Ordenação Segura de Eventos na Periferia da Rede

Cláudio Correia, Miguel Correia e Luís Rodrigues
{claudio.correia, miguel.p.correia, ler}@tecnico.ulisboa.pt

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

Resumo A computação na periferia é um paradigma que estende a computação na nuvem com armazenamento e processamento próximos dos utilizadores, suportando aplicações que precisam de baixa latência como realidade aumentada ou jogos móveis. Os nós que fornecem estes serviços, que designamos por cúmulos, estão mais vulneráveis a modificações ilegítimas do que os nós localizados em grandes centros de dados, sendo crucial proteger as suas funções de falhas maliciosas. Neste artigo apresentamos um serviço seguro de ordenação de eventos para cúmulos. O serviço, denominado Omega, tira partido de hardware confiável baseado na tecnologia Intel SGX para oferecer aos clientes garantias quanto à ordem em que seus eventos são aplicados e servidos, até mesmo na eventualidade do cúmulo se encontrar comprometido. Adicionalmente, apresentamos o OmegaCV, um sistema de armazenamento chave-valor que usa o Omega para oferecer causalidade. Resultados experimentais demonstram que o Omega oferece segurança e baixa latência.

1 Introdução

A computação na nuvem é um modelo para instanciar aplicações da Internet, e que permite às empresas executarem serviços em infraestruturas partilhadas, tipicamente grandes centros de dados geridos por fornecedores de serviços da nuvem. As economias de escala resultantes do uso de grandes infraestruturas partilhadas reduzem os custos de instanciar aplicações e permitem adaptar recursos associados a cada aplicação em resposta a mudanças na procura. Devido a estas vantagens, a computação na nuvem tem sido amplamente adotada.

Apesar destes benefícios, a computação na nuvem tem também as suas limitações. O número de centro de dados que oferecem serviços na nuvem é relativamente pequeno e estes estão tipicamente em localizações centrais, o que leva a alguns clientes observarem longas latências [2]. Isto inviabiliza a sua utilização para suportar aplicações como as de realidade aumentada ou certos jogos que exigem latências no acesso à nuvem entre 5ms e 30ms [14]. Uma solução para atender aos requisitos de latência destas aplicações é processar os dados perto dos terminais, um paradigma chamado computação na periferia da rede (do inglês, *edge computing*). Para suportar este paradigma, pode-se complementar os serviços prestados pelos centros de dados centralizados com o serviço de centros de dados menores, ou até mesmo servidores individuais, localizados

próximos da periferia. Este conceito é frequentemente chamado computação em neblina [3] (do inglês, *fog computing*) e pressupõe a disponibilidade de nós de neblina, também designados por *fog nodes* ou *cloudlets*, que traduzimos para cúmulos. Espera-se que o número de cúmulos seja várias ordens de magnitude maior do que o número de centro de dados na nuvem.

Os nós da nuvem encontram-se fisicamente em locais seguros e administrados por um único fornecedor. Os cúmulos, pelo contrário, são administrados por vários fornecedores locais e instalados em locais físicos que estão mais expostos. Portanto, os cúmulos são mais vulneráveis a serem comprometidos [12], levantando a segurança como uma das principais preocupações. Neste artigo, abordamos o problema de proteger o middleware que se executa na periferia da rede. Especificamente, concentramo-nos em proteger um *serviço de ordenação de eventos* que seja capaz de capturar as dependências de causa-efeito entre eventos. Um serviço de ordenação é um elemento fundamental para muitas aplicações distribuídas, como serviços de armazenamento [4], redes sociais, jogos, entre outros. A ideia de fornecer um serviço para ordenar eventos não é nova [5], mas pelo que sabemos, somos os primeiros a resolver o problema de fornecer uma concretização que pode ser executada com segurança nos cúmulos.

O nosso serviço, chamado *Omega*, aproveita a ampla disponibilidade de suporte para hardware confiável, nomeadamente de enclaves Intel SGX [11], para oferecer aos clientes da neblina garantias quanto à ordem pela qual os eventos são aplicados e servidos, mesmo quando os cúmulos ficam comprometidos. Um dos principais objetivos é proteger o serviço de ordenação sem violar as restrições de latência impostas pelas aplicações na periferia da rede. Conseguimos isto através do uso do enclave apenas para algumas operações importantes. Em particular, as aplicações são executadas fora do hardware confiável e usam o enclave seletivamente para solicitar provas sobre a ordem das operações.

Qualquer aplicação pode usar o serviço Omega. Para ilustrar o seu uso escolhemos desenvolver um sistema de armazenamento chave-valor chamado *OmegaCV*, que oferece coerência causal [9] na periferia. Este sistema serve para avaliar o desempenho do Omega. O OmegaCV é uma extensão de sistemas de armazenamento chave-valor com coerência causal que foram projetados anteriormente para a nuvem [4]. Como o OmegaCV usa o Omega para ordenar todas as suas operações, os clientes do OmegaCV podem executar operações de escrita e leitura em dados replicados por cúmulos, tendo a garantia de que as escritas são aplicadas e que as leituras são servidas numa ordem que respeita a causalidade.

Os nossos resultados experimentais mostram que o Omega introduz uma latência adicional de aproximadamente 4ms, o que permite servir os clientes com latências na gama dos 5ms-30ms, valores dentro dos tipicamente exigidos por aplicações sensíveis ao tempo.

2 Panorâmica e Trabalho Relacionado

Nesta secção, apresentamos uma panorâmica de serviços na periferia da rede e o trabalho relacionado.

Tabela 1: API do Omega. *Tag* é etiqueta.

void registerTag (EventTag tag)	*Registar uma etiqueta no Omega*
Event createEvent (EventId id, EventTag tag)	*Criar um evento com registo temporal dado um identificador e uma etiqueta*
Event orderEvents (Event e_1, Event e_2)	*Ordena dois eventos e retorna o primeiro*
Event lastEvent ()	*Retorna o último evento temporal do Omega*
Event lastEventWithTag (EventTag tag)	*Retorna o último evento temporal de uma dada etiqueta*
Event predecessorEvent (Event e)	*Retornar o predecessor imediato de um dado evento*
Event predecessorWithTag (Event e)	*Retorna o predecessor mais recente com a mesma etiqueta*
EventId getId (Event e)	*Retornar o identificador de um evento que faz parte do contexto da aplicação*
EventTag getTag (Event e)	*Retorna a etiqueta associada a um evento*

Protegendo serviços na neblina O facto de os cúmulos estarem dispersos entre múltiplas localizações geográficas, perto da periferia, aumenta o risco de serem atacados e se tornarem maliciosos [12]. Um cúmulo comprometido pode excluir, copiar ou alterar operações, fazendo com que informação seja perdida, divulgada ou alterada de maneira a corromper o funcionamento da aplicação. Para enfrentar este desafio, é preciso recorrer a uma combinação de técnicas, das quais destacamos a *replicação* e o *robustecimento* (do Inglês, *hardening*). A replicação consiste em usar vários cúmulos em vez de apenas um, usando técnicas como quóruns bizantinos [10,2]. No entanto, contactar múltiplos cúmulos pode aumentar a latência, algo a ser evitado. O robustecimento consiste em usar mecanismos de software e/ou hardware para reduzir a capacidade de um adversário comprometer um dispositivo. Um mecanismo relevante neste contexto é o uso de hardware confiável que oferece integridade e confidencialidade até quando o sistema operativo está comprometido.

Intel SGX As *Intel Software Guard Extensions* (SGX), introduzidas nos microprocessadores Intel Core de sexta geração, implementam uma forma de modo de execução confiável que se designa por *enclave*. As aplicações desenhadas para usar SGX têm duas partes: uma parte não confiável e uma parte confiável. A parte confiável é executada dentro do enclave, onde o código e os dados têm integridade e confidencialidade; a parte não confiável é executada como uma aplicação normal. A parte não confiável pode fazer uma chamada ao enclave (ECALL) para alternar para o enclave e iniciar a execução confiável. A arquitetura SGX oferece vários mecanismos para garantir a integridade do código, incluindo um procedimento de *atestação* que permite um cliente obter uma prova de que está a comunicar com um código específico num enclave SGX real, e não com um impostor [1]. Uma limitação das concretizações atuais do SGX é que a região de memória protegida está limitada a 128 MB.

Ordenação de eventos A maioria das aplicações distribuídas precisa capturar a ordem de eventos. Na maioria dos casos, o serviço de ordenação de eventos é um componente crítico da aplicação e, se este for comprometido, a correta execução da aplicação não poderá mais ser garantida. Tipicamente, a lógica do serviço de ordenação está entrelaçada com a lógica da aplicação, aumentando a

complexidade do código e a dificuldade de capturar a dependência de eventos entre várias aplicações. Uma abordagem alternativa recentemente proposta é o Kronos [5] que consiste em oferecer um serviço de ordenação de eventos que pode ser usado por várias aplicações. Neste artigo, seguimos este caminho e descrevemos o desenho e a concretização do Omega, um serviço seguro (ao contrário do Kronos) de ordenação de eventos a ser executado em cúmulos.

Armazenamento na periferia da rede Para aproveitar todo o seu potencial, os cúmulos não devem apenas fornecer processamento, mas também armazenar dados que podem ser usados com frequência; caso contrário, as vantagens do processamento na periferia podem ser prejudicadas por frequentes acessos a dados remotos. Consequentemente, neste documento, também descrevemos a concretização de um serviço de armazenamento a ser fornecido por cúmulos, que denominámos OmegaCV. O OmegaCV estende os serviços de armazenamento de chave-valor desenhados para a nuvem que oferecem *coerência causal* [4].

ShieldStore e Pesos O ShieldStore [7] é um recente serviço de armazenamento chave-valor baseado no SGX que foi desenhado para operar em centros de dados na camada da nuvem. Omega é um serviço mais geral, que pode ser usado para concretizar um serviço de armazenamento chave-valor, mas também outros serviços na camada da neblina. O Pesos [8] é um serviço seguro de armazenamento de objetos que também aproveita o SGX. O Pesos também foi criado para a nuvem e assume a existência de uma entidade externa segura para armazenar persistentemente os dados, enquanto que o OmegaCV armazena os dados localmente na parte não confiável, superando assim a limitação a 128 MB.

3 O Serviço Omega

O Omega é um serviço de ordenação de eventos seguro que é executado num cúmulo e que atribui carimbos temporais lógicos a eventos. Estes carimbos não podem ser adulterados, mesmo que o cúmulo tenha sido comprometido. Os clientes podem solicitar ao Omega que atribua carimbos temporais lógicos aos eventos que eles produzem e podem usar estes carimbos para extrair informação sobre possíveis relações de causa e efeito entre eventos. Na implementação atual do Omega, os carimbos são apenas um inteiro que é incrementando a cada evento. Além disto, o Omega guarda os últimos eventos que foram registados no sistema e também guarda o predecessor de cada evento. Estas últimas características são relevantes, pois permitem que um cliente verifique se as informações fornecidas por um cúmulo são recentes e completas (ou seja, se um cúmulo comprometido omitir alguns eventos no passado causal de um cliente, o cliente pode sinalizar o cúmulo como malicioso). Mais precisamente, Omega estabelece uma linearização [6] de todos os pedidos que recebe, definindo uma ordem total coerente com a causalidade para todos os eventos que ocorrem no cúmulo.

3.1 API do Omega

A interface do serviço Omega está representada na Tabela 1. O Omega atribui, a pedido de clientes, carimbos temporais lógicos a eventos (*Event*) de uma

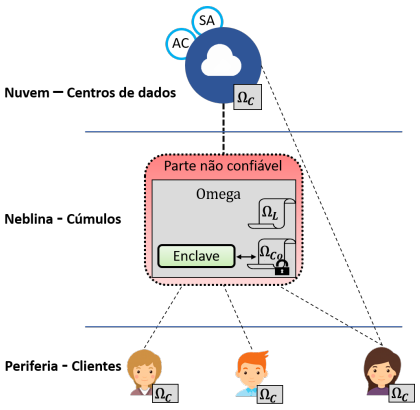

Figura 1: Arquitetura do Omega. AC é autoridade de certificação, SA é servidor de atestação, Ω_C é o cliente, Ω_{Co} é o cofre e Ω_L é o histórico de eventos.

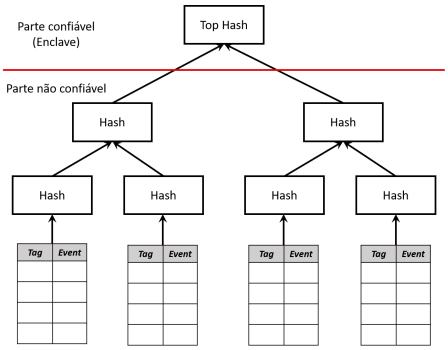

Figura 2: Árvore de Merkle armazenada no cofre Omega na parte não confiável do cúmulo (com $N = 4$).

aplicação. Cada evento é considerado como tendo um identificador único que é atribuído pelo cliente. O Omega também permite que a aplicação associe uma determinada etiqueta (*tag*) a cada evento. A aplicação pode usar as etiquetas livremente (podem ser associadas a utilizadores, a chaves num armazenamento chave-valor, a fontes de eventos, etc.), desde que sejam registadas antes de serem usadas (registerTag). A operação createEvent atribui um carimbo temporal a um evento do cliente e retorna um objeto do tipo Event que tem vinculado de uma forma segura o carimbo temporal e uma etiqueta.

Os clientes não precisam de conhecer o formato interno usado pelo Omega para codificar os carimbos temporais lógicos, que são encapsulados num objeto do tipo Event. Em vez disto, o cliente pode usar as restantes primitivas do Omega para consultar a ordem dos eventos e explorar a linearização de eventos que foi definida pelo Omega. A primitiva orderEvents recebe dois eventos e retorna o mais antigo de acordo com a ordem de linearização. O cliente também pode pedir ao Omega o último evento que foi marcado temporalmente (lastEvent), ou o evento mais recente associado a uma determinada etiqueta (lastEventWithTag). Dado um evento alvo, o cliente também pode obter o evento que é o predecessor imediato deste alvo na ordem de linearização (predecessorEvent) ou o predecessor mais recente que partilha a mesma etiqueta com o alvo (predecessorWithTag). Por fim, getId e getTag extraem o identificador único e a etiqueta do evento que fazem parte do contexto da aplicação e que foram vinculados com segurança ao carimbo temporal.

Note-se que, embora o Omega seja inspirado em serviços como o Kronos [5], oferece uma interface que faz compromissos distintos. Primeiro, permite que os clientes associem etiquetas específicas a eventos, podendo depois adquirir todos os eventos anteriores com essas etiquetas; no Kronos os clientes têm que percorrer

toda a história de eventos para obter a versão anterior de um objeto. Em segundo lugar, no Kronos a aplicação tem que declarar explicitamente as relações causais entre objetos; isto é mais versátil mas mais complexo de usar do que o Omega, que define automaticamente uma dependência causal entre a última operação de um cliente e todas as operações que este cliente executou ou observou no seu passado. Finalmente, ao contrário do Kronos, o Omega estabelece automaticamente uma linearização de todas as operações, o que simplifica o desenho de aplicações que precisem de ordenar totalmente operações concorrentes de forma coerente com a causalidade.

4 Concretização do Omega

Nesta secção, descrevemos o desenho e a concretização do serviço Omega.

4.1 Arquitetura do Sistema e Interações

O serviço Omega é executado em cúmulos e é usado por processos em execução na periferia da rede ou em centros de dados na nuvem (Figura 1). Os dispositivos na periferia e a nuvem podem usar o Omega para criar e ler eventos no cúmulo de maneira segura. Por exemplo, os dispositivos da periferia podem fazer atualizações nos dados armazenados no cúmulo que são posteriormente enviados para a nuvem (neste caso, os dispositivos da periferia criam eventos e a nuvem lê-os). Além disto, a nuvem pode receber atualizações de outros locais e atualizar o conteúdo no cúmulo com novos dados que são subsequentemente lidos pelos dispositivos da periferia. Para o Omega não é preciso distinguir os processos em execução na periferia dos processos em execução na nuvem, que simplesmente denotamos como *clientes*. O método usado pelos clientes para obter o endereço dos cúmulos é ortogonal a este artigo. Um cliente pode descobrir os cúmulos próximos de si fazendo um pedido ao Domain Name System (DNS) ou à nuvem.

O Omega pressupõe a existência de dois componentes externos, que são executados na nuvem e que são considerados seguros. Estes componentes são uma *Autoridade de Certificação* (AC) usada para gerar certificados de chave pública e um *Servidor de Atestação* (SA), que é usado quando um cúmulo se liga ao Omega por meio de um procedimento de ligação (Secção 4.3).

Um aspeto importante do Omega é manter a funcionalidade do sistema no caso de um cúmulo ser comprometido. Para resolver este problema, o Omega usa a tecnologia Intel SGX, como se ilustra na Figura 1, gerando todos os eventos dentro do enclave (operação `createEvent`). Além disto, todos os eventos possuem uma assinatura digital obtida dentro do enclave usando a chave privada do cúmulo, também armazenada dentro do enclave. De seguida apresentamos os pressupostos de segurança do nosso sistema (Secção 4.2); depois o protocolo usado pelos clientes para garantir que estão a interagir com a concretização correta do Omega (Secção 4.3); e por fim dois subcomponentes chamados *cofre* e *histórico de eventos* que são usados para preservar o estado do Omega (Secção 4.4).

4.2 Pressupostos de Segurança

Devido à sua localização exposta, os cúmulos podem sofrer diversos ataques e ficar comprometidos (um atacante pode inclusive obter acesso físico a um cúmulo). Assumimos assim que os cúmulos podem falhar de forma arbitrária. Por exemplo, um cúmulo malicioso pode: modificar a ordem das mensagens no sistema; modificar o conteúdo das mensagens; repetir mensagens (*replay attack*); adulterar os dados armazenados; e gerar eventos incorretos. Todas estas ações, se não forem tratadas com cuidado, podem levar o sistema a um estado de falha. Por outro lado, assumimos que os clientes e os serviços na nuvem (SA, AC) são confiáveis, isto é, supõe-se que falham apenas por paragem de execução (os mesmos pressupostos que o trabalho relacionado [5,4]).

Também assumimos que cada cúmulo possui um processador com SGX (Figura 1). Os clientes e os cúmulos possuem pares de chaves assimétricas. A chave privada do cúmulo nunca sai do enclave. Para a distribuição de chaves públicas, consideramos a existência de uma Infraestrutura da Chave Pública (PKI). Fazemos as suposições habituais sobre a segurança do hardware confiável/enclaves (dados executados/armazenados dentro do enclave têm integridade e confidencialidade garantidas) e sobre os mecanismos criptográficos (por exemplo, chaves privadas não são divulgadas, assinaturas não podem ser criadas sem a chave privada e a função de hash é resistente a colisões). Para as assinaturas digitais usamos o algoritmo ECDSA com chaves de 256 bits e para a função hash o SHA-256. Usamos as implementações do SGX SDK (dentro do enclave) e do Java (externo).

4.3 Vinculação do Cliente

Antes de um cliente invocar qualquer método da API Omega, deve executar um procedimento de *vinculação*. O objetivo deste procedimento é garantir que o cliente tem uma ligação segura com um componente de software que está a ser executado num enclave real. Este procedimento, também é conhecido como atestação. Uma limitação do procedimento de atestação definido pela Intel é que envolve várias etapas de comunicação, aumentando a latência. Por isto recorremos a uma estratégia diferente. O nosso algoritmo requer que o enclave gere um par de chaves fresco (ortogonal às chaves do SGX), que depois é atestado pelo nosso Servidor de Atestação (SA) a ser executado na nuvem; se tiver sucesso gera um certificado para esta chave pública do enclave, que fica depois guardado na parte não confiável do cúmulo. Um cliente em vez de executar a atestação da Intel apenas solicita ao Omega o certificado que foi emitido pelo SA.

4.4 O Cofre Omega e o Histórico de Eventos

O Omega necessita armazenar com segurança diferentes dados, como a chave privada associada ao certificado assinado pelo SA, o último evento gerado pelo Omega e o último evento associado a cada etiqueta. No entanto, a memória do enclave é limitada a algumas dezenas de megabytes e o Omega deve manter um

número arbitrário de etiquetas. Portanto, é necessária uma maneira de armazenar com segurança esses dados (em particular, o último evento para um número arbitrário de etiquetas, para garantir frescura). Além disto, o Omega deve ter acesso a eventos que gerou no passado, uma vez que os clientes podem usar o método predecessorEvent para percorrer o histórico de eventos. Para atender a estes requisitos, o Omega usa dois serviços de armazenamento com propriedades diferentes, o cofre e o histórico de eventos. Nos dois casos, o Omega armazena eventos na zona não confiável. Estes eventos podem estar visíveis para a parte não confiável, mas é necessário garantir a sua integridade, ou seja, que a zona não confiável é incapaz de modificar estes valores caso o cúmulo seja comprometido. Dado que os eventos são assinados pelo Omega, a zona não confiável não pode modificar eventos individuais; no entanto, ela pode excluir eventos ou substituir novos eventos por eventos mais antigos. A concretização do histórico e do cofre, que descrevemos de seguida, permitem detectar este tipo de ações.

O *histórico de eventos* é apenas um registo de todos os eventos gerados. Optámos por concretizar este componente como um armazenamento chave-valor no qual os eventos são armazenados usando o seu identificador único (atribuído pela aplicação) como chave. Sempre que o Omega faz uma pesquisa por um evento específico (por exemplo, quando um cliente percorre o histórico de eventos), ele simplesmente verifica a integridade do evento do valor ser retornado ao cliente. Se um evento não puder ser encontrado no armazenamento chave-valor, é sinal de que os componentes não confiáveis do cúmulo foram comprometidos.

O *cofre* é mais difícil de implementar porque precisa manter o último evento gerado para cada etiqueta e tem de garantir que os componentes não confiáveis não podem substituir o último evento por um evento mais antigo. Portanto, verificar a integridade do evento retornado não é suficiente: a concretização do cofre Omega deve garantir que os valores não foram alterados. Logicamente, isto é obtido se o enclave fizer uma hash do cofre sempre que atualizar seu conteúdo e a hash for armazenada no próprio enclave. No entanto, esta concretização ingénua não teria um bom desempenho pois a aplicação pode usar um grande número de etiquetas e calcular uma hash de todas estas etiquetas pode levar muito tempo. Além disto, quando os valores que são usados no cálculo da hash estão armazenados fora do enclave, há o risco de poderem ser alterados pelo adversário durante a execução da função de hash, levando a que esta obtenha um valor errado.

Para resolver os problemas acima, a concretização do cofre usa as seguintes técnicas. Primeiro, o conteúdo do cofre é armazenado como uma *árvore de Merkle*. Embora conceptualmente o cofre seja apenas uma tabela, mantida na zona não confiável, onde cada linha é uma etiqueta (índice) e uma coluna para o evento (ver Figura 2), na concretização, esta tabela é dividida em N partes, e para cada parte, o enclave calcula um hash para garantir a integridade. Como o enclave pode não ter memória suficiente para armazenar todos estas hashes, usamos uma árvore de Merkle de forma que o enclave só precise armazenar a hash de topo. Todas as hashes são calculadas dentro do enclave. Para o efeito, usamos o atributo *user_check* do SGX, que permite passar um ponteiro do espaço

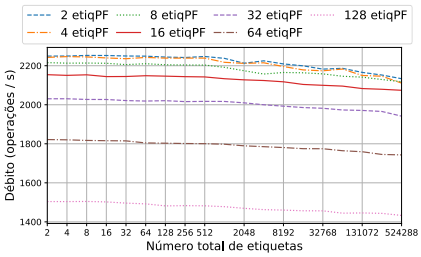

Figura 3: Débito variando a dimensão do cofre e da folha, etiqPF é etiquetas por folha.

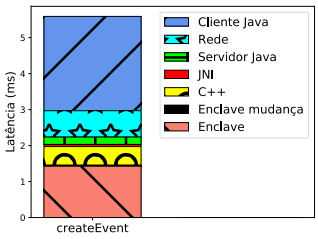

Figura 4: Latência da operação createEvent.

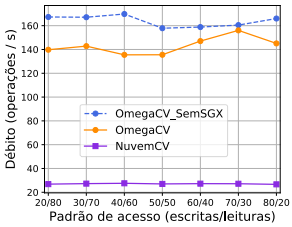

Figura 5: Débito.

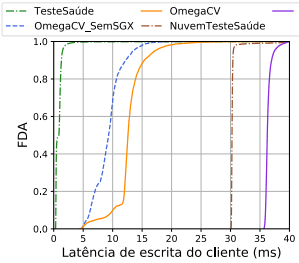

Figura 6: Latência da escrita (cúmulo e nuvem).

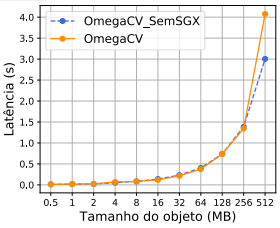

Figura 7: Latência da escrita com e sem SGX.

de memória de zona não confiável como um argumento numa ECALL, para que o enclave possa aceder a dados que se encontram na parte não confiável. Desta forma, o enclave pode verificar e gerar as hashes da árvore quando necessário na zona não confiável, necessitando apenas de armazenar a hash superior.

Quando o enclave modifica uma parte da tabela, ele tem de: computar a árvore para verificar os dados, alterar os dados e, finalmente, recalcular algumas hashes da árvore (tantas quanto a profundidade da árvore). Estas operações devem ser realizadas de maneira atómica, caso contrário, um atacante pode alterar a tabela entre os dois cálculos da árvore e o enclave não conseguirá detetar. Para garantir a atomicidade, o enclave calcula as hashes em paralelo, isto é, calcula a hash antiga e a nova hash da tabela em simultâneo, para que no final possa simplesmente substituir a antiga.

5 Avaliação

Nesta secção discutimos a correta configuração do Omega, apresentamos o seu desempenho quando usado isoladamente, e avaliamos o impacto no uso do Omega para proteger um caso de uso concreto: um serviço de armazenamento de chave-valor, o OmegaCV. Na bancada de testes da nossa avaliação usamos como cúmulo um computador com um CPU Intel i9-9900K (com SGX) e 16 GB de RAM. A

máquina dos clientes tem um CPU i7-4710HQ e 16 GB de RAM. Tanto os clientes quanto o cúmulo estão localizados no nosso laboratório, na mesma rede, emulando uma estação 5G (ou seja, uma comunicação de 1-hop). Os serviços da nuvem são executados num centro de dados em Londres da Amazon EC2 (máquinas virtuais t2.micro). O SDK do Intel SGX e o código para o enclave foram escritos em C/C++. O Omega foi desenvolvido em Java 11 e a Java Native Interface (JNI) foi usada como interface entre o código Java e C++. Para armazenamento persistente usamos o sistema Redis[13].

5.1 Configuração do Omega

Primeiro discutimos como configurar a árvore de Merkle usada pelo Omega; em seguida, fornecemos uma visão geral do desempenho do Omega usando a configuração selecionada para a árvore de Merkle. A árvore de Merkle é usada nas principais operações do Omega, logo a sua correta configuração é fundamental para o desempenho do serviço. Para configurar a árvore, é de notar que qualquer operação que envolva a verificação/alteração do conteúdo do cofre requer a realização de um número de cálculos que é uma função do tamanho do cofre, mas também do tamanho selecionado para as folhas da árvore de Merkle. Qualquer operação no cofre implica calcular a hash do nó folha em causa e, em seguida, a hash de todos os nós internos da árvore. A computação da hash do nó da folha tem um custo linear com o tamanho da folha; como implementamos a árvore de Merkle como uma árvore binária, atualizar/verificar um nó interno envolve fazer a hash de dois valores e o número de nós internos que precisam ser calculados cresce logaritmicamente com o tamanho do cofre.

Com base nisto decidimos executar várias experiências no sistema, nas quais medimos o desempenho do cofre Omega com diferentes tamanhos de folha e diferentes tamanhos de cofre. Os resultados estão na Figura 3. Observamos que o tamanho ótimo para os nós da folha da árvore deve ser muito próximo de 1, dado que o custo de calcular a hash do nó da folha cresce linearmente, enquanto o custo de calcular as hash dos nós internos cresce logaritmicamente. Como pode ser observado, os melhores resultados são obtidos para tamanhos de folha de 2 e 4 (na verdade, as diferenças de desempenho para estes dois valores não são significativas), mas cai rapidamente se folhas maiores forem usadas. Portanto, em todos as outras experiências, usamos um tamanho de folha de 2.

5.2 Desempenho do Omega

Avaliamos agora o desempenho do Omega quando usado isoladamente. Para esta experiência, medimos a latência observada por um cliente ao executar o pedido mais complexo da API, a operação `createEvent`, que modifica o estado do cofre. É de notar que todas as mensagens contêm assinaturas digitais. Medimos o atraso com que cada componente de software contribui no caminho crítico do cliente. Os resultados estão na Figura 4. Como o cúmulo está localizado a 1-hop de distância dos clientes, o tempo gasto na rede é insignificante para a latência observada pelos clientes. O tempo perdido da camada Java para o enclave também

é pequeno (1-2ms). O tempo perdido a fazer a troca de contexto também é pequeno porque o enclave mantém um estado pequeno (aproveitando o cofre) e há um pequeno número de parâmetros a entrar e sair do enclave. Assim, a maior penalização na latência são as funções criptográficas executadas no cliente e no enclave. No cliente, são necessários 2-2,5ms para calcular e verificar assinaturas digitais. No lado do servidor, a maior parte do tempo é também gasto no processo de computação e verificação de assinaturas digitais.

5.3 Desempenho do OmegaCV

Agora medimos o impacto do uso do Omega para tornar outros serviços seguros. Para este propósito, desenvolvemos o OmegaCV, que usa o Omega para ordenar as leituras e as escritas. A chave de um objeto guardado no Redis corresponde a uma etiqueta no Omega, de modo a garantir que os clientes lêem sempre o valor mais recente de cada objeto. Comparamos o desempenho do OmegaCV com um serviço não seguro semelhante também executado no cúmulo (denotado OmegaCV_SemSGX) e com uma versão em que a segurança é alcançada executando o serviço na nuvem (NuvemCV). Todas as concretizações dos sistema de armazenamento foram desenvolvidas em Java e usam o Redis para manter o seu estado persistente. Além disto, todas as mensagens dos sistemas possuem um *nonce* e uma assinatura digital. A principal diferença entre os sistemas é que o NuvemCV e o OmegaCV_SemSGX não usam o enclave (nem a árvore de Merkle do cofre), não fazem nenhum esforço para verificar a integridade dos dados e não usam o JNI entre a camada Java e C++.

A Figura 5 apresenta o débito máximo que um cliente pode alcançar usando os três sistemas. Na concretização do NuvemCV, a latência do centro de dados afeta severamente o débito do cliente. Curiosamente, embora os mecanismos de segurança usados no Omega tenham introduzido alguma latência, esta é parcialmente diluída quando Omega é apenas uma parte de um sistema maior, que tem muitas outras fontes de latência. Nas experiências, o OmegaCV oferece um débito que é aproximadamente 18% menor que a versão não segura do mesmo serviço, mas que é, no entanto, muito superior ao débito suportado pelo NuvemCV. A Figura 6 compara a latência que um cliente observa ao usar os serviços OmegaCV, OmegaCV_SemSGX e NuvemCV. As linhas TesteSaúde para o cúmulo e NuvemTesteSaúde para a nuvem mostram o tempo de ida-e-volta na rede. Como esperado, o cliente pode executar operações com latência muito menor usando o cúmulo em vez de usar os serviços NuvemCV que estão num centro de dados, uma redução de 36ms para 12ms, perto de 67%. O OmegaCV tem maior latência que o OmegaCV_SemSGX, devido ao uso do enclave. Em valor absoluto, observamos um aumento na latência na ordem de 4ms que, não sendo desprezável, é significativamente inferior à latência introduzida pelas ligações à nuvem. Isto permite que o OmegaCV ofereça valores de latência no intervalo de 5ms a 30ms exigido por aplicações na periferia sensíveis à latência [14]. Finalmente, a Figura 7 mostra o efeito do tamanho dos dados no desempenho. É visível que nosso sistema segue a mesma latência que o armazenamento tradicional de chave-valor. Isto porque, com ficheiros grandes, o peso do enclave e das

operações criptográficas tornam-se insignificantes comparando com os custos de transferência dos dados.

6 Conclusões

A computação em neblina pode abrir o caminho para o desenvolvimento de novas aplicações sensíveis à latência na periferia, como a realidade aumentada, mas obriga a utilizar serviços robustos às vulnerabilidades que resultam da utilização de cúmulos. Este artigo dá um passo nessa direção, descrevendo a arquitetura e concretização do Omega, um serviço de ordenação que pode ser executado em cúmulos de uma maneira segura, aproveitando as propriedades de hardware confiável, como o Intel SGX. Para o cenário estudado, as aplicações baseadas no Omega podem fornecer uma latência muito menor e débito superior às soluções atuais baseadas em nuvem, apesar dos custos incorridos com o enclave.

Agradecimentos: Este trabalho foi suportado pela FCT – Fundação para a Ciência e a Tecnologia, através dos projectos UID/CEC/50021/2019 e COSMOS (financiado pelo OE com a ref. PTDC/EEI-COM/29271/2017 e pelo Programa Operacional Regional de Lisboa na sua componente FEDER com a ref. Lisboa-01-0145-FEDER-029271).

Referências

1. Barbosa, M., Portela, B., Scerri, G., Warinschi, B.: Foundations of hardware-based attested computation and application to SGX. In: Proc. of the 1st IEEE EuroSP. Saarbrücken, Germany (Mar 2016)
2. Bessani, A.N., Correia, M., Quaresma, B., André, F., Sousa, P.: Depsky: Dependable and secure storage in a cloud-of-clouds. ACM TOS **9**(4) (2013)
3. Bonomi, F., Milito, R., Zhu, J., Addepalli, S.: Fog computing and its role in the internet of things. In: Proc. of the 1st ACM MCC. Helsinki, Finland (Aug 2012)
4. Bravo, M., Rodrigues, L., Van Roy, P.: Saturn: A distributed metadata service for causal consistency. In: Proc. of the 12th EuroSys. Belgrade, Serbia (Apr 2017)
5. Escriva, R., et. al: Kronos: The design and implementation of an event ordering service. In: Proc. of the 9th EuroSys. Amsterdam, The Netherlands (Apr 2014)
6. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM TOPLAS **12**(3) (1990)
7. Kim, T., Park, J., Woo, J., Jeon, S., Huh, J.: Shieldstore: Shielded in-memory key-value storage with SGX. In: Proc. of the 14th EuroSys. Porto, Portugal (2019)
8. Krahn, R., et. al: Pesos: policy enhanced secure object store. In: Proc. of the 13th EuroSys. Belgrade, Serbia (2018)
9. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7) (Jul 1978)
10. Malkhi, D., Reiter, M.: Byzantine quorum systems. In: Proc. of the 29th ACM STOC. El Paso, TX, USA (1997)
11. McKeen, F., et. al: Innovative instructions and software model for isolated execution. In: Proc. of the HASP. Tel-Aviv, Israel (Jun 2013)
12. Mukherjee, M., et. al: Security and privacy in fog computing: Challenges. IEEE Access **5**(6) (Sep 2017)
13. Redis: `http://redis.io`, accessed: 2019-05-03
14. Ricart, G.: A city edge cloud with its economic and technical considerations. In: Proc. of the IEEE PerCom Workshops. Kona (HI), USA (2017)