Capítulo 3

Ambientes de Computação Segura

Romeo Bulla Jr. (USP), Nelson Yamamoto (USP), Marcos Antonio Simplicio Jr. (USP), Julião Braga (USP/UFABC), Stephan Kovach (USP), Wilson Vicente Ruggiero (USP)

Abstract

This paper discusses Trusted Execution Environments (TEEs), a technology that provides confidentiality and integrity of data during its processing. We present the main characteristics of TEEs, such as isolation and attestation, and explore the attacks that can be carried out. Then, we detail Intel SGX, an instruction set that offers isolation through enclaves. We also describe the main development tools of Intel SGX, such as Intel SGX SDK and Occlum, and we illustrate these tools through experiments, testing TEEs even against high-privileged users. Finally, we present a prototype showing the application of TEEs in a streaming platform authentication.

Resumo

Este trabalho aborda os Ambientes de Execução Confiáveis (TEEs), uma tecnologia que fornece confidencialidade e integridade dos dados durante seu processamento. O estudo apresenta as principais características dos TEEs, como isolamento e atestação, e explora os ataques que podem ser realizados contra eles. Em seguida, o trabalho detalha o Intel SGX, um conjunto de instruções que oferece isolamento através de enclaves. São também descritas as principais ferramentas de desenvolvimento, como Intel SGX SDK e Occlum, e seu uso é ilustrado através de experimentos práticos, testando TEEs mesmo contra usuários com altos privilégios. Por fim, é apresentado um protótipo que mostra a aplicação de TEEs na autenticação de uma plataforma de streaming.

3.1. Introdução

Nos dias de hoje, a segurança de dados e a privacidade são preocupações essenciais em um mundo cada vez mais interconectado [Kanno 2005]. A explosão do uso de dispositivos móveis, serviços online e a crescente utilização de serviços em nuvem tornaram a proteção de informações pessoais e a integridade das funções críticas mais importantes

do que nunca. Afinal, em um ambiente onde grandes volumes de dados são comumente armazenados em ambientes de terceiros, corre-se o risco de que agentes maliciosos com acesso a esses ambientes extraiam informações estratégicas ou sensíveis, ou subvertam a operação de aplicações que fazem uso desses dados. Esse cenário leva à necessidade de mecanismos capaz de prover o que é comumente chamado na literatura de "computação confiável" (*trusted computing*), termo guarda-chuva que engloba diversas tecnologias capazes de prover serviços de confidencialidade e integridade para processos computacionais [Li et al. 2023].

Dentre as tecnologias de computação confiável existentes, e que desempenham um papel relevante na garantia de segurança em sistemas computacionais modernos, este minicurso se interessa particularmente por uma: o *Trusted Execution Environment* (TEE), ou Ambiente de Execução Confiável em uma tradução livre. Essencialmente, esse mecanismo tem por objetivo fornecer um ambiente isolado e de alta confiança para execução de aplicações, em particular aquelas que exigem confidencialidade e integridade em relação a outros componentes do sistema. Para isso, os componentes de hardware e software na CPU isolam as cargas de trabalho em execução no TEE, comumente chamadas de Aplicações Confiáveis (*Trusted Applications* - TAs), do sistema operacional principal (SO) [Paju et al. 2023]. Caso haja necessidade de interação com componentes externos (e.g., armazenamento de dados em memória RAM, ou sua transmissão pela rede), essa comunicação é protegida por algoritmos de criptografia. Consequentemente, mesmo na hipótese de um sistema operacional comprometido ou de um administrador de sistema malicioso, ainda é possível executar as aplicações confiáveis da forma esperada, ao mesmo tempo em que se preserva o sigilo dos dados por elas processados.

Diversos microprocessadores modernos têm suporte a tecnologias de TEE. Isso significa que esses dispositivos possuem internamente ao seu hardware uma área isolada e protegida contra ataques físicos e lógicos [Muñoz et al. 2023]. Como resultado, é possível executar funções críticas de segurança e proteger os dados correspondentes, separando-os do ambiente potencialmente inseguro em seu entorno [Bursell 2021]. Essa estratégia permite, assim, reforçar conceitos modernos de segurança computacional, como confiança-zero (*zero trust*) [Rose et al. 2020, Syed et al. 2022].

Uma das muitas aplicações de características de TEEs está na Internet das Coisas (IoT). Dispositivos IoT, que possuem baixo poder de processamento, geralmente precisam enviar os dados coletados para um servidor centralizado na nuvem. Nesse contexto, os TEEs podem reforçar a confidencialidade e a integridade durante o processamento desses dados [Gremaud et al. 2017]. Da mesma forma, áreas como a Inteligência Artificial (IA) e a mineração de dados sensíveis também podem se beneficiar das garantias oferecidas pelos TEEs [Kaissis et al. 2020, Geppert et al. 2022].

Assim, esse trabalho apresenta as principais características dos TEEs, como isolamento e atestação, e explora algumas de suas implementações, com o foco em Intel SGX. Em seguida, são abordadas as principais ferramentas para o desenvolvimento de aplicações utilizando TEEs, com exemplos práticos que ilustram seu uso. Por fim, o trabalho discute um caso de uso específico, apresentando um protótipo que realiza a autenticação de uma plataforma de streaming utilizando o Intel SGX.

Este texto é organizado da seguinte maneira: a Seção 3.2 apresenta uma discussão

teórica sobre alguns dos termos utilizados (enclaves e atestação) e as principais funcionalidades de segurança, e a Seção 3.3 detalha o Intel SGX, a solução TEE com maior foco no minicurso. Em seguida, a Seção 3.4 descreve brevemente outras soluções de TEE (e.g., ARM TrustZone e ARM SEV), a Seção 3.5 discute algumas das principais ferramentas de desenvolvimento e programação de TEEs, e as Seções 3.6 e 3.7 descreve alguns experimentos. Por fim, a Seção 3.8 apresenta um protótipo com um caso de uso, e a Seção 3.9 traz uma conclusão.

3.2. Fundamentação teórica

De certa forma, um TEE pode ser visto como uma extensão do conceito de módulo de plataforma confiável (*trusted platform module* - TPM [Aaraj et al. 2009, Arthur and Challener 2015]), adicionando flexibilidade ao seu uso. Enquanto TPMs normalmente vêm configurados com um conjunto de funcionalidades bem definidas, sejam elas programadas pelo fabricante ou pelos usuários, acessadas por meio de uma interface de programação de aplicações (*application programming interface* - API), um TEE permite a execução de código arbitrário de forma dinâmica, sem que ele precise ser pré-programado no hardware [Sabt et al. 2015, Arfaoui et al. 2014].

A fim de isolar o ambiente de execução, os componentes de hardware e software na CPU isolam as cargas de trabalho em execução no TEE, comumente chamadas de Aplicações Confiáveis (*Trusted Applications* - TAs), do sistema operacional principal (SO) [Paju et al. 2023]. Caso haja necessidade de interação com componentes externos (e.g., armazenamento de dados em memória RAM, ou sua transmissão pela rede), essa comunicação é protegida por algoritmos de criptografia. Assim, mesmo na hipótese de um sistema operacional comprometido ou de um administrador de sistema malicioso, ainda é possível executar as aplicações confiáveis da forma esperada preservando o sigilo dos dados por elas processados.

Com o intuito de apresentar os conceitos de forma clara, bem como consolidar as bases teóricas para a parte prática do minicurso, esta Seção visa apresentar a fundamentação acerca do que engloba os TEEs. Assim, ela divide-se em duas partes principais: a Seção 3.2.1 preocupa-se com a apresentação dos conceitos básicos e funcionalidades de segurança fornecidas pelos TEEs, e a Seção 3.2.2 discute os principais mecanismos de segurança dos TEEs.

3.2.1. Conceitos Básicos

Enclave é um conceito típico do Intel SGX e corresponde a uma área de memória protegida dentro de um processo que é isolada do resto do sistema operacional por meio da cifragem das páginas de memória e consequente decifração somente durante a execução de modo a prover confidencialidade e integridade impedindo que outros processos com privilégios elevados consigam alterar a memória associada à carga de trabalho em execução [Bursell 2020]. Durante a execução, dados e códigos são colocados em uma região especial de memória do processador chamada *Enclave Page Cache* – EPC – a fim de garantir o isolamento de outros programas.

Atestação estabelece a relação de confiança entre dois enclaves na mesma plataforma (local) ou um enclave e uma outra máquina (remota) por meio da criação de um

canal de comunicação seguro (e.g., acordo de chaves Diffie-Hellman) pelo qual os dados serão transmitidos ao enclave [Anati et al. 2013].

3.2.2. Mecanismos de Segurança

A gama de mecanismos oferecidos pelos Ambientes de Computação Segura é extremamente vasta a fim de cumprir os requisitos de confidencialidade e integridade [Li et al. 2023]. Dentre elas destacam-se as seguintes:

i) Boot seguro: refere-se ao processo no qual, antes do SO inicializar, garante-se a carga correta do TEE e a impossibilidade de adulteração por nenhum host. Desse modo, logo que a máquina inicializa, é feita a carga de imagens imutáveis e verificadas a fim de estabelecer uma cadeia de confiança entre as imagens do enclave, os componentes do SO e as configurações;

ii) Execução isolada: durante a execução, tanto códigos quanto dados, são mantidos na EPC de modo que processos externos não conseguem acessar nenhuma das informações;

iii) Selamento: refere-se à recuperação (e armazenamento) de dados secretos do (no) disco local. Esse mecanismo vale-se da decifração (cifração, respectivamente) dos dados por meio de uma "chave de selamento" que permite a execução do mecanismo de selamento.

3.2.3. Superfícies de Ataques

Naturalmente, os ambientes de computação segura tem como objetivo criar ambientes íntegros e confidenciais de execução. Contudo, do mesmo modo que diversas tecnologias, métodos e algoritmos que buscam proteger conteúdos sensíveis, os TEEs também possuem pontos de vulnerabilidade. Algumas vezes por implementação falha, outras vezes pela concepção falha.

Nesse sentido, se faz necessária a compreensão de algumas superfícies de ataques aplicáveis aos ambientes de computação segura. Elas são os importantes pontos de partida para exploração de vulnerabilidades.

A seguir, há um resumo das principais superfícies de ataques a TEEs. Elas estão classificadas de acordo como proposto pelo trabalho de [Fei et al. 2021], mas complementadas pelo trabalho de [Muñoz et al. 2023].

- *Vulnerabilidades em Tradução de Endereços (ou address translation vulnerabilities*: Nesses tipos de ataque, o atacante se utiliza de acesso privilegiado a máquina vítima. Por vezes, o ele já conhece os binários e o código-fonte. Então o atacante deduz informações sensíveis analisando os perfis de acesso às páginas de memória;

- *Ataques Baseados em cache (ou cache timing attacks)*: Nesses tipos de ataques, o atacante se apoia na monitoração do uso de memória cache. Por vezes, o atacante primeiramente preenche a memória *cache* com valores conhecidos, permite que o programa vítima seja executado por instantes e, depois, analisa o perfil de acesso

da memória *cache* (baseado no tempo de resposta do acesso). Como ela está diretamente relacionada a memória principal, o atacante pode inferir dados sensíveis a partir desse perfil;

- *Vulnerabilidades da memória DRAM*: Esse tipo de vulnerabilidade de aplica a programas que compartilham a memória RAM como, por exemplo, máquinas virtuais. O perfil de acesso à memória RAM é mapeado, de modo semelhante aos ataques baseados em *cache*, isto é, observando-se o tempo de resposta. Isso ocorre, pois existe um *buffer* de linha na própria memória RAM. Contudo, há muito ruído para o atacante inferir dados sensíveis, então ele apresenta uma acurácia menor que os ataques baseados em *cache*;

- *Vulnerabilidades de execução especulativa (ou Branch Prediction Vulnerabilities)*: Os processadores modernos realizam uma predição de próxima instrução a ser executada mesmo sem ter concluído a instrução corrente. É um mecanismo para aumentar a eficiência computacional. Mas ciente disso o atacante pode, por exemplo, preencher o *buffer* de instruções preditivas para averiguar (medir o tempo de resposta) se o programa vítima executará uma determinada instrução. A partir dessa medição, o ele obtém o perfil de execução do programa vítima. Com o perfil, é possível que ele obtenha algum dado sensível;

- *Vulnerabilidades de hardware*: O sucesso desse tipo de ataque depende muito do conhecimento específico que um atacante possui sobre o *hardware* utilizado na máquina que executa o programa vítima. Se for interessante, o atacante pode ter muito sucesso ao provocar uma negação de serviço pelo fato de conhecer bem as vulnerabilidades de um *hardware*.

Como se pode observar, muitas das superfícies de ataques são baseadas na observação de perfis de acesso à memória. Isso decorre do fato que acessos diretos a conteúdos cifrados ou a endereços de memória protegidos por ambientes de computação segura são muito difíceis de ocorrer.

Por fim, as superfícies de ataques são importantes, pois demonstram meios de aferição das soluções de TEE, muitos dos quais utilizados por diversos autores.

3.3. Intel SGX

Esta seção descreve o funcionamento do Intel SGX (*Software Guard Extensions*), a solução TEE que é o foco do minicurso.

Conforme mostra a Figura 3.1, o SGX divide a aplicação em duas partes: uma região não-confiável (não-crítica) e uma região confiável (crítica). A primeira envolve todo o sistema operacional e grande parte do código da aplicação. A segunda, chamada de enclave, acessa as informações confidenciais da aplicação. Essas informações são cifradas numa região específica da memória denominada PRM (*Processor Reserved Memory*), onde qualquer acesso externo é negado (detalhes na Seção 3.3.1). É recomendado que a quantidade de código na parte confiável seja o menor possível, de modo a diminuir a superfície de ataque e reduzir o uso de memória protegida. Além disso, os enclaves

devem ser projetados de modo a ter o menor número possível de interação com a parte não-confiável.

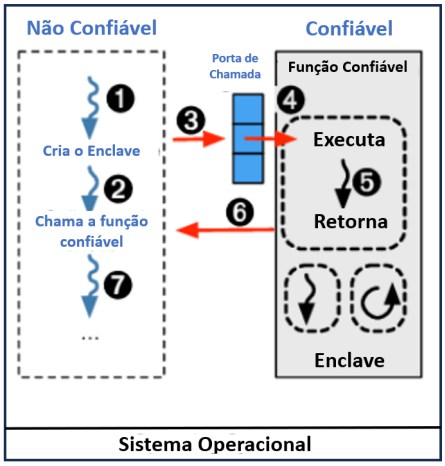

Figura 3.1. Arquitetura do SGX da Intel. Adaptado de [Shih et al. 2016].

3.3.1. A organização da memória

Todo o código e os dados de um enclave são armazenados na PRM (*Processor Reserved Memory*), uma parte isolada da memória DRAM (*Dynamic Random Access Memory*). A PRM é dividida em duas partes principais, a EPC e a EPCM, conforme mostra a Figura 3.2. A CPU protege a PRM de todos os acessos que não sejam do enclave, inclusive do *kernel* e *hypervisor*. O endereço da PRM é definido pela BIOS durante o boot.

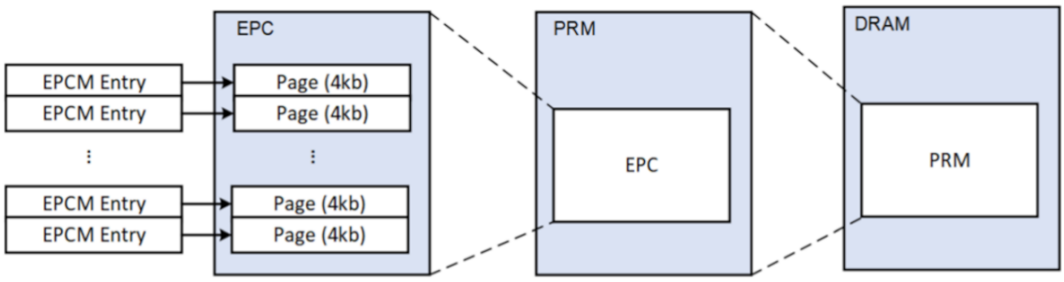

Figura 3.2. Organização da memória do SGX. Fonte: [Fei et al. 2021].

A *Enclave Page Cache* (EPC) é a região de memória onde os dados e os códigos das instâncias do enclave são mantidos. Cada página do EPC estabelece uma relação unívoca com o enclave correspondente e possui um tamanho fixo de 4 kB. Assim, diferentes enclaves podem ter o seu espaço único de memória, permitindo que elas coexistam. O gerenciamento das páginas EPC é feito pelo mesmo software não confiável que gerencia o resto do sistema, como o sistema operacional. A *Enclave Page Cache Map* (EPCM) é onde as informações de alocação das páginas EPC são registradas. Ela informa, por exemplo, se a página EPC foi atribuída para algum enclave e para qual. Assim, o SGX consegue verificar as alocações fornecidas pelo SO. Por exemplo, se o SO tentar alocar a mesma página EPC para dois enclaves, a instrução SGX irá falhar.

O SGX possui uma estrutura de dados especial chamada SECS (SGX Enclave Control Structure). Ela contém os metadados de um enclave, e é armazenada em uma página EPC dedicada. Assim, logo que um enclave é criado, uma página EPC é alocada para armazenar o SECS correspondente, e quando o enclave é destruído, a página SECS é desalocada. Como SECS deve ser usado somente pela CPU SGX, eles não são mapeados no espaço de endereçamento do enclave.

3.3.2. Ciclo de vida do enclave

O ciclo de vida de um enclave no Intel SGX consiste em quatro fases principais: criação, carga, inicialização e destruição (Figura 3.3).

- *Criação*: Para criar um enclave, o sistema invoca a instrução *ECREATE*, que salva a nova estrutura SECS em uma página EPC livre.
- *Carga*: O sistema usa a instrução *EADD* para carregar o código inicial e os dados para o enclave. Para isso, essa função salva o código em um EPC, associa essa página EPC no SECS, e salva as informações no EPCM. Após isso, o sistema usa a função *EEXTEND* para atualizar no SECS a *medição* do enclave, que será usado no processo de atestação.
- *Inicialização*: Após o sistema executar toda a carga inicial do enclave, o sistema inicializa o enclave através da instrução *EITN*. Nesse momento, o enclave não pode mais carregar dados através da instrução *EADD*, e o enclave está apto para receber as instruções do usuário.
- *Finalização*: Para finalizar a execução do enclave, o sistema chama a instrução *EREMOVE*. Assim, todas as páginas EPC utilizadas pelo enclave são liberadas, assim como o SECS correspondente.

Duas instruções importantes são as *ECALL* ("Enclave call"), e o *OCALL* ("Out call"). O *ECALL* é utilizado pelo código externo para chamar as funções protegidas pelo enclave. Similarmente, a instrução *OCALL* é uma chamada de dentro do enclave para a aplicação externa (e.g., IO).

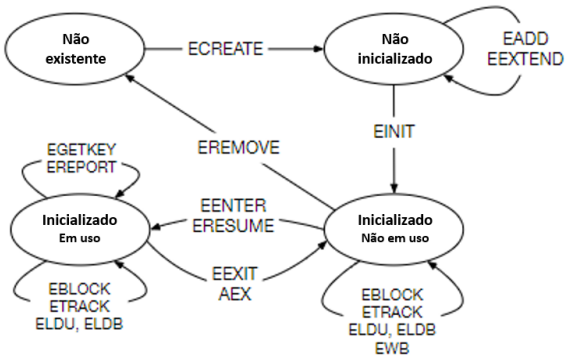

Figura 3.3. Diagrama de estados com o ciclo de vida de um enclave. Adaptado de: [Intel Corporation 2021].

3.3.3. Atestação Remota

Esta Seção caracteriza o processo de atestação remota no Intel SGX[1], um processo que permite que enclaves forneçam provas de que foram corretamente configurados e que podem ser confiados.

Para isso, o SGX usa um tipo de enclave especial, chamada de *enclave de citação* (*Quoting Enclave*). Esse enclave armazena de maneira segura a *chave de atestação*, utilizada para provar a confiabilidade de todos os enclaves da máquina. Enclaves de citação solicitam a chave de atestação para um serviço remoto da Intel (*Intel Provisioning Service*) através de um processo chamado provisionamento.

Assim, as aplicações podem utilizar o enclave de citação para atestar os seus enclaves. A Figura 3.4 mostra os passos envolvidos na atestação remota:

- Primeiramente, o provedor de serviços (desafiante) constrói e envia um desafio para a aplicação (flecha 1 do diagrama da Figura 3.4);
- A aplicação então envia o desafio para o seu enclave, que constrói um relatório (*report*, no contexto do SGX) que reflete tanto o estado da plataforma quanto do enclave – flechas 2 e 3 da figura;
- A aplicação envia esse relatório para o enclave de citação, que verifica a corretude do enclave de aplicação (atestação local). Se estiver correto, o enclave de citação constrói um *quote* (citação, em tradução livre) que comprova a corretude do enclave de aplicação, processo descrito nas flechas 4 e 5.
- Esse *quote* é enviado para o desafiante, que o redireciona para o serviço de atestação da Intel. Esse serviço verifica a validade do *quote* e retorna o resultado para o desafiante – flechas 6 e 7.

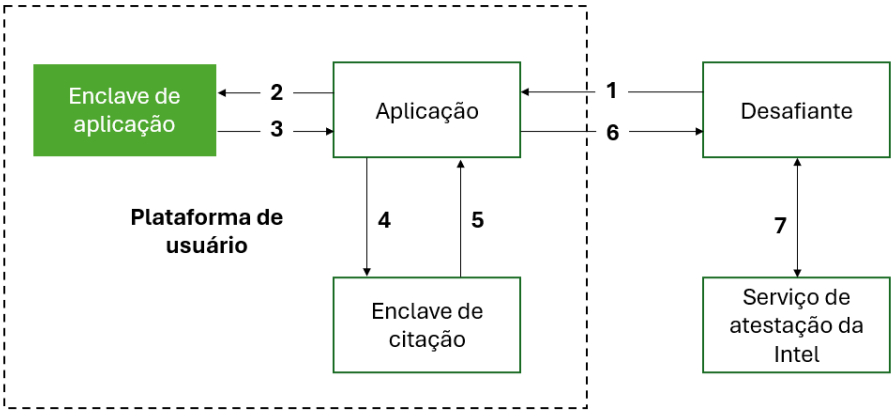

Figura 3.4. Fluxo de atestação remota.

[1] Intel SGX Attestation Documentation

3.3.4. Segurança do Intel SGX

Intel SGX assume um modelo de atacante bastante poderoso: exceto pelo software do enclave, o SGX não confia em nenhum software rodando no processador, incluindo o sistema operacional, o hypervisor, o firmware (BIOS), ou os drivers. As únicas coisas confiáveis são o hardware do processador, e os enclaves fornecidos e assinados pela Intel. Assim, mesmo que um usuário implemente um software SGX num provedor de nuvem malicioso, o atacante não conseguirá quebrar as proteções do enclave. Desse modo, o SGX deve garantir: 1) confidencialidade, de modo que nenhum dado ou código do enclave seja acessível e 2) integridade, de modo que qualquer modificação no código ou nos dados de um enclave possa ser detectado. Note que ataques comprometendo a disponibilidade (DOS) estão fora de escopo. Afinal, o SO sempre pode impedir a execução do enclave ao interromper a fonte de energia.

Analisar a segurança do SGX é ainda mais importante se considerarmos o impacto caso um enclave é comprometido. Como o SGX previne qualquer tipo de auditoria ou análise nos enclaves, é difícil para o SO detectar um comprometimento do enclave. Além disso, um enclave comprometido não só pode extrair os segredos da aplicação, mas também servir potencialmente como *man-in-the-middle* (MiTM) ao comunicar-se com outros enclaves.

Essa seção descreve alguns dos principais ataques no Intel SGX, seguindo a taxonomia sugerida por [Fei et al. 2021]. Em seguida, também descreveremos possíveis mitigações.

3.3.4.1. Ataques

Uma das categorias de ataques são os ataques baseados na **tabela de páginas** (*Page Table attacks*). Esse é um tipo de ataque de canal lateral, que se exploram os padrões de acesso à memória para inferir dados sensíveis. Uma das maneiras é explorando a falha de páginas (*Page-fault*). Intuitivamente, o atacante edita a região da memória em que a aplicação está localizado. Dessa forma, sempre que a aplicação tentar acessar essa região, será gerado uma exceção de falha de página. Assim, ao investigar quando e quais regiões da memória estão sendo acessados, o atacante pode extrair alguma informação sensível da aplicação alvo. Uma outra abordagem é explorar outros atributos da tabela de página (*Fault-less page table attacks*) ou o comportamento de cache da parte não protegida da tabela de páginas.

Outra categoria de ataque de canal lateral são os exploram o comportamento do **cache da CPU** (*CPU Cache Vulnerabilities*). Como o acesso a memória cache é muito mais rápido do que o acesso a memória DRAM, ao medir o tempo de execução da aplicação, é possível saber se os dados estão no cache (*cache hit*) ou se foi necessário buscar os dados na memória (*cache miss*). Assim, considerando que os caches são compartilhados para diferentes CPUs, o adversário pode manipular o cache para provocar *cache hits* e *cache misses* e extrair informações sensíveis.

Existem também ataques que exploram a técnica de **execução especulativa**[2].

[2]Execução especulativa é uma técnica utilizada em processadores modernos para aumentar o desempe-

Como o histórico de execução especulativa não é descartado quando há uma mudança entre o modo enclave e o não-enclave, o atacante pode acessar informações de baixa granularidade sobre a previsão. Desse modo, ataques podem utilizar técnicas semelhantes ao *Meltdown* ou ao *Spectre* [Kocher et al. 2019]. Por exemplo, o ataque *Foreshadow* conseguiu extrair de um enclave SGX não só informações confidenciais da memória, mas também a chave de atestação privada da máquina [Van Bulck et al. 2018, Weisse et al. 2018]. A mitigação para o ataque *Foreshadow* foi publicada em atualizações pela Intel.

Se o atacante tiver acesso ao código da aplicação, ele pode tentar os **ataques baseados em ROP** (*Return-Oriented Programming attacks*). Essencialmente, atacantes exploram pontos de "estouro de buffer"no software dentro do enclave para sobrescrever o endereço de retorno de uma função, sequestrando o fluxo de controle do programa alvo. Dessa maneira, o atacante poderia executar um código malicioso, extraindo segredos e comprometendo a integridade.

Em 2017, um grupo de pesquisadores desenvolveu uma ferramenta para demonstrar um ataque, cujo objetivo era furtar uma chave AES de algumas implementações de bibliotecas SSL. A ferramenta se chama **CacheZoom** [Moghimi et al. 2017] e implementa um ataque do tipo canal lateral.

Os pesquisadores observaram o comportamento do SGX e perceberam que apesar da memória RAM ser cifrada e ter os os acessos aos seus endereços protegidos, havia uma vulnerabilidade na memória cache L1D (cache nível 1 de dados). Os endereços da memória cache não podem ser acessados diretamente por um programa (por construção do processador). Contudo, é possível realizar testes para verificar se um determinado valor se encontra na memória cache. O teste funciona do seguinte modo: se o programa lê um dado de um determinado endereço e o valor é obtido com um tempo de resposta extremamente baixo, provavelmente está na memória cache L1D. Caso contrário o tempo de resposta é alto, pois ele precisa ser obtido de outra memória: L2, L3 ou RAM.

Cientes desse fato e, conhecendo a implementação das bibliotecas SSL, os pesquisadores também perceberam que as implementações de alto desempenho se apoiam no uso de endereçamento de memória para executarem mais rapidamente. Ou seja, ao usar uma biblioteca SSL cuja implementação usa o endereçamento para ficar mais rápida, tais conteúdos dos endereços são trazidos e armazenados nas memórias caches, inclusive a L1D.

Então eles construíram um ataque que inicialmente preenche a memória cache com dados conhecidos - isso é realizado fazendo a leitura de uma região de memória do programa atacante. Com o controle do SO, o ataque permite que a biblioteca SSL consiga executar por poucos instantes. Após, o programa atacante testa a memória cache L1D para averiguar quais endereços foram utilizados pela biblioteca SSL. Conhecendo o endereço base das páginas de memória do programa vítima (SSL) e conhecendo as fórmulas da lógica de mapeamento da memória L1D, o ataque consegue montar um perfil

nho ao prever o fluxo de execução de um programa. Quando um processador encontra uma instrução de desvio (como um branch), ele pode não saber imediatamente qual caminho seguir, pois isso depende do resultado de instruções anteriores que ainda não foram executadas. Para evitar períodos de inatividade, o processador tenta adivinhar qual será o resultado do desvio e executa as instruções que ele prevê que serão necessárias.

dos endereços utilizados e, pouco a pouco, consegue obter informações suficientes para remontar a chave criptográfica usada na biblioteca SSL.

Por fim, vale comentar o ataque **SGX-bomb**, um ataque que explora o *bug RowHammer* para realizar um DoS na máquina alvo. Ao violar a integridade da memória do enclave, o processador fica indisponível, necessitando uma reinicialização do sistema para voltar a funcionar. Assim, um atacante pode subir um enclave malicioso para impactar o servidor, um ataque particularmente preocupante em ambientes de nuvem pública.

3.4. Outros TEEs Existentes

Esta seção descreve alguns dos TEEs existentes tendo-se em vista os desafios existentes [Jauernig et al. 2020] em contraste com o caráter promissor que o TEE tem em comparação com TPMs [Arthur and Challener 2015], *hardware secure modules* (HSMs, [Song et al. 2024]) e *secure elements* (SEs, [Vauclair 2011]). Após diversos trabalhos explorando as vulnerabilidades em implementações de chips com suporte para TEEs [Ghaniyoun et al. 2023], tecnologias como o *ARM TrustZone* surgem para atendimento da demanda crescente de TEEs em sistemas embarcados e dispositivos móveis inicialmente. Em 2021 que a ARM expandiu seu escopo com a apresentação da *Confidential Compute Architecture* [Architecture 2024] visando contemplar a utilização de TEEs em provedores de nuvem com arquitetura ARM.

3.4.1. ARM TrustZone

O *ARM TrustZone* [ARM 2024] é implementado como uma extensão de hardware nas CPUs ARM. O TrustZone divide a CPU em dois mundos: o normal e o seguro. O primeiro é onde o sistema operacional e os aplicativos são executados normalmente enquanto o segundo refere-se à região onde os códigos e os dados autorizados têm autorização para serem executados e armazenados. Trata-se de uma solução amplamente integrada com tecnologias Samsung e Google [Ning et al. 2023].

De celulares, *tablets*, *notebooks* e dispositivos IoT, o ARM TrustZone é capaz de prover isolamento de memória inviabilizando acessos indevidos pelo processador ou pela DMA, possibilidade de cifração de memória, isolamento de periféricos bloqueando acessos maliciosos em busca de dados armazenados nesses dispositivos, de interrupções com a captura de sinais trocados entre hardware e software (ataques *man-in-the-middle*, por exemplo) e isolamento da Cache e do *Translation Lookaside Buffer* (TLB).

3.4.2. ARM CCA

A solução da ARM voltada para o uso de TEEs em servidores *ARM-based* é o ARM CCA. Seus objetivos principais são dois: confiança mínima e garantia de confiança durante atestação. É importante notar que comparando-se com o TrustZone, a diferença é notável. O ARM TrustZone visa a proteção de códigos e dados ao acesso de software (e.g., SO em *Rich Execution Environment* – REE). A arquitetura do CCA é definida por de maneira semelhante, no entanto. Os projetistas usam o mesmo conceito de mundo, mas agora, em vez de normal e seguro, adicionam-se os termos *root* e *realm world* existindo, portanto, quatro estados de segurança no CCA.

O mundo normal e o mundo seguro no TrustZone e no CCA são parecidos a menos

das relações de pertencimento de cada: enquanto que no CCA, dos níveis de exceção (*exception levels* – EL), o último (de número 3) pertence ao *root world*, no TrustZone ele pertence ao mundo seguro [Huang et al. 2024]. No root world, o monitor é responsável pelo boot seguro e troca de contexto entre diferentes mundos. O realm world encarrega-se da execução de aplicações de terceiros e é isolado do mundo normal e do mundo seguro (respectivamente, *normal world* e *secure world*). A Figura 3.5 ilustra esses elementos.

Em termos de mecanismo de bloqueio de acesso, o CCA adiciona a funcionalidade de *Granule Protection Check* (GPC) para bloqueio de acesso inválido às regiões de memória dedicadas a partir de uma tabela de proteção granularizada no root world. Para além disso, a GPC ainda é responsável pelo controle de acesso de dispositivos.

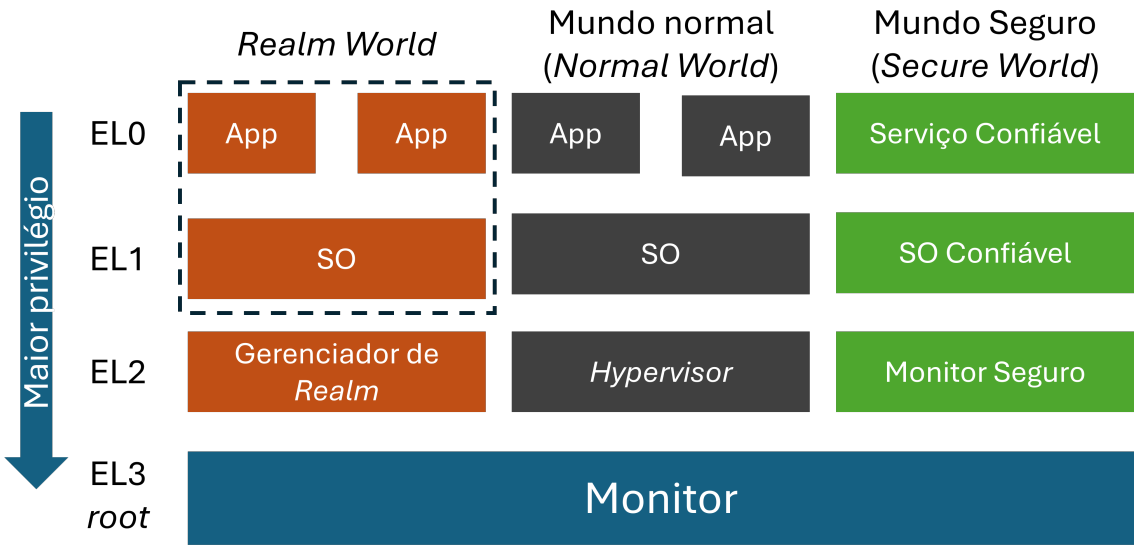

Figura 3.5. Ilustração para representar os níveis de exceção e suas correlações com o mundo normal (*normal world*), seguro (*secure world*) e o *Realm world*. Adaptado de [Architecture 2024]

Em termos de segurança, o CCA possui mecanismo nativo de cifração de memória, protegendo, assim, códigos e memória em zonas dedicadas. Para cada mundo existe uma chave de cifração, o que garante o isolamento especial e maior resiliência de ataques de *dump* da memória uma vez que o atacante precisa da chave de decifração correspondente. Além disso, o CCA possui mecanismo de atestação assistido por hardware para as máquinas virtuais em execução no *realm world*. A esse processo dá-se o nome de *realm attestation*. A menos do processo de isolamento de interrupções, o CCA estende ou supera os mecanismos de segurança do ARM TrustZone.

3.4.3. AMD SEV

A AMD introduziu a sua solução de TEE em 2017, o *Secure Encrypted Virtualization* (SEV) [Kaplan et al. 2016]. Ao contrário do Intel SGX, cujo foco inicial era aplicações executando do lado do cliente, o SEV foi projetado para execução em nuvem. Assim, o AMD SEV apresenta um isolamento em nível de máquina virtual (VM), permitindo que as aplicações rodem em um espaço protegido sem nenhuma modificação em seu código [Jauernig et al. 2020]. Em geral, o SEV oferece um desempenho melhor do que SGX em

aplicações complexas ou com computação intensiva [Mofrad et al. 2018].

Para isolar uma VM, o SEV cifra o código e a memória da VM com uma chave individualizada. O acesso a essa chave é limitada por hardware, de modo que não é acessível por outros softwares, como o hypervisor ou o próprio SO. Posteriormente, a AMD introduziu também o SEV-ES (*Encrypted State*), que adiciona maior proteção ao cifrar o conteúdo dos registradores sempre que a VM é suspensa [Kaplan 2017]. Uma outra funcionalidade é o SEV-SNP (*Secure Nested Paging*) que traz proteções de integridade na memória [Sev-Snp 2020].

Com relação a segurança, existem alguns ataques publicados. Um deles é o **CrossLine** [Li et al. 2021], que possui duas versões: v1 e v2. No artigo os autores demonstram uma vulnerabilidade aplicável ao AMD-SEV capaz de expor os dados sensíveis de uma aplicação vítima executada dentro de uma máquina virtual da vítima (VM-vítima). O experimento foi realizado em um computador com o processador AMD EPYC 725, utilizando o SO Ubuntu 64-bit 18.04 e o *hypervisor* QEMU 2.12.

Como preparação, o atacante já conseguiu previamente executar um *hypervisor* modificado que hospeda a VM-vítima. Adicionalmente, ele consegue executar uma máquina virtual (VM-atacante) no mesmo *hypervisor*. A VM-atacante se encontra pausada após seu início. O ataque *CrossLine* v1 funciona com os seguintes passos:

1. O atacante limpa o bit de presença: Em todas entradas de páginas de memória da VM-atacante, com a futura intenção de produzir uma falha de página de memória, forçando sua futura reconstrução;

2. Remapeamento do registrador contendo o ponteiro de base das páginas de memória (gCR3) da VM-atacante: O *hypervisor* modifica o registrador gCR3 da VM-atacante e o aponta para uma página de memória da VM-vítima;

3. Modificação do bloco de controle da máquina virtual (VMCB ou *Virtual Machine Control Block*) da VM-atacante: O *hypervisor* copia o identificador da máquina virtual (ASID[3]) da VM-vítima para a VM-atacante, de modo que ambas compartilhem a mesma chave de cifração/decifração;

4. Especificação do deslocamento (ou *offset*) na página de memória alvo: O *hypervisor* modifica o ponteiro de instrução (nRIP) e aponta ele para o endereço da página de memória da VM-vítima que vai ser atacado. O endereço é formatado de acordo com o mapeamento realizado pelo SO Linux (utilizado no experimento dos autores);

5. Extração de segredos a partir de falhas das páginas de memória aninhadas (ou *nested*): A VM-atacante é retirada da pausa. Nesse momento, o processador obtém a próxima instrução a ser executada, mas como as entradas de página de memória estão limpas, a VM-atacante vai obter eles a partir do endereço de base das páginas de memória gCR3 (já previamente copiado da VM-vítima). A operação vai falhar, mas um bloco de memória com 8 *bytes* será extraído do endereço e notificados para

[3]ASID (ou *Address Space ID*) é o identificador dos meta dados de uma máquina virtual. Um desses dados é a identificação da chave criptográfica associada com a máquina virtual.

a VM-atacante em seu VMCB, mais especificamente no registrador EXITINFO2. Ou seja, uma leitura de um bloco de dados da VM-vítima bem sucedida.

Por meio da repetição dos passos acima, é possível alterar a página de memória a ser atacada, bem como o ponteiro de instrução para, aos poucos, obter blocos de 8 *bytes* de dados em cada ataque. O ataque **CrossLine** v1 é silencioso, ou seja, indetectável para a máquina virtual da vítima.

3.5. Ferramentas para desenvolvimento de aplicações

Existem dois paradigmas para desenvolver uma aplicação protegida por TEE: através de SDKs (Software Development Kit), ou através de frameworks de mais alto nível (como LibOS).

A primeira maneira é mais baixo nível, envolve separar mais minunciosamente a parte confiável e não-confiável do software, e desenhar cuidadosamente a iteração entre elas. Para isso, é possível utilizar SDKs (como Intel SGX SDK ou Open Enclave SDK), para definir a interface entre a parte confiável e a não-confiável. Geralmente esse processo é bastante complexo, envolvendo bastante depuração (ou *debug*). Além disso, é preciso refatorar a aplicação inteira, saber quais são os recursos a serem protegidos e os trechos de código que utilizam eles. Entretanto, essa abordagem permite que o desenvolvedor construa o enclave de modo a proteger contra ataques de canal lateral. Ferramentas automatizadas podem ajudar no desenvolvimento, como por exemplo, no particionamento automático das aplicações [Lind et al. 2017]. Tecnologias como o Open Enclave SDK ainda se propõe a ser independente de hardware, com a proposta de suportar diversas implementações.

A segunda abordagem envolve inserir a aplicação inteira na parte confiável, não se preocupando em separá-la. Isso simplifica bastante o desenvolvimento, e permite que as aplicações usufruam dos benefícios de ambientes de computação segura sem nenhuma modificação no seu software. Para isso, são utilizados sistemas operacionais como bibliotecas do SO, ou LibOS (do inglês *Library Operating System*), que atuam como uma camada intermediária entre o sistema operacional e as chamadas de um enclave (vide Figura 3.6). Assim, as aplicações realizam as suas chamadas normalmente para o LibOS, que fica responsável por realizar as chamadas fora do enclave de modo seguro.

Com relação às desvantagens, pode ser mencionado que código extra deve estar dentro do enclave, o que acaba impactando o seu desempenho por dois motivos: utilização de memória e tradução de chamadas de sistema (*system calls* comumente abreviadas para *syscalls*) emuladas para o sistema operacional que efetivamente realiza a operação. Além disso, menores quantidades de código binário abrigado em um enclave implicam maior resistência a tentativas de quebra de segurança [Intel 2024] tornando, portanto, uma LibOS uma alternativa mais arriscada.

3.5.1. Intel SGX SDK

O Intel SGX SDK é composto por um conjunto de bibliotecas, ferramentas e APIs que auxiliam os desenvolvedores a criar aplicativos que utilizam o hardware Intel SGX. Ele provê uma camada de abstração entre a aplicação e as instruções de baixo nível, forne-

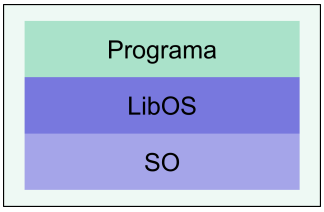

Figura 3.6. Ilustração de camadas ao entorno de uma LibOS. O programa usa a LibOS como se estivesse usando o SO.

cendo uma interface de alto nível para a inicialização de enclaves, comunicação com a parte não-confiável e gerenciamento de chaves.

A Intel sugere também um roteiro para facilitar o projeto de aplicações protegidas pelo SGX composto dos seguintes passos: identificar os recursos a serem protegidos, identificar os códigos que utilizam tais recursos e definir cuidadosamente interface do enclave. Para definir a interface, o desenvolvedor precisa descrever as chamadas que o código não-confiável faz ao enclave (ECALLs), e as chamadas que o enclave faz para o código não-confiável (OCALLs). Tanto ECALLS quando OCALLs são definidas através da *linguagem de definição de enclave* (EDL).

Esse SDK se encontra disponível para os sistemas operacionais Linux e Windows possibilitando, ainda, o uso a partir de algumas linguagens de programação como *Rust* e C++. Neste trabalho as experiências práticas utilizam o Intel SGX SDK dentro do contexto do Linux. Ele é composto por diferentes programas, são eles:

- *Intel SGX Driver SGX*: contém o *driver* para utilização do hardware SGX. Contudo, se o versão do Kernel Linux for igual ou superior à 5.1, o *driver* já estará integrado no próprio Kernel;
- *Intel SGX PSW - Platform Software*: contém os módulos que permitem a execução de enclaves no Linux (em tempo de execução - *runtime*), o Intel SGX Architecture Enclaves, a Intel SGX Runtime System Library e o Intel SGX Architecture Enclave Service Manager (AESM);
- *Intel SGX DCAP - Data Center Attestation Primitives*: contém as funcionalidades para a realização de atestação;
- *Intel SDK*: são as bibliotecas para serem referenciadas e integradas ao programa em construção.

Na parte prática deste documento, há um código-fonte desenvolvido em C++ para Linux que demonstra o uso desse SDK (ver subseção 3.6.3).

3.5.2. Open Enclave SDK

O Open Enclave SDK (OESDK) é um conjunto de desenvolvimento de software com código aberto voltado para a criação de uma única abstração unificada para enclaves [OESDK 2024]. Assim, o OESDK permite o desenvolvimento de aplicações com TEE

sem se preocupar com as particularidades de cada fabricante, abstraindo a complexidade do Intel SGX SDK.

Atualmente o Open Enclave SDK é compatível com apenas duas implementações de hardware: Intel SGX e ARM TrustZone. Mas é importante ressaltar que o suporte ao ARM TrustZone só é possível se ele estiver utilizando o OP-TEE [OP-TEE 2024], que é um conjunto de bibliotecas e *drivers* para uso de TEE no ARM.

Dentre algumas das funcionalidades que o Open Enclave SDK é capaz de abstrair, podem ser mencionadas: atestação, medição (ou *measuremet*) de um TEE e cifração de dados para armazenamento (também conhecido como *data sealing*). Esse SDK se encontra disponível para os sistemas operacionais Linux e Windows. Ele também possibilita o seu uso a partir de duas linguagens de programação: C e C++.

3.5.3. Occlum

O Occlum é uma LibOS de código aberto, compatível com a tecnologia Intel SGX [Shen et al. 2020]. Uma das características dela é prover um ambiente de execuções multitarefa de modo eficiente, permitindo que diversos processos sejam abrigados em uma única instância do Occlum. Esse é um fato importante sob a ótica de desempenho, pois muitas aplicações são compostas por diversos processos e, se eles podem compartilhar a mesma LibOS, há a economia de recursos computacionais utilizados, e.g., menor consumo de processamento e memória. Isso também facilita que os processos comuniquem os dados entre si, já que todas eles rodam dentro de um mesmo enclave.

Um exemplo de aplicação que utiliza de diversos processos em um computador é o servidor MySQL. Quando ele está em execução na LibOS do Occlum, seus processos se comunicam diretamente sem cifração e decifração de dados. Mas caso se cada processo estivesse abrigado em uma LibOS diferente, seria necessária a cifração e decifração na comunicação entre eles.

Além disso, o Occlum é uma LibOS que usufrui de mecanismos de confiabilidade. Isso é um benefício de ter sido construído na linguagem de programação *Rust* que, de acordo com o fabricante, possui mecanismos para garantir confiabilidade [Rust 2024].

Uma funcionalidade de grande utilidade oferecida pelo Occlum é o sistema de arquivos seguro para os programas executados sobre a LibOS. O sistema de arquivos seguro é transparente para os programas em execução, isto é, do ponto de vista dos programas, os arquivos que eles leem ou escrevem estão em um sistema de arquivos (ou *filesystem*) aparentemente sem cifração. Porém, o sistema de arquivos percebido pela aplicação é, na verdade, cifrado sobre o sistema de arquivos real. A Figura 3.7 ilustra essa funcionalidade no diagrama da arquitetura (localizada no canto inferior direito). Há uma subseção na parte prática desse documento de demonstra o uso do Occlum (subseção 3.6.4).

3.6. Parte Prática

A fim de demonstrar o funcionamento correto de tecnologias para ambientes de computação segura, foram criados alguns experimentos para averiguar se os dados abrigados permanecem inacessíveis ou confidenciais em tais ambientes.

Todos os experimentos propostos possuem como cenário um servidor fictício de

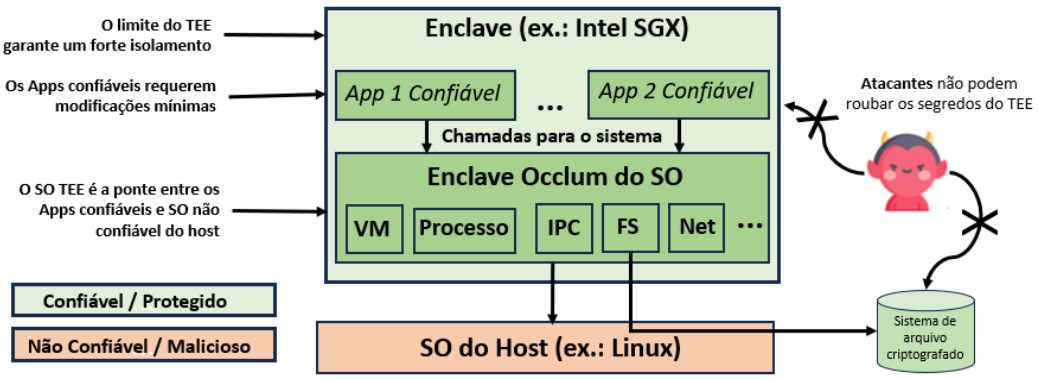

Figura 3.7. Arquitetura interna do Occlum. Adaptado de [Occlum 2024].

autenticação. Nesse cenário, há um atacante que tem acesso ao console como usuário *root* na mesma máquina que executa o programa autenticador. O atacante tem interesse em obter os dados sensíveis de algum usuário – por vezes ele pode até mesmo conhecer um login. O atacante não tem interesse em derrubar o autenticador executando um ataque de negação de serviço (ou *Denial of Service - DoS*), pois isso eliminaria sua furtividade. Ele está interessado em capturar dados sensíveis. De uma forma geral, imagina-se os seguintes passos:

i) O sistema de autenticação se encontra no ar;

ii) O atacante adquire acesso *root* ao console da máquina que executa o autenticador;

iii) O atacante realiza uma tentativa de autenticação com algum usuário que ele sabe que existe, mas não conhece a senha;

iv) Prontamente, o atacante realiza uma captura (ou *dump*) de memória para obter os dados confidenciais do usuário em questão (ou outros se ele tiver a oportunidade), visto que no momento da autenticação, o atacante sabe que o programa de autenticação precisa ter dados confidenciais do usuário carregados na memória para verificar se ele foi autenticado ou não.

Para o cenário proposto anteriormente, há uma programação de experimentos que envolvem a aprendizagem de uma ferramenta de ataque a fim de que, em experimentos subsequentes, seja possível averiguar se o ataque foi bem sucedido ou não. Caso o ataque seja bem sucedido, há a certeza que a tecnologia falhou em proteger os dados confidencias. Caso contrário, significa que a tecnologia conseguiu atingiu os requisitos de proteção dos dados. Com relação à parte tecnológica de apoio aos experimentos, são necessários os seguintes programas e infraestrutura:

- Computador ou máquina virtual com processador Intel que possui as instruções *SGX* e *FLC*;
- Sistema Operacional Ubuntu 22.04 em execução no computador;

- Programa *gcore* [Ubuntu Manpage Repository 2024] instalado - pode ser instalado por meio do pacote *gdb* caso não esteja presente. Este programa é o responsável por realizar as capturas de memória (ou *dump*);

- Programa *Docker*[4] [Docker, Inc. 2024] para simplificar a execução de programas, criando instâncias (ou *containers*) deles a partir de imagens;

- Programa *Git* [Software Freedom Conservancy 2024] para baixar os códigos-fonte dos experimentos, disponíveis em um repositório na plataforma *GitHub* [GitHub, Inc. 2024] por meio do endereço Web: `https://github.com/larc-sbseg-2024-demo/amb-comp-segura`;

- Ferramenta de visualização de *dump* ou arquivos binários de preferência. No caso foi utilizado o 'HexEdit version 4.0' do Windows para visualização dos arquivos de captura de memória.

3.6.1. Experimentos com a ferramenta de captura ou *dump* de memória

Nesse primeiro experimento, o objetivo é conseguir furtar um dado sensível de autenticação de acordo com o cenário explicado anteriormente, em especial um *hash*. Na montagem do ambiente experimental assume-se a existência de um servidor de autenticação executando na máquina vulnerável ao ataque, mas a base de dados se encontra em um ambiente protegido do atacante.

Para simular esse experimento, é necessário executar a base de dados MongoDB [MongoDB, Inc. 2024] que é utilizada do experimento. As instruções específicas se encontram no repositório GitHub. Após, é necessário executar o servidor de autenticação. O servidor de autenticação foi desenvolvido na linguagem *python* e também se encontra no repositório GitHub.

Logo que o servidor estiver em execução, são executados os passos *iii* e *iv*, simulando as ações do atacante. O usuário vítima do ataque é o usuário 'asilva'. Há um *shell script* para facilitar essas ações. É o arquivo 'dump-vitima.sh' que se encontra na pasta 'ferramenta-dump'. Ao final, é produzido um arquivo com prefixo 'dump.bin'. Se o *hash* de autenticação do usuário 'asilva' for encontrado no arquivo, o ataque foi bem sucedido e, consequentemente, a aplicação da ferramenta de *dump* de memória. A Figura 3.8 ilustra uma captura bem sucedida.

É importante destacar que nem sempre o ataque é bem sucedido, pois as variáveis de memória que armazenam o *hash* para comparação podem ser sobrescritas rapidamente pela própria pilha de execução do programa. Nesse caso não é possível encontrar o segredo do usuário 'asilva'.

O resultado esperado desse experimento é demonstrar a ferramenta de captura ou *dump* de memória em funcionamento (*gcore*), pois ela será utilizada como ferramenta nos próximos experimentos para simular os ataques.

[4]Instalável a partir de `https://docs.docker.com/engine/install/`

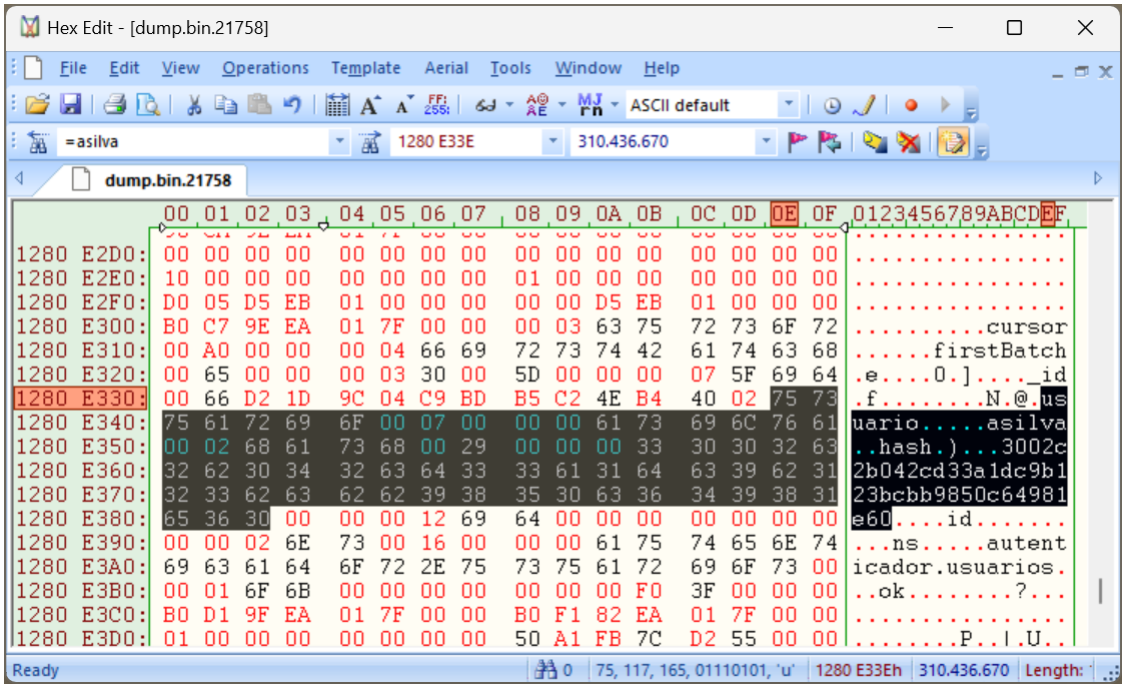

Figura 3.8. Exemplo de captura realizada e bem sucedida. Destaque para o *hash* encontrado do usuário 'asilva', à direita da imagem.

3.6.2. Apresentação do Autenticador desenvolvido em C++

Nessa parte prática, há a demonstração do código-fonte do Autenticador desenvolvido em C++ que é utilizado em mais de um experimento. O programa foi desenvolvido com o SDK da Intel, utilizando como base de sua construção os exemplos de implementação oferecidos pelo fabricante.

Esse programa simula uma situação na qual os dados de autenticação de usuários já se encontram na memória, dispensando o uso de uma base de dados. De fato, o programa foi construído de modo a facilitar o ataque, pois ao manter os dados sensíveis na memória RAM, eles sempre serão capturados pela ferramenta *gcore* toda vez que o programa estiver em execução e a proteção SGX estiver desativada.

No código-fonte existem dados usuários vítimas que foram programados de forma fixa (ou *hard-coded*) que estão presentes no autenticador. São eles: 'joao', 'marcela' e 'renata'. Bem como seus *hashes*: 'hash-12345678', 'hash-abcde678', 'hash-12354abced'.

A demonstração consiste em uma navegação pelo código-fonte e exposição dos principais arquivos presentes, inclusive a parte confiável e não confiável do programa. A visualização do programa e suas principais partes tem como objetivo facilitar a compreensão dos experimentos descritos a seguir.

3.6.3. Experimentos com Autenticador C++ usando o SDK Intel SGX

Nesse experimento, é utilizado um programa desenvolvido na linguagem C++ mencionado anteriormente. Ao executar o programa, ele permanece em execução por até quinze segundos. Durante esse período, o ataque (captura de memória) representando os passos

iii e *iv* deve ser realizado – caso contrário o processo não mais existirá na máquina. Nesse experimento são realizadas duas simulações:

- SGX desativado: primeiramente o programa é compilado com as instruções SGX desativadas, isto é, no modo 'SIM' (*hardware* simulado). Em seguida, um ataque simulado é realizado e a captura de memória é obtida. Como resultado, é esperado que dados sensíveis como logins e *hashes* estejam expostos na captura;

- SGX ativado: dessa vez o programa é compilado no modo 'PRERELEASE' com as instruções SGX ativadas, ou seja, efetivamente utilizando o *hardware*, mas sem a assinatura de um executável final. Após a realização de um ataque simulado, a captura de memória é obtida. Como resultado, é esperado que nenhum dado sensível esteja exposto, somente dados da parte 'não confiável' (ou *untrusted*) do programa.

Após a execução das duas simulações, os resultados obtidos devem coincidir com o esperado. Ou seja, na simulação com o SGX desativado, os dados dos usuários vítimas devem estar expostos e devem ser facilmente localizados com uma ferramenta de busca em arquivos binários. A Figura 3.9 ilustra uma captura bem sucedida.

Já na simulação com o SGX ativado, não devem ser encontrados os dados sensíveis dos usuários vítimas, com exceção do usuário 'joao', pois há uma tentativa de autenticação na parte não confiável do programa. Nesse caso, a ativação da proteção SGX é evidenciada.

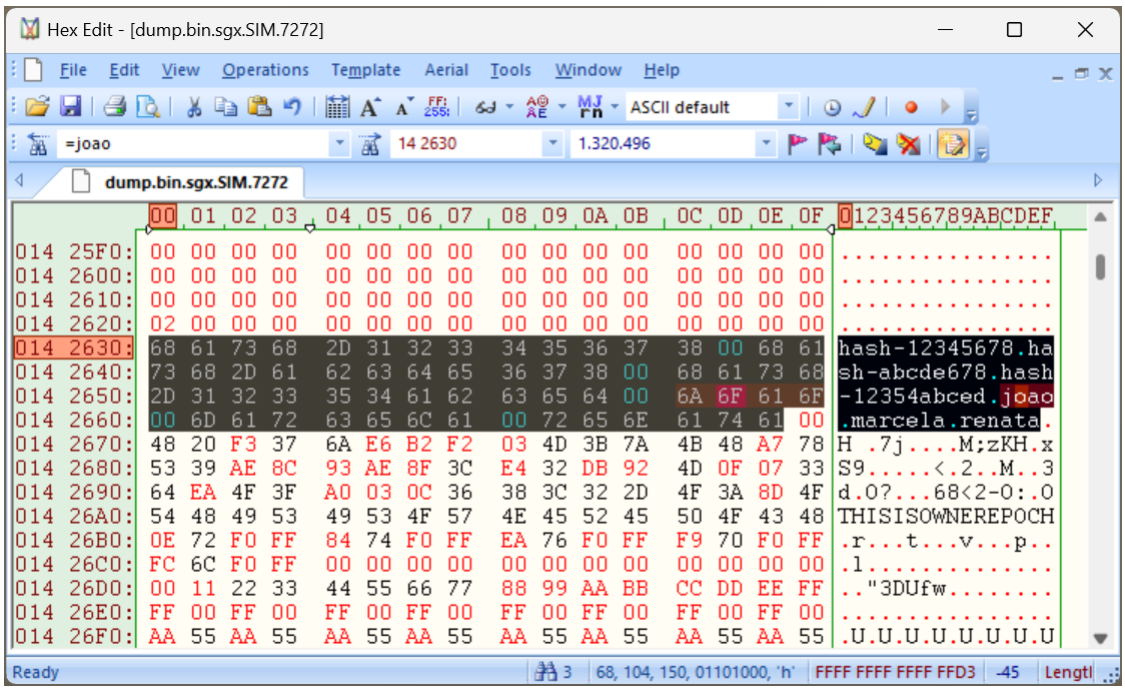

Figura 3.9. Exemplo de captura realizada: Destaque para os usuários 'joao', 'marcela' e 'renata' encontrados, bem como seus *hashes*.

3.6.4. Experimentos com Autenticador C++ usando Occlum

Nesse experimento utiliza-se o *framework* Occlum. A sequência é muito semelhante ao que é realizado no experimento anterior. Mas desta vez utiliza-se o Occlum para proteger o programa autenticador e os dados sensíveis dos usuários com SGX.

Primeiramente executa-se um programa vítima, baseado no Autenticador em C++ previamente apresentado. O programa não é idêntico, a menos a parte correspondente à autenticação. Isso decorre do fato que o Occlum abriga programas por completo sem modificações para executar em SGX. Logo, o programa é modificado para executar em um ambiente comum, sem estar previamente preparado para execução usando o SDK da Intel para SGX. Em outras palavras: é o mesmo programa utilizado no experimento anterior, mas sem os recursos do SDK Intel SGX.

Dessa vez, o programa Autenticador já se encontra compilado e disponível para uso direto: é o programa 'app' sob a pasta 'occlum-demo'. Talvez seja necessário fornecer permissões de execução ao programa usando o comando 'chmod 777'. No programa 'app', os mesmos usuários vítimas do Autenticador desenvolvido em C++ estão presentes no código-fonte. Logo ao iniciar a execução, ele permanece em execução por até quinze segundos. Nesse momento o ataque (captura de memória) deve ser realizado. Igualmente ao experimento anterior, são realizadas duas simulações:

- SGX desativado: primeiramente o ambiente Occlum é instanciado e, em seguida, é compilado com *framework* Occlum no modo 'SIM' (*hardware* simulado). Após, o programa 'app' é executado nesse ambiente. Então um ataque simulado é realizado e a captura de memória é obtida. Como resultado, é esperado que dados sensíveis como logins e *hashes* estejam expostos na captura;

- SGX ativado: dessa vez o ambiente Occlum é instanciado e, em seguida, é compilado com *framework* Occlum no modo 'HW', ou seja, efetivamente utilizando o *hardware* SGX. Após, o programa 'app' é executado nesse ambiente. Então há a realização de um ataque simulado e a captura de memória é obtida. Como resultado, é esperado que nenhum dado sensível esteja exposto absolutamente.

Após a execução das duas simulações, os resultados obtidos devem coincidir com o esperado. Ou seja, na simulação com o SGX desativado, os dados dos usuários vítimas devem estar expostos e ser facilmente localizados com uma ferramenta de busca em arquivos binários (muito semelhante ao ilustrado pela Figura 3.9). Já na simulação com o SGX ativado, não devem ser encontrados os dados sensíveis dos usuários vítimas absolutamente. Nesse caso, a ativação da proteção SGX por meio do *framework* Occlum é evidenciada.

3.7. Experimentos adicionais

Essa seção descreve dois experimentos adicionais. O primeiro explora a ferramenta *Open Enclave SDK* (OESDK), uma alternativa ao SGX SDK. Já o segundo experimento implementa no SGX uma aplicação mais completa usando *Occlum*, com um autenticador e

banco de dados cifrado[5].

3.7.1. Open Enclave SDK (OESDK)

Essa seção exemplifica o uso do Open Enclave SDK (Seção 3.5.2). Assim, o experimento exemplifica o passo a passo de um *Hello World* para a criação e execução de um ambiente de computação segura. Após isso, ele realiza uma alteração simples no código fonte. O experimento utiliza os seguintes recursos computacionais:

- Intel Core I5-9400F com SGX e FLC[6]. O SGX deve ter sido habilitado na BIOS (instruções em apêndice A).
- Memória RAM 8 GB
- HD 100 GB livre
- Kernel Linux 6.0: Debian 12.

Para facilitar a experiência, é utilizada uma imagem Docker fornecida pela Occlum. Essa imagem já contém os recursos da Intel SGX SDK pré-configurados. O passo-a-passo é o seguinte:

i) Em um terminal, instanciar o contêiner Docker do Occlum:

```
sudo docker run -it --network host \
    --device /dev/sgx_enclave:/dev/sgx/enclave \
    --device /dev/sgx_provision:/dev/sgx/provision \
    --name oe -v \
    /home/larc:/home/larc occlum/occlum:latest-ubuntu20.04
```

ii) No terminal do contêiner recém instanciado, adicionar o repositório que contém os pacotes do OESDK:

```
echo "deb [arch=amd64] \
    https://packages.microsoft.com/ubuntu/20.04/prod focal \
    main" | tee etc/apt/sources.list.d/msprod.list

deb [arch=amd64] \
    https://packages.microsoft.com/ubuntu/20.04/prod \
    focal main

wget -qO - https://packages.microsoft.com/keys/microsoft.asc

sudo apt-key add
```

iii) Instalar os pacotes OESDK e outros:

[5]Observação: Os experimentos usam o usuário local do terminal 'larc' para a realização de comandos. Assim, muitos dos comandos e resultados utilizam o diretório raiz (**`/home/larc`**).

[6]FLC - Flexible Launch Control, característica que permite o acionamento de um Enclave sem necessitar de um token fornecido pela Intel. O token pode ser gerado por um segundo Enclave SGX (LE - Launch Enclave) [Intel Corporation 2023]

```
apt update
apt -y install clang-11 libssl-dev gdb \
    libsgx-enclave-common \
    libsgx-quote-ex libprotobuf17 libsgx-dcap-ql \
    libsgx-dcap-ql-dev az-dcap-client open-enclave
```

iv) E então, clonar o repositório do OESDK a partir do GitHub:

```
git clone https://github.com/openenclave/openenclave
```

v) Por fim compilar e executar o primeiro exemplo (do fabricante):

```
cd openenclave/samples/helloworld/build
mkdir build && cd build
cmake ..
make run
```

A seguir temos o resultado da execução do comando *make run* indicado na etapa E. A impressão "Hello world from the enclave!"é realizada a partir do Enclave, com código sendo executado dentro da área segura (ECALL). A impressão "Enclave called into host to print: Hello World!"é realizada através de código sendo executado no host (OCALL), fora da área segura.

```
[ 40%] Built target helloworld_host
[ 80%] Built target enclave
[100%] Built target sign
Hello world from the enclave!
Enclave called into host to print: Hello World!
[100%] Built target run
```

O próximo passo é alterar alguns arquivos para modificar o texto impresso na tela. Para isso, realizamos os seguintes passos:

i) No mesmo terminal do contêiner Docker, utilize o comando 'vi' para editar o arquivo 'enclave/enc.c', que representa o componente seguro e alterar a linha contendo a *string* `Hello world from the enclave!` para `Hello world from the enclave of LARC!`.

```
vi enclave/enc.c
```

Abaixo está o trecho após a alteração:

```
fprintf(stdout, "Hello world from the enclave of LARC!\n");
```

ii) Utilize o comando 'vi' para editar o arquivo 'host/host.c', o qual representa o componente não confiável, e alterar a linha que contém a *string* `Enclave called`

`into host to print: Hello World!` para `Enclave called into host to print: Hello World of LARC!.`

```
vi host/host.c
```

Abaixo está o trecho após a alteração:

```
fprintf(stdout, "Enclave called into host to print: Hello World of
    LARC!\n");
```

iii) Por fim, execute o comando `make run` novamente.

Após a execução, é produzido o resultado a seguir, indicando que as alterações foram bem sucedidas.

```
[100%] Generating enclave/enclave.signed
Created /home/larc/openenclave/samples/helloworld/build/enclave/enclave
    .signed
[100%] Built target sign
Hello world from the enclave of LARC!
Enclave called into host to print: Hello World of LARC!
[100%] Built target run
```

3.7.2. Occlum

O objetivo desse experimento é executar um servidor de autenticação abrigado em um enclave SGX por meio do *framework* Occlum. Esse servidor, por sua vez, se conecta a um banco de dados cifrado hospedado em um ambiente não confiável. A Figura 3.10 mostra essa arquitetura.

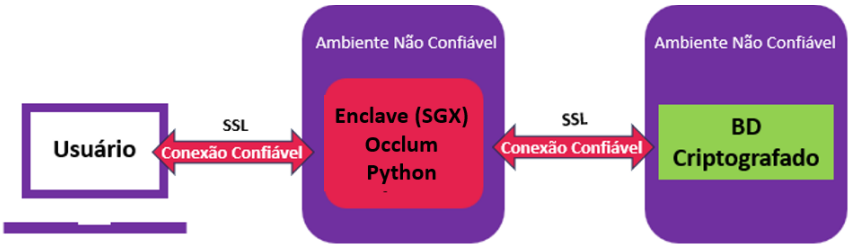

Figura 3.10. Aplicação da experiência com o Occlum

O experimento utiliza os seguintes recursos computacionais:

- Intel Core I5-9400F com SGX e FLC. O SGX deve ter sido habilitado na BIOS (instruções em apêndice A).
- Memória RAM 8 GB
- HD 100 GB livre
- Kernel Linux 6.0: Debian 12.

A Figura 3.11 ilustra os passos necessários.

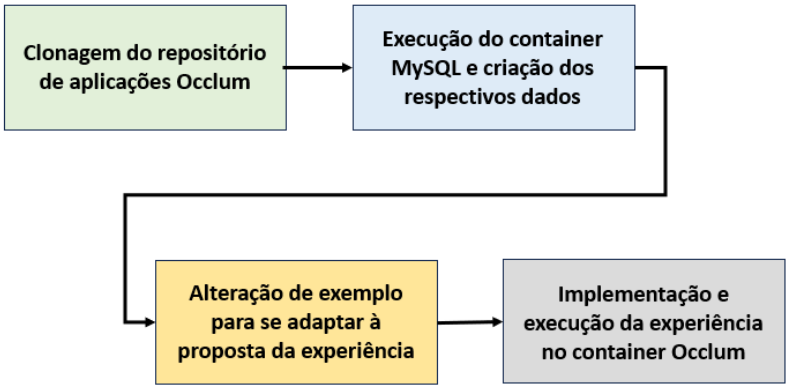

Figura 3.11. Sequência de execução da experiência

i) Clonagem do repositório de aplicações Occlum

Essa experiência utiliza como base uma imagem fornecida pela equipe do Occlum. O diretório **`occlum-fs`** será construído posteriormente na execução da imagem Docker do Occlum.

```
$ mkdir occlum-fs
$ cd occlum-fs
$ git clone https://github.com/occlum/occlum
```

ii) Execução do contêiner do MySQL e criação dos respectivos dados

Nesse passo é descrito como baixar a imagem Docker do MySQL, instanciar e configurar um Banco de Dados. Abaixo estão os comandos para instanciar um contêiner a partir da imagem Docker do MySQL:

```
mkdir mysql-data
docker run  -p 127.0.0.1:3306:3306 --restart unless-stopped  \
    -v /home/larc/mysql-data:/var/lib/mysql \
    --name mysql-doc -e MYSQL_ROOT_PASSWORD=larc123 \
    -d mysql:8.0.36-debian
```

Abaixo estão os comandos para imprimir os logs do contêiner e verificar se ele já está em execução:

```
sudo docker logs mysql-doc
```

Caso o resultado se assemelhe ao resultado da listagem abaixo, a instância (contêiner) MySQL está pronta para receber comandos.

```
2024-06-06T20:06:20.697008Z 0 [System] [MY-010931] [Server] /usr/
    sbin/mysqld: ready for connections. Version: '8.0.36' socket:
    '/var/run/mysqld/mysqld.sock' port: 3306 MySQL Community
    Server - GPL.
```

Nesse momento, é necessário acessar o console do MySQL para executar comandos. O acesso é realizado por meio do comando abaixo:

```
$ mysql -h 127.0.0.1 -P 3306 -u root -plarc123
```

Uma vez no console do MySQL, executar os seguintes comandos para criar e configurar uma base de dados intitulada 'auth_db' e um usuário do MySQL intitulado 'larc':

```
CREATE USER 'larc'@'%' IDENTIFIED BY 'larc123';
GRANT ALL PRIVILEGES ON *.* TO 'larc'@'%';
FLUSH PRIVILEGES;
CREATE DATABASE auth_db;
use auth_db;
CREATE TABLE users (login VARCHAR(255), password VARCHAR(255));
insert into users (login, password) values ('user1', 'passwd1');
insert into users (login, password) values ('user2', 'passwd2');
```

iii) Alteração de exemplo para se adaptar à proposta da experiência

O código original do Occlum exemplifica a autenticação sem se conectar a nenhum banco de dados, apenas mantendo as credenciais em memória. Assim, modificaremos o código de modo a acessar o banco de dados instanciado em um ambiente externo.

Mudar para o diretório a seguir:

```
cd occlum-fs/occlum/demos/python/flask
```

Editar o arquivo **`install_python_with_conda.sh`** no mesmo diretório e acrescentar ao comando da linha 11, o seguinte:

`conda-forge::mysql-connector-python`

Esta dependência permite ao código python se conectar ao banco de dados MySQL. Abaixo está o resultado esperado após a edição:

```
#!/bin/bash
set -e
script_dir="$( cd "$( dirname "${BASH_SOURCE[0]}" )" >/dev/null
    2>&1 && pwd )"

# 1. Init occlum workspace
[ -d occlum_instance ] || occlum new occlum_instance

# 2. Install python and dependencies to specified position
[ -f Miniconda3-latest-Linux-x86_64.sh ] || wget https://repo.
    anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
[ -d miniconda ] || bash ./Miniconda3-latest-Linux-x86_64.sh -b -p
     $script_dir/miniconda
```

```
$script_dir/miniconda/bin/conda create --prefix $script_dir/python
    -occlum -y python=3.9.11 flask=2.2.2 flask-restful=0.3.9
    jinja2=3.1.2  werkzeug=2.3 conda-forge::mysql-connector-python

# 3. Remove miniconda and installation scripts
rm -rf ./Miniconda3-latest-Linux-x86_64.sh $script_dir/miniconda
```

Criar o arquivo **`auth_api.py`** contendo o código-fonte abaixo:

```
#!/usr/bin/python3

import sys
import os
import mysql.connector

from flask import Flask, request
from flask_restful import Resource, Api
sys.path.insert(0, os.path.dirname(__file__))
cert = '/etc/flask.crt'
cert_key = '/etc/flask.key'
app = Flask(__name__)
mydb = mysql.connector.connect(
  host="127.0.0.1",
  user="larc",
  password="larc123",
  database="auth_db",
  port=3306)


# Para tratar os requests de posts podemos pegar os valores
# dos campos do formulario usando o atributo do POST
# Os valores apos a submissao nao serao mostrados na URL
@app.route('/login', methods=['POST'])
def handle_post():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        print(username, password)
        mycursor = mydb.cursor()
        mycursor.execute("SELECT * FROM users \
        WHERE login='" + username + \
        "' AND password='" + password + "'")
        myresult = mycursor.fetchall()
        mydb.commit()
        print(len(myresult))
        mycursor.close()
        if len(myresult) > 0:
            return '<h1>Authenticado!!</h1>\n'
        else:
            return '<h1>Credenciais invalidas!</h1>\n'
    else:
        return 'Indice da pagina\n'

if __name__ == '__main__':
    app.debug = False
```

```
47      ssl_context = (cert, cert_key)
48      app.run(host='0.0.0.0', port=4996, threaded=True,
49      ssl_context=ssl_context)
```

Na linha 13 observa-se a criação de uma conexão com o banco de dados MySQL (previamente configurado). Nas linhas 29-37, é realizada uma busca no banco de dados pelo usuário e senha recebidos por meio da requisição nas linhas 27 e 28. O resultado desta atividade é verificado na linha 38 e retornada a resposta apropriada, na linha 39 ou na linha 41.

iv) A seguir, alterar o arquivo **`flask.yaml`**, no mesmo diretório e adicionar uma referência ao arquivo **`../auth_api.py`** na linha 18. O arquivo **`flask.yaml`** contém as instruções do que será copiado para execução dentro do enclave. A linha 18 instrui o criador do enclave para copiar o novo arquivo criado para a área de arquivos executáveis do sistema de arquivos do enclave.

A seguir está o resultado esperado após a edição:

```
1  includes:
2    - base.yaml
3  targets:
4    - target: /usr/bin
5      createlinks:
6        - src: /opt/python-occlum/bin/python3
7          linkname: python3
8    # python packages
9    - target: /opt
10     copy:
11       - dirs:
12           - ../python-occlum
13   # below are python code and data
14   - target: /bin
15     copy:
16       - files:
17         - ../rest_api.py
18         - ../auth_api.py
19   # Flask server key/cert
20   - target: /etc
21     copy:
22       - files:
23         - ../flask.crt
24         - ../flask.key
```

v) Editar o arquivo **`run_flask_on_occlum.sh`** e substituir nas linhas 9 e 10, **`rest_api.py`** por **`auth_api.py`**. Desta forma, o arquivo criado será executado dentro do enclave criado. O arquivo **`run_flask_on_occlum.sh`** deve ficar igual a:

```
1  #!/bin/bash
2  set -e
3
4  BLUE='\033[1;34m'
```

```
NC='\033[0m'

# Run the python demo
cd occlum_instance
echo -e "${BLUE}occlum run /bin/auth_api.py${NC}"
occlum run /bin/auth_api.py
```

vi) *Implementação e execução da experiência*

Executar a imagem do Docker do Occlum por meio do comando:

```
sudo docker run -it --network host \
    --device /dev/sgx_enclave:/dev/sgx/enclave \
    --device /dev/sgx_provision:/dev/sgx/provision \
    --name occlum \
    -v /home/larc/occlum-fs/occlum/demos:/root/demos \
    occlum/occlum:latest-ubuntu20.04
```

No terminal dentro do contêiner, acessar **`/root/demos/python/flask/`** e executar o comando:

```
./install_python_with_conda.sh
```

Se o resultado do comando for bem sucedido é esperado o texto abaixo:

```
added / updated specs:
    - conda-forge::mysql-connector-python
    - flask-restful=0.3.9
    - flask=2.2.2
```

Ainda no mesmo diretório, executar:

```
./gen-cert.sh
```

Como resultado do comando, os arquivos **`flask.crt`** (certificado) e **`flask.key`** (chave privada) devem ser gerados.

Executar o comando:

```
./build_occlum_instance.sh
```

A listagem abaixo ilustra o resultado esperado:

```
Succeed.
Built the Occlum image and enclave successfully
```

Finalmente, executar o programa no Occlum, através do comando abaixo:

```
$ ./run_flask_on_occlum.sh
```

Como resultado, é esperado que o servidor HTTP, escrito na linguagem **`Python`** seja executado e no terminal apareça o seguinte:

```
occlum run /bin/auth_api.py
 * Serving Flask app 'auth_api'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a
    production deployment.
 * Running on all addresses (0.0.0.0)
 * Running on https://127.0.0.1:4996
 * Running on https://172.20.4.158:4996
Press CTRL+C to quit
```

É possível que os endereços de rede indicados sejam diferentes, pois depende da máquina na qual o servidor esteja sendo executado. Mas a porta utilizada sempre será a 4996.

vii) **Teste da autenticação.** A partir deste momento, o servidor de autenticação se encontra em execução e conectado ao banco de dados como esperado. Contudo, é necessário realizar um acesso ao servidor de autenticação simulando uma requisição do usuário mencionado anteriormente (ilustrado na Figura 3.10). Para realizar esse teste de acesso, primeiramente é necessário abrir outro terminal e copiar o certificado gerado de modo a permitir o acesso ao servidor de autenticação. O certificado (**`flask.crt`**) pode ser copiado por meio do comando:

```
cp /home/larc/occlum-fs/occlum/demos/python/flask/flask.crt .
```

Uma vez que o certificado para acesso ao servidor de autenticação se encontra disponível, é possível testar o acesso fazendo uma chamada ao servidor. Para realizar esse acesso, é utilizado o comando 'curl'. A seguir está visível a execução deste comando, com o objetivo de realizar a autenticação do usuário 'user1' utilizando a senha 'passwd1':

```
curl -k --cacert flask.crt -X \
    POST https://127.0.0.1:4996/login -d \
    "username=user1&password=passwd1"
```

O resultado da execução deste comando é mostrado a seguir indicando que a autenticação foi bem sucedida:

```
<h1>Authenticado!!!</h1>
```

Como conclusão deste experimento, é evidenciado que a conexão com o banco de dados MySQL existe e que, o usuário e a respectiva senha estão corretos. Também pode-se perceber que a aplicação de autenticação, executada dentro do enclave, acessou um banco de dados fora deste enclave. O banco de dados está criptografado e portanto seguro, independente de onde ele estiver sendo executado, que pode ser em outro enclave ou no ambiente inseguro.

3.8. Protótipo: usando TEE para plataforma de streaming

Nessa seção, discutimos uma aplicação de TEE: autenticação de usuários em um ambiente em nuvem não confiável. Construímos um protótipo com o protocolo OAuth2, e avaliamos a sua segurança.

Para motivar a discussão, imagine o seguinte. Um usuário deseja assistir a um filme disponível em uma plataforma de *streaming* de vídeos fictícia intitulada LarcFlix. Para isso, o usuário deve se autenticar em uma plataforma de pagamento ("Banco Vermelho"), através de algum protocolo, como OAuth. Somente com a autenticação bem sucedida a plataforma LarcFlix libera a visualização do filme ao usuário. Porém, o Banco Vermelho não confia nos provedores de nuvem: assim, todo o armazenamento de dados do usuário e a autenticação deve ser feita através de tecnologias de ambientes de computação segura.

Para implementar esses ambientes, foram idealizados um servidor de autenticação OAuth2, para fazer o papel do Banco Vermelho, e uma base de dados com as informações de autenticação de clientes do banco. Adicionalmente foi utilizado um servidor de vídeo (LarcFlix) para realizar a autenticação de seus usuários junto ao servidor de autenticação do Banco Vermelho. Contudo, o servidor de vídeos não se encontra no ambiente de computação segura, pois tem apenas como objetivo usar a autenticação OAuth2.

Para esse experimento, a solução de ambiente de computação segura selecionada é a Intel SGX juntamente com o *framework* Occlum (vide seção 3.5.3).

3.8.1. Infraestrutura do protótipo

A montagem foi elaborada em um provedor de nuvem com duas máquinas virtuais. Também foram utilizadas uma máquina adicional para executar o servidor de vídeos e outra para simular um usuário. Resumidamente, foram utilizados quatro ambientes de computação:

- *Servidor de banco de dados*: máquina virtual não confiável na nuvem para abrigar o servidor de base de dados com as informações de autenticação de usuários, mais especificamente os logins e senhas dos usuários Para a implementação foi utilizado um servidor MySQL abrigado dentro de um enclave SGX por meio do *framework* Occlum.

- *Servidor de autenticação*: máquina virtual não confiável na nuvem para abrigar o servidor de autenticação OAuth2 do Banco Vermelho. Para sua implementação foi utilizado o *software* Casdoor [Casdoor 2024], também abrigado em um enclave SGX por meio do *framework* Occlum. O servidor OAuth2 realiza conexões a base de dados MySQL para obtenção dos dados sensíveis dos clientes do Banco Vermelho (logins e senhas) de modo a autenticá-los. Esse servidor é abrigado em um enclave por manipular dados sensíveis de usuários em um computador não confiável.

- *Servidor de vídeos LarcFlix*: máquina com *software* servidor de vídeos. A implementação utilizada foi uma criação própria dos autores, mimetizando uma plata-

forma de *streaming* de vídeos. Para fins de demonstração, ela está presente em um ambiente confiável.

- *Cliente*: máquina para simulação de um acesso do cliente do Banco Vermelho que deseja assistir a um filme na plataforma LarcFlix. No caso desta PoC, foi utilizado um computador *desktop* com navegador Web.

A Figura 3.12 ilustra os quatro ambientes computacionais da infraestrutura desse protótipo e as conexões de rede estabelecidas entre os ambientes.

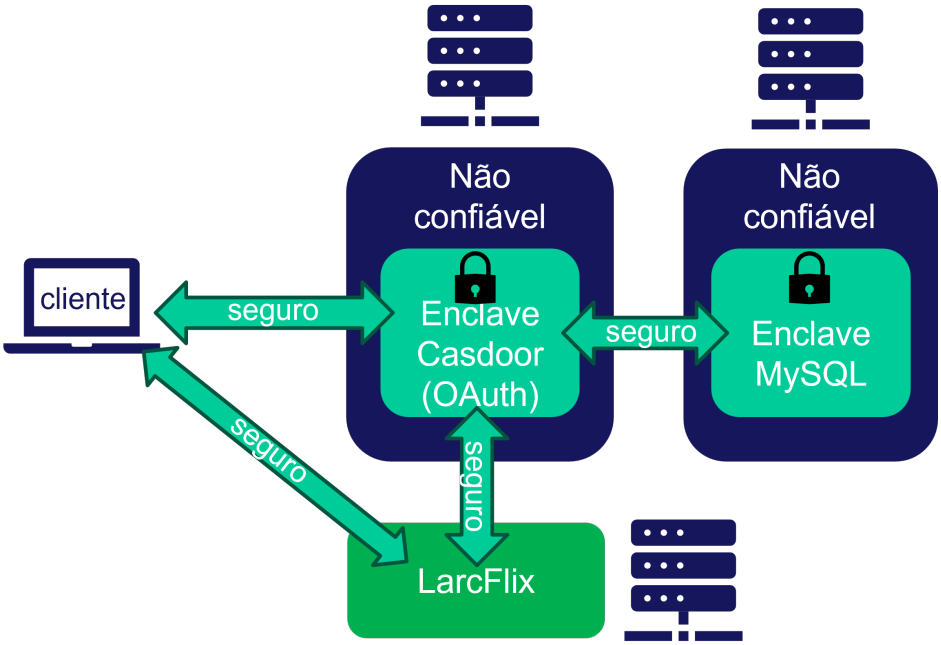

Figura 3.12. Infraestrutura utilizada para o protótipo: ambientes computacionais e conexões entre eles.

Uma vez que no cenário proposto o Banco Vermelho não confia no provedor de nuvem, há alguns requisitos importantes presentes na configuração desta PoC: os arquivos de dados do MySQL utilizado se encontram cifrados e são utilizado nesta forma - se trata de um *feature* oferecido pelo *framework* Occlum; a conexão estabelecida entre a base de dados MySQL e o servidor de autenticação OAuth2 do Banco Vermelho (Casdoor) é cifrada com SSL.

De fato, toda a comunicação de dados (cliente - servidor OAuth; LarcFlix - servidor OAuth) deve também ser cifrada de modo que a nuvem não tenha acesso aos dados e informações trafegadas, preservando a confidencialidade das comunicações. É importante ressaltar que o canal SSL entre o servidor OAuth do Banco Vermelho e o servidor de base de dados MySQL é fechado dentro dos respectivos ambientes de computação segura, isto é, as máquinas não confiáveis apenas manipulam dados de comunicação cifrados.

O mesmo pode ser afirmado do canal SSL estabelecido entre a máquina do cliente e o servidor OAuth do Banco Vermelho: a máquina não confiável não consegue decifrar

o tráfego. Para o canal SSL estabelecido entre o servidor LarcFlix e o servidor OAuth do Banco Vermelho, o mesmo se aplica.

O principal objeto de avaliação do experimento é a integridade e confidencialidade das informações dos clientes do Banco Vermelho. Ou seja, se os dados permaneceram íntegros e confidenciais no armazenamento e na memória dos ambientes que hospedam os servidores OAuth do Banco Vermelho e o MySQL. Um elemento de avaliação secundário é com relação a estabilidade oferecida pelo *framework* Occlum.

3.8.2. Fluxo de autenticação

Como situação inicial já se tem iniciados os servidores do Banco Vermelho e o MySQL utilizado. O servidor do Banco Vermelho já se encontra conectado ao MySQL com as informações de clientes. Igualmente, o servidor de vídeos LarcFlix já está no ar. Em um dado momento, o usuário acessa a página Web de filmes da plataforma LarcFlix via protocolo *https*. O usuário seleciona um filme deseja e clica nele para assistir.

Visto que o usuário não está autenticado, ele é redirecionado ao autenticador OAuth2 do Banco Vermelho para realizar sua autenticação e, por consequência, a aquisição do filme. Logo que o navegador do usuário inicia a conexão com o servidor OAuth2 do Banco Vermelho, ele recebe o certificado do banco para validar sua origem.

Após a validação do certificado do banco, o navegador do usuário recebe a página de autenticação de clientes do banco. Ele então preenche os dados de autenticação, informando seu login e senha. Em seguida, o servidor OAuth2 realiza a busca pelos dados do usuário no MySQL para conseguir autenticar o usuário.

Caso a autenticação seja bem sucedida, o navegador do usuário é redirecionado ao servidor de vídeos LarcFlix com um código *authentication code* do protocolo OAuth2. O servidor LarcFlix valida o código junto ao servidor do Banco Vermelho e obtém um *access token* que representa a compra com sucesso. Por fim, a plataforma LarcFlix libera o vídeo ao usuário.

3.8.3. Resultados obtidos

Durante a realização do protótipo, houve a monitoração dos processos para averiguação de alguns itens referentes a requisitos não funcionais. Dentre eles: a estabilidade do sistema e a segurança.

O protótipo se comportou de forma estável durante a realização das autenticações com diversos usuários e ficou no ar durante uma semana sem quedas. Uma observação dos autores é com relação ao consumo de memória RAM: ao utilizar o *framework* Occlum, há um grande consumo de memória que se acredita ser devido ao carregamento de sua arquitetura baseada em LibOS.

Sobre segurança, que é o foco principal do protótipo, foi observado de que as conexões estabelecidas entre os ambientes computacionais estavam protegidos com SSL de ponta a ponta nas situações indicadas na seção 3.8.1. Também foram realizadas tentativas de *dump* da memória RAM em arquivos para simular o ataque de alguém com privilégio de *root* nas máquinas. Os arquivos obtidos não apresentaram dados sensíveis, indicando o funcionamento esperado da tecnologia SGX. Ou seja, os dados sensíveis ficaram ocultos.

Finalmente, ao observar os resultados obtidos, é possível afirmar que o experimento produziu os resultados esperados. Como possibilidades de trabalhos futuros, técnicas mais avançadas de ataques podem ser experimentadas, inspirados na ferramenta CacheZoom [Moghimi et al. 2017], por exemplo.

3.9. Conclusões

Esse minicurso apresentou conceitos de Ambientes de Execução Confiável (TEE), uma tecnologia que permite aos usuários executar códigos de maneira íntegra e confidencial mesmo diante de ambientes comprometidos (e.g., sistema operacional malicioso ou provedores de nuvem mal intencionados). Para isso, TEEs fornecem um ambiente de execução isolado, que impede o acesso de qualquer software não-autorizado. Um exemplo de TEE é o Intel SGX, um conjunto de instruções que fornece isolamento através de *enclaves*, e que impede por hardware o acesso e a modificação de regiões de memória em uso dos enclaves. Desenvolvedores devem projetar cuidadosamente as aplicações de modo a não torna-las vulneráveis a ataques, como ataques de canal lateral explorando a tabela de páginas ou o comportamento do cache. Este trabalho também citou outros exemplos de TEEs, como ARM trustzone e AMD SEV.

Em seguida, o minicurso foi para um aspecto mais prático, apresentando algumas ferramentas para facilitar o desenvolvimento de aplicações seguras. Foram apresentadas duas SDKs, Intel SGX SDK e Open Enclave SDK, que permitem o desenvolvimento de aplicações usando enclaves; e foi apresentada uma LibOS, Occlum, que permite subir uma aplicação inteira dentro de um enclave sem exigir modificações no código-fonte.

Por fim, o minicurso apresentou alguns experimentos usando Intel SGX. Ele apresentou primeiramente como informações confidenciais podem ser extraídas pelo administrador através de um dump de memória, e então explicou como isso esse ataque é impedido quando o serviço é implantado em um enclave. Em seguida, o trabalho ilustrou algumas implementações usando Open Enclave SDK e Occlum. Por fim, esse trabalho apresentou um protótipo que mostra a aplicação de TEEs para autenticação de uma plataforma de vídeo.

A. Ativando SGX

Em algumas máquinas mais antigas (principalmente desktops), o Intel SGX não está habilitado por padrão. Assim, essa seção explica como habilita-lo.

Primeiramente, é preciso verificar se o SGX está disponível na máquina para a realização do experimento. Para isso, é necessário visualizar se aparecem dois *devices* do Linux sob a pasta 'dev'. Então, deve-se executar o seguinte comando para a verificação:

```
sudo ls -l /def/sgx*
```

Como resultado esperado, devem aparecer dois *devices*. Caso não apareçam, mas a funcionalidade SGX está acionada na BIOS, é necessário habilitar os *devices* usando um programa da Intel disponível no GitHub[7] (https://github.com/intel/sgx-software-enable). Para tal, executar os seguinte comandos:

```
git clone https://github.com/interl/sgx-software-enable
cd sgx-software-enable/
make
sudo ./sgx_enable
./sgx_enable -s
```

Após a execução, o programa indica o estado atual do SGX (se está habilitado ou não). Em seguida, é necessário verificar a capacidade SGX instalada no processador. Pode ser verificada por meio dos passos abaixo, que utilizam um programa disponível no GitHub (https://github.com/ayeks/SGX-hardware):

```
git clone https://github.com/ayeks/SGX-hardware/
sudo apt-get install libcap-dev
gcc -Wl,--no-as-needed -Wall -Wextra -Wpedantic -masm=intel \
    -o test-sgx -lcap cpuid.c rdmsr.c xsave.c vdso.c test-sgx
sudo ./test-sgx
```

O comando acima produz um relatório com diversas informações do SGX presente na máquina. Para que o experimento corrente seja bem sucedido, é fundamental que o FLC esteja ativado. Se no relatório produzido aparecerem linhas com o conteúdo:

```
Supports SGX
SGX Launch Configuration (SGX_LC): 1
```

Se além dessas também aparecerem linhas com um desses conteúdo 'SGX1 leaf instructions (SGX1): 1' ou 'SGX1 leaf instructions (SGX2): 1', significa que o computador é capaz de executar o experimento proposto. Ou seja, está com SGX e FLC ativados.

[7]O repositório encontra-se arquivado, visto que é focado para processadores mais antigos. A Intel removeu o SGX de processadores mais novos (a partir da 11º geração), focando em processadores de servidores (Intel Xeon).

Referências

[Aaraj et al. 2009] Aaraj, N., Raghunathan, A., and Jha, N. K. (2009). Analysis and design of a hardware/software trusted platform module for embedded systems. *ACM Trans. Embed. Comput. Syst.*, 8(1).

[Anati et al. 2013] Anati, I., Gueron, S., Johnson, S., and Scarlata, V. (2013). Innovative technology for cpu based attestation and sealing. *Proceedings on hardware and architectural support for security and privacy, (HASP)*, 13.

[Architecture 2024] Architecture, C. C. (2024). Cca. Acesso em: 12 Set 2024.

[Arfaoui et al. 2014] Arfaoui, G., Gharout, S., and Traoré, J. (2014). Trusted execution environments: A look under the hood. In *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pages 259–266.

[ARM 2024] ARM (2024). TrustZone for Cortex-A. Disponível em: https://www.arm.com/technologies/trustzone-for-cortex-a. Acesso em: 07 ago. 2024.

[Arthur and Challener 2015] Arthur, W. and Challener, D. (2015). *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Apress, USA, 1st edition.

[Bursell 2020] Bursell, M. (2020). *Systems and Trust*, chapter 8, pages 185–209. John Wiley & Sons, Ltd.

[Bursell 2021] Bursell, M. (2021). *Trust in computer systems and the cloud*. John Wiley & Sons, NJ, USA. ISBN: 978-1119549246.

[Casdoor 2024] Casdoor (2024). An open-source UI-first Identity and Access Management (IAM) / Single-Sign-On (SSO) platform. Acesso em: 07 ago. 2024.

[Docker, Inc. 2024] Docker, Inc. (2024). Docker - Accelerate how you build, share, and run applications. Acesso em: 07 ago. 2024.

[Fei et al. 2021] Fei, S., Yan, Z., Ding, W., and Xie, H. (2021). Security vulnerabilities of sgx and countermeasures: A survey. *ACM Computing Surveys (CSUR)*, 54(6):1–36. DOI: https://doi.org/10.1145/3456631.

[Geppert et al. 2022] Geppert, T., Deml, S., Sturzenegger, D., and Ebert, N. (2022). Trusted execution environments: Applications and organizational challenges. *Frontiers in Computer Science*, 4:930741.

[Ghaniyoun et al. 2023] Ghaniyoun, M., Barber, K., Xiao, Y., Zhang, Y., and Teodorescu, R. (2023). Teesec: Pre-silicon vulnerability discovery for trusted execution environments. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA. Association for Computing Machinery.

[GitHub, Inc. 2024] GitHub, Inc. (2024). GitHub. Acesso em: 09 ago 2024.

[Gremaud et al. 2017] Gremaud, P., Durand, A., and Pasquier, J. (2017). A secure, privacy-preserving iot middleware using intel sgx. In *Proceedings of the Seventh International Conference on the Internet of Things*, IoT '17, New York, NY, USA. Association for Computing Machinery.

[Huang et al. 2024] Huang, H.-J., Zhang, F., Yan, S., Wei, T., and hao He, Z. (2024). Sok: A comparison study of arm trustzone and cca. In *2024 International Symposium on Secure and Private Execution Environment Design*.

[Intel 2024] Intel (2024). Intel Software Guard Extensions SDK for Linux OS - Application Design Considerations. Disponível em: https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/linux-overview.html, Acesso em 11 out 2024.

[Intel Corporation 2021] Intel Corporation (2021). Life Cycle of an SGX Enclave. URL: https://www.intel.com/content/dam/develop/external/us/en/documents/intelsgxenclavelifecycle.pdf. Accessado em 2024-09-12.

[Intel Corporation 2023] Intel Corporation (2023). Intel® Software Guard Extensions (Intel® SGX) Data Center Attestation Primitives: ECDSA Quote Library API. URL: https://download.01.org/intel-sgx/latest/dcap-latest/linux/docs/Intel_SGX_ECDSA_QuoteLibReference_DCAP_API.pdf. Accessado em 2024-09-12.

[Jauernig et al. 2020] Jauernig, P., Sadeghi, A.-R., and Stapf, E. (2020). Trusted execution environments: Properties, applications, and challenges. *IEEE Security & Privacy*, 18(2):56–60.

[Kaissis et al. 2020] Kaissis, G. A., Makowski, M. R., Rückert, D., and Braren, R. F. (2020). Secure, privacy-preserving and federated machine learning in medical imaging. *Nature Machine Intelligence*, 2(6):305–311.

[Kanno 2005] Kanno, Y. (2005). An introduction to information security evaluation. *Journal of Information Processing and Management*, 48(6):320–332. DOI: https://doi.org/10.1241/johokanri.48.320.

[Kaplan 2017] Kaplan, D. (2017). Protecting vm register state with sev-es. *White paper*, page 13.

[Kaplan et al. 2016] Kaplan, D., Powell, J., and Woller, T. (2016). Amd memory encryption. *White paper*, 13.

[Kocher et al. 2019] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. (2019). Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (SP)*, pages 1–19, New York, NY, USA. IEEE. DOI: https://10.1109/SP.2019.00002.

[Li et al. 2021] Li, M., Zhang, Y., and Lin, Z. (2021). Crossline: Breaking "security-by-crash"based memory isolation in amd sev. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2937–2950, New York, NY, USA. Association for Computing Machinery.

[Li et al. 2023] Li, X., Zhao, B., Yang, G., Xiang, T., Weng, J., and Deng, R. H. (2023). A survey of secure computation using trusted execution environments.

[Lind et al. 2017] Lind, J., Priebe, C., Muthukumaran, D., O'Keeffe, D., Aublin, P.-L., Kelbert, F., Reiher, T., Goltzsche, D., Eyers, D., Kapitza, R., et al. (2017). Glamdring: Automatic application partitioning for intel {SGX}. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 285–298.

[Mofrad et al. 2018] Mofrad, S., Zhang, F., Lu, S., and Shi, W. (2018). A comparison study of intel sgx and amd memory encryption technology. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '18, New York, NY, USA. Association for Computing Machinery.

[Moghimi et al. 2017] Moghimi, A., Irazoqui, G., and Eisenbarth, T. (2017). CacheZoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES'17)*, pages 69–90, Cham. Springer International Publishing. DOI: https://doi.org/10.1007/978-3-319-66787-4.

[MongoDB, Inc. 2024] MongoDB, Inc. (2024). MongoDB. Acesso em: 09 ago 2024.

[Muñoz et al. 2023] Muñoz, A., Ríos, R., Román, R., and López, J. (2023). A survey on the (in)security of trusted execution environments. *Computers & Security*, 129:103180. DOI: https://doi.org/10.1016/j.cose.2023.103180.

[Ning et al. 2023] Ning, Z., Wang, C., Chen, Y., Zhang, F., and Cao, J. (2023). Revisiting arm debugging features: Nailgun and its defense. *IEEE Transactions on Dependable and Secure Computing*, 20(1):574–589.

[Occlum 2024] Occlum (2024). Occlum - A library OS empowering everyone to run every application in secure enclaves. Disponível em: https://occlum.io/. Acesso em: 07 ago. 2024.

[OESDK 2024] OESDK (2024). Open Enclave SDK. Disponível em: https://openenclave.io/sdk/. Acesso em: 07 ago. 2024.

[OP-TEE 2024] OP-TEE (2024). About OP-TEE. Disponível em: https://optee.readthedocs.io/en/latest/general/about.html. Acesso em: 07 ago. 2024.

[Paju et al. 2023] Paju, A., Javed, M. O., Nurmi, J., Savimäki, J., McGillion, B., and Brumley, B. B. (2023). SoK: A Systematic Review of TEE Usage for Developing Trusted Applications. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, pages 1–15, New York, NY, USA. Association for Computing Machinery. DOI: https://doi.org/10.1145/3600160.36001691.

[Rose et al. 2020] Rose, S., Borchert, O., Mitchell, S., and Connelly, S. (2020). Zero Trust Architecture. *NIST special publication*, 800:207. DOI: https://doi.org/10.6028/NIST.SP.800-207.

[Rust 2024] Rust (2024). Rust. Disponível em: https://www.rust-lang.org/. Acesso em: 11 ago. 2024.

[Sabt et al. 2015] Sabt, M., Achemlal, M., and Bouabdallah, A. (2015). Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64.

[Sev-Snp 2020] Sev-Snp, A. (2020). Strengthening vm isolation with integrity protection and more. *White Paper, January*, 53:1450–1465.

[Shen et al. 2020] Shen, Y., Tian, H., Chen, Y., Chen, K., Wang, R., Xu, Y., Xia, Y., and Yan, S. (2020). Occlum: Secure and efficient multitasking inside a single enclave of Intel SGX. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 955–970, New York, NY, USA. Association for Computing Machinery. DOI: https://doi.org/10.1145/3373376.337846.

[Shih et al. 2016] Shih, M.-W., Kumar, M., Kim, T., and Gavrilovska, A. (2016). S-nfv: Securing nfv states by using sgx. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pages 45–48. DOI: https://doi.org/10.1145/2876019.2876032.

[Software Freedom Conservancy 2024] Software Freedom Conservancy (2024). Git. Acesso em: 09 ago 2024.

[Song et al. 2024] Song, Y., Zhao, W., Tian, Y., and Wang, B. (2024). Hsm: A hybrid storage method based on the heat of data and global disk space utilization. *IEEE Access*, 12:48630–48639.

[Syed et al. 2022] Syed, N. F., Shah, S. W., Shaghaghi, A., Anwar, A., Baig, Z., and Doss, R. (2022). Zero Trust Architecture (ZTA): A Comprehensive Survey. *IEEE Access*, 10:57143–57179. DOI: https://doi.org/10.1109/ACCESS.2022.3174679.

[Ubuntu Manpage Repository 2024] Ubuntu Manpage Repository (2024). gcore. Acesso em: 09 ago 2024.

[Van Bulck et al. 2018] Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. (2018). Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association. See also technical report Foreshadow-NG [Weisse et al. 2018].

[Vauclair 2011] Vauclair, M. (2011). *Secure Element*, pages 1115–1116. Springer US, Boston, MA.

[Weisse et al. 2018] Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T. F., and Yarom, Y. (2018). Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*. See also USENIX Security paper Foreshadow [Van Bulck et al. 2018].