint Archive*.

[Curvelo 2025] Curvelo, J. P. (2025). Monitoramento Seguro de Métricas de Nível de Serviço em Nuvem. `https://github.com/Joao1905/sgx`.

[De Carvalho et al. 2017] De Carvalho, C., de Castro Andrade, R., de Castro, M., Coutinho, E., and Agoulmine, N. (2017). State of the art and challenges of security SLA for cloud computing. *Computers & Electrical Engineering*, 59:141–152.

[Docker 2024] Docker (2024). REDIS oficial images. `https://hub.docker.com/_/redis`.

[Dong et al. 2023] Dong, C., Shen, Q., Ding, X., Yu, D., Luo, W., Wu, P., and Wu, Z. (2023). T-counter: Trustworthy and efficient CPU resource measurement using SGX in the cloud. *IEEE Transactions on Dependable and Secure Computing*.

[Jain et al. 2016] Jain, P., Desai, S., Shih, M.-W., Kim, T., Kim, S., Lee, J.-H., Choi, C., Shin, Y., Kang, B., and Han, D. (2016). OpenSGX: An open platform for SGX research. In *Network and Distributed System Security Symposium (NDSS)*, volume 16, pages 21–24.

[Jamous et al. 2020] Jamous, N., Volk, M., Barouqa, H., Ghnaim, F., and Turk, T. (2020). Towards smart service level agreement (SSLA) using blockchain. In *Pacific Asia Conf. on Information Systems*.

[Leach et al. 2005] Leach, P., Mealling, M., and Salz, R. (2005). RFC 4122: A universally unique identifier (UUID) URN namespace.

[Liu 2011] Liu, H. (2011). A measurement study of server utilization in public clouds. In *9th Intl Conference on Dependable, Autonomic and Secure Computing*, pages 435–442. IEEE.

[Loch et al. 2021] Loch, W., Koslovski, G., Pillon, M., Miers, C., and Pasin, M. (2021). A novel blockchain protocol for selecting microservices providers and auditing contracts. *The Journal of Systems & Software*.

[Lucas et al. 2017] Lucas, P., Chappuis, K., Paolino, M., Dagieu, N., and Raho, D. (2017). Vosysmonitor, a low latency monitor layer for mixed-criticality systems on ARMv8-A. In *29th Euromicro Conference on Real-Time Systems (ECRTS)*.

[Nguyen and Ganapathy 2017] Nguyen, H. and Ganapathy, V. (2017). Engarde: mutually-trusted inspection of SGX enclaves. In *37th Intl Conference on Distributed Computing Systems (ICDCS)*, pages 2458–2465. IEEE.

[Oleksenko et al. 2018] Oleksenko, O., Trach, B., Krahn, R., Silberstein, M., and Fetzer, C. (2018). Varys: Protecting SGX enclaves from practical Side-Channel attacks. In *USENIX Annual Technical Conference*, pages 227–240, Boston, MA. USENIX Association.

[Park et al. 2024] Park, J., Kang, S., Lee, S., Kim, T., Park, J., Kwon, Y., and Huh, J. (2024). Hardware-hardened sandbox enclaves for trusted serverless computing. *ACM Transactions on Architecture and Code Optimization*, 21(1):1–25.

[Robin and Irvine 2000] Robin, J. S. and Irvine, C. E. (2000). Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *9th USENIX Security Symposium*.

[Scontain 2024a] Scontain (2024a). Installing intel SGX Driver - determine SGX device. `https://sconedocs.github.io/sgxinstall/#determine-sgx-device`.

[Scontain 2024b] Scontain (2024b). Scone file protection. `https://sconedocs.github.io/SCONE_Fileshield/`.

[Siriwardena 2014] Siriwardena, P. (2014). Mutual authentication with TLS. *Advanced API Security: Securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE*, pages 47–58.

[Solis 2022] Solis, T. (2022). *Adicionando segurança, proveniência e gerenciamento de dados demográficos ao OpenEHR*. Dissertação de mestrado, Universidade Federal do Paraná.

[Tharunya et al. 2016] Tharunya, S., Divya, M., and Shunmuganathan, K. (2016). A multi-agent based query processing system using RETSINA with intelligent agents in cloud environment. In *Intl Conference on Computing Technologies and Intelligent Data Engineering (ICCTIDE)*, pages 1–6. IEEE.

[Zhan et al. 2021] Zhan, D., Tan, K., Ye, L., Yu, H., and Liu, H. (2021). Container introspection: using external management containers to monitor containers in cloud computing. *Computers, Materials & Continua*, 69(3):3783–3794.

[Zhao et al. 2020] Zhao, W., Lu, K., Qi, Y., and Qi, S. (2020). MPTEE: bringing flexible and efficient memory protection to Intel SGX. In *5th European Conference on Computer Systems (EuroSys)*, New York, NY, USA. ACM.