Capítulo 2

Mecanismos de segurança baseados em *hardware*: uma introdução à arquitetura Intel SGX

Newton C. Will, Rafael C. R. Condé, Carlos A. Maziero
Programa de Pós-Graduação em Informática - Universidade Federal do Paraná (UFPR)

Abstract

Data confidentiality is becoming more and more important to the corporate and domestic computer users. In addition, it is extremely important to ensure security in the execution of applications that manipulate such data, which must have their confidentiality and integrity guaranteed. In this way, there are several solutions that aim to maintain the data confidentiality and integrity, as well as security in the execution of applications. This chapter presents a review of the most relevant hardware-based security mechanisms and their evolution, culminating with the presentation of the Intel SGX architecture.

Resumo

A confidencialidade dos dados está se mostrando cada vez mais importante para os usuários de computadores, seja em um ambiente corporativo ou mesmo em um ambiente doméstico. Além disso, é extremamente importante garantir a segurança na execução das aplicações que manipulam tais dados, cuja confidencialidade e integridade devem ser garantidas. Neste sentido, existem várias soluções que se propõem a manter a confidencialidade e integridade dos dados e também a segurança na execução de aplicações. Este capítulo visa apresentar uma revisão dos mecanismos de segurança baseados em hardware mais relevantes e sua evolução, culminando na apresentação da arquitetura Intel SGX.

2.1. Introdução

Atualmente a tecnologia está presente na vida de muitas pessoas, que entregam seus dados a dispositivos como computadores, *smartphones*, *tablets*, e até mesmo a serviços de armazenamento *online*. Os dados e informações presentes nesses dispositivos, ou meios de armazenamento, são dos mais variados tipos, passando desde fotos de família e senhas, aos mais diversos serviços, agendas pessoais, dados médicos, bancários e mesmo informações sobre o trabalho, que podem ser extremamente importantes para as empresas. Em virtude desta expansão no uso de dispositivos digitais, pessoas e empresas têm ficado cada vez mais dependentes de recursos computacionais e da Internet, para exercer desde atividades simples do dia a dia até complexas atividades industriais.

A McAfee Labs estima que, devido à convergência da Internet das Coisas (IoT), 200 bilhões de dispositivos estejam conectados à Internet em 2020. Ainda segundo a McAfee Labs, em 2006 as empresas enfrentavam, em média, 25 novas ameaças por dia, enquanto hoje 500.000 ameaças são vistas diariamente [Weafer 2016]. O FBI estima que crimes cibernéticos geram 67,2 bilhões de dólares de prejuízo por ano [FBI 2005]. Desta forma, não causa estranheza que os ataques a recursos computacionais estejam ficando cada vez mais sofisticados e difíceis de serem detectados e contidos.

Brechas de segurança em dados e aplicações são alvos constantes de ataques, que atingem até mesmo grande empresas, como o ocorrido com a Apple, que supostamente estaria sendo chantageada por uma equipe de *hackers* que teria conseguido acesso a 300 milhões de contas de *e-mail* de usuários [Cox 2017]. Ataques como este causam grandes prejuízos às empresas, podendo até levá-las à falência, em casos extremos.

Conforme descrito por [Hoekstra et al. 2013], as práticas para o desenvolvimento de aplicações seguras são padronizadas em muitas empresas, tendo o teste de segurança como parte chave na validação do *software*. Ainda assim a confidencialidade e integridade dos dados dependerá da correta utilização de outros *softwares* que estarão sendo utilizados no mesmo ambiente, alguns dos quais podem ter privilégios para inspecionar a memória ou outros meios de armazenamento no dispositivo. Além disso, cabe ao usuário final o bom senso no uso das aplicações, o qual deverá seguir as boas práticas e adotar uma política cautelosa para a execução de diversas aplicações, seja no ambiente de trabalho ou para uso pessoal.

Neste contexto, grandes esforços têm sido empenhados no sentido de garantir a segurança de dados e execução de aplicações sensíveis, seja através da exploração de tecnologias existentes, ou através da pesquisa e desenvolvimento de novas tecnologias. No intuito de expandir cada vez mais o nível de proteção, tecnologias baseadas na implementação em *hardware* têm ganho grande relevância.

Este capítulo apresenta uma revisão das principais tecnologias de segurança baseadas em *hardware*, focando principalmente na recentemente proposta arquitetura Intel *Software Guard Extensions* (SGX), descrevendo as suas funcionalidades e usos, incluindo teoria e prática. Este capítulo foi dividido em sete seções, com a primeira delas trazendo esta introdução.

A Seção 2.2 faz uma revisão das diversas técnicas e mecanismos de segurança que foram ou são utilizados ao longo dos anos para manter a segurança das aplicações e

confidencialidade dos dados, com o objetivo de apresentar uma evolução destas técnicas. São abordados desde os primeiros *hardwares* criptográficos, passando pela especificação do TPM (*Trusted Platform Module*), implementações como o AMD *Secure Processor* e a Intel *Trusted Execution Technology*, a arquitetura ARM *TrustZone*, que permite uma comunicação segura entre um processador ARM e seus demais dispositivos compatíveis, uma breve introdução à tecnologia Intel SGX e, por fim, um comparativo entre esses mecanismos.

Na Seção 2.3 é detalhada a arquitetura Intel *Software Guard Extensions* (SGX), que foi incluída na família de processadores *Skylake* no final de 2015, e permite que uma aplicação seja encapsulada dentro de um enclave, que é gerenciado pelo próprio processador.

Na Seção 2.4 são apresentados alguns trabalhos que já fazem uso da arquitetura SGX, seja em caráter de produção ou experimental; a Seção 2.5 lista as limitações da arquitetura Intel SGX e alguns tópicos de interesse em pesquisa, além de um resumo das principais vulnerabilidades conhecidas do SGX.

A Seção 2.6 apresenta os recursos disponíveis no *Software Development Kit* (SDK) do SGX para a construção de aplicações, trazendo, inclusive, exemplos práticos. Por fim, a Seção 2.7 traz as considerações finais acerca dos tópicos que são apresentados neste capítulo.

2.2. Mecanismos de Segurança Baseados em *Hardware*

Esta seção visa apresentar um histórico da evolução dos principais mecanismos de segurança baseados em *hardware* que, de alguma maneira, pavimentaram a criação da arquitetura Intel SGX.

2.2.1. Primeiros *Hardwares* Criptográficos

[Anderson et al. 2006] estabelece que um cripto-processador típico é um processador embarcado dedicado que executa um conjunto pré-definido de operações criptográficas usando chaves internas, protegidas do ambiente externo, que se comunicam com o computador principal. Os cripto-processadores devem ser resistentes a ataques físicos e por *software*. Operações criptográficas são naturalmente custosas, e a utilização de um *hardware* dedicado para esta tarefa contribui para reduzir os impactos negativos no desempenho. Processadores deste tipo são utilizados há muito tempo em aplicações específicas, como em sistemas bancários e aplicações militares. No entanto, com a crescente necessidade da proteção de dados sensíveis nas mais diversas aplicações, cripto-processadores têm ganho maior relevância e diferentes implementações têm sido empregadas.

Dentre estas implementações, [Bossuet et al. 2013] destaca os seguintes mecanismos:

- **GPP Customizado:** Utiliza processadores de propósito geral customizados para implementação de algoritmos criptográficos eficientes. Eles absorvem operações criptográficas específicas, como o *Data Encryption Standard* (DES) ou o *Advanced Encryption Standard* (AES). Estas soluções melhoram o desempenho, mas não a segurança, uma vez que as chaves criptográficas são armazenados em memória e

tratadas como dados ordinários da aplicação, podendo sofrer ataques por software. Outra dificuldade é a seleção do conjunto de instruções, visto que algoritmos de chave simétrica e assimétrica têm necessidades distintas.

- **Cripto-coprocessador:** Implementação em *hardware* customizado, altamente eficiente para funções criptográficas específicas. Eles podem conter um ou mais núcleos de processamento, mas não são programáveis e sim controlados, configurados ou parametrizados usando o processador a que estão acoplados. São usados para acelerar cálculos criptográficos. Seu uso é indicado para aplicações com necessidade de alta vazão.

- **Cripto-processador:** É um módulo de hardware que se difere do coprocessador por ser programável, com um conjunto de instruções dedicadas para obter eficiência em funções criptográficas. Cripto-processadores contam com uma ou mais unidades lógico-aritméticas (ULAs) projetadas especificamente para cálculos criptográficos; por isso, para realizar operações convencionais é necessário o uso de um processador de propósito geral. Esta arquitetura é mais enxuta que as duas anteriores, podendo ser considerada relativamente flexível, uma vez que as ULAs podem ser reconfiguradas. Outra grande diferença é que as chaves de criptografia são armazenadas internamente pelo cripto-processador, e não mais na memória principal, como nos casos anteriores. Apesar de executar somente cálculos criptográficos, cripto-processadores são autônomos e representam um ótimo custo-benefício entre desempenho e capacidade de processamento. São boas opções para inclusão em *Multi-Processor-System-on-a-Chip* (MP-SoC).

- ***Cripto array*:** É um acelerador criptográfico que, assim como os cripto-processadores, precisa ser acoplado a processadores de propósito geral. Esta arquitetura usa vetores de núcleos de processamento para realizar as operações criptográficas. Normalmente os núcleos são reconfiguráveis para obtenção de uma maior flexibilidade.

2.2.2. *Trusted Platform Module* (TPM)

O *Trusted Computing Group* [TCG 2008] é uma organização internacional de normas que conta com cerca de 140 empresas, sendo responsável pela especificação de uma série de padrões de segurança, tais como IPSEC (*Internet Protocol Security Protocol*), IKE (*Internet Key Exchange*), VPN (*Virtual Private Network*), PKI (*Public Key Infrastructure*), S/MIME *(Secure Multi-purpose Internet Mail Extensions*), SSL (*Secure Socket Layer*), SET (*Secure Electronic Transaction*), dentre outros [Amin et al. 2008]. Uma das frentes de trabalho atuais do TCG é a especificação dos padrões para o *Trusted Computing Module*.

TPM (*Trusted Platform Module*) é um *chipset* capaz de armazenar artefatos usados para autenticar uma plataforma computacional, o que inclui senhas, certificados e chaves criptográficas. Um TPM também pode ser utilizado para armazenar as medidas da plataforma[1], para garantir que ela continua confiável. As especificações destacam

[1]Medidas (*measurements*) são informações previamente colhidas da plataforma em uma situação sabidamente segura, como *hashes* criptográficos, que podem ser usadas posteriormente para atestar sua integridade.

ainda que os dois elementos fundamentais para o estabelecimento de um ambiente seguro são: a) garantir que a plataforma pode provar que é quem ela diz ser (autenticação); e b) provar a entidades externas que a plataforma é confiável e não foi adulterada (atestação) [TCG 2008]. O conceito de TPM é aplicável a outros equipamentos além de PCs, tais como *smartphohes* ou equipamentos de rede.

Conforme [TCG 2016], o TPM é um componente do sistema que tem um estado separado do sistema hospedeiro. A interação entre o sistema hospedeiro e o TPM é feita somente através de uma interface definida nas especificações do TPM. O TPM é construído em recursos físicos dedicados exclusivamente a ele. Normalmente são implementados em um único chip acoplado ao sistema (tipicamente um PC) e são compostos de processador, RAM, ROM e memória *flash*, e sua única interface é um barramento LPC. A Figura 2.1 mostra a composição básica de um TPM. O sistema hospedeiro não pode alterar valores diretamente na memória do TPM, isto só pode ser feito através do *buffer* de E/S da interface, dedicado para este fim.

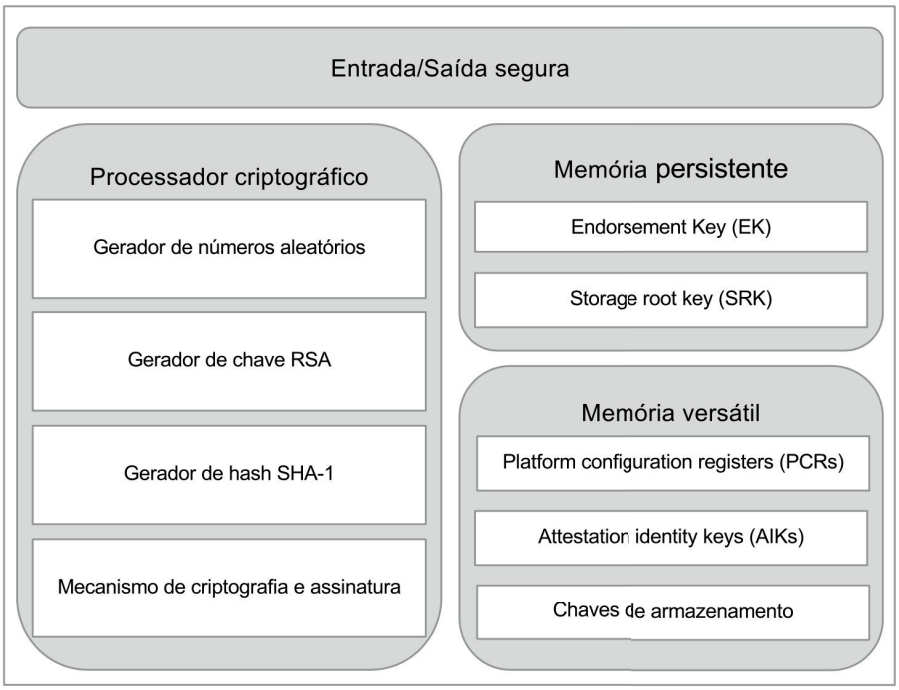

Figura 2.1. Arquitetura de um TPM (adaptado de [Vacca 2016]).

Ainda segundo as especificações [TCG 2016], outra implementação possível é executar o código seguro no processador do hospedeiro enquanto o mesmo se encontra em um modo de execução especial. Nesse caso, a memória do sistema é particionada pelo *hardware* de modo que a parte dedicada ao TPM fique inacessível, exceto quando o processador se encontra no modo especial. Quando o processador alterna entre os modos, ele sempre inicia a execução em pontos de entrada específicos. Existem diversos esquemas de implementação possíveis para isto, tais como *System Management Mode*, *TrustZone* e virtualização.

Dentro da construção de uma plataforma confiável existem alguns elementos cujas falhas não podem ser detectadas, e portanto precisam ser considerados confiáveis. Assim,

estes elementos devem ser projetados de modo a garantir que seu comportamento será sempre aquele desejado. [TCG 2016] define estes elementos como *Root of Trust*. Este conjunto de elementos deve ser o mínimo para prover as funcionalidades necessárias a uma plataforma confiável. Foram estabelecidos três *Roots of Trust* necessários ao TPM:

- ***Root of Trust for Measurement* (RTM):** Responsável por gerar informações relevantes à integridade da plataforma (suas medidas) e armazenar no RTS (próximo elemento). Normalmente este elemento é a CPU controlada pelo *Core Root of Trust for Measurement* (CRTM), que é o primeiro conjunto de instruções executadas quando uma nova cadeia de confiança é estabelecida, por exemplo, quando o sistema é reiniciado. O CRTM envia valores que indicam sua identidade ao RTS.
- ***Root of Trust for Storage* (RTS):** Usado para armazenar elementos como as medidas da plataforma, a *Endorsement Key* (EK)[2] e a *Storage Root Key* (SRK)[3]. A memória do TPM tem seu acesso bloqueado a qualquer entidade, exceto o próprio TPM. Se, no entanto, o elemento de memória contém informações não sensíveis, como no caso de algum *Platform Configuration Register* (PCR), o TPM permite a leitura.
- ***Root of Trust for Reporting* (RTR):** Responsável por reportar acerca do conteúdo do RTS para determinadas finalidades. Normalmente o *report* é um resumo criptográfico assinado digitalmente do conteúdo de valores selecionados do TPM. A interação entre o RTR e o RTS é crítica, sua implementação deve prevenir todas as formas de ataques físicos ou por *software* e prover os resumos criptográficos com precisão.

Estes três *Roots of Trust* usam certificação e atestação para fornecer evidências da confiabilidade da informação. Por fim, a plataforma mantém o *log* das suas mudanças de estado internas, para validação posterior.

2.2.3. Intel *Trusted Execution Technology*

Intel *Trusted Execution Technology* (Intel TXT) é uma tecnologia da Intel que utiliza componentes de *hardware* para criar ambientes de execução seguros contra ameaças físicas e virtuais, de modo a complementar proteções em tempo de execução, como *softwares* de anti-virus.

Conforme [Greene 2012], o Intel TXT necessita de alguns componentes fundamentais: extensões integradas ao Intel Xeon e *chipset* Intel, módulos de código autenticados (*Authenticated Code Modules* - ACMs), ferramentas de LCP (*Launch Control Policy*), um módulo TPM, BIOS e hipervisor ou sistema operacional habilitado para o Intel TXT. Se qualquer destes componentes faltar ou falhar, a plataforma irá ser carregada no seu estado não-confiável.

A execução segura é obtida com inicializações verificadas através de um ambiente de inicialização medido (*Measured Launch Environment* - MLE) que permite que os elementos críticos da inicialização possam ser confrontados contra as medições destes elementos vindas de uma fonte confiável. A medição consiste na criação de um identificador

[2] A EK é criada durante a fabricação e não pode ser alterada; ela é usada no processo de autenticação.

[3] A SRK é usada para armazenamento cifrado, sendo gerada em tempo de execução.

criptográfico único para cada componente aprovado na inicialização. O provisionamento destas medições confiáveis é feito através de um microcódigo autenticado (ACM) embarcado; uma vez realizadas as medições, elas são gravadas e travadas dentro do TPM. A Intel provê, ainda, um mecanismo baseado em *hardware* para bloquear a inicialização de código que não confere com aqueles previamente aprovados.

Existem dois métodos de estabelecimento de confiança através de medição (RTM): *estático* (S-RTM) e *dinâmico* (D-RTM). O S-RTM começa a medição em um evento de *reset* da plataforma com a medição da BIOS, passando pela medição do hipervisor (ou sistema operacional) e seus componentes. As medições realizadas são confrontadas com aquelas armazenadas no TPM. Se as medições forem equivalentes àquelas consideradas seguras, o sistema é carregado com o indicativo de confiável. Caso contrário, o sistema Intel fornecerá um indicativo de inicialização não-confiável. Apesar de ser mais simples, a S-RTM resulta em uma TCB (*Trusted Computing Base*[4]) grande e difícil de gerenciar. Uma vez estabelecida a confiança, qualquer mudança ou atualização da plataforma ocasionará na necessidade de migração ou atualização dos segredos. Já no D-RTM, as propriedades de confiança dos componentes podem ser ignoradas até a chamada de um evento seguro. Os componentes anteriores ao evento seguro serão excluídos da TCB e não poderão ser executados depois que a confiança do sistema for estabelecida. Isto resultará em uma TCB menor, o que é desejável.

Além da inicialização verificada, o Intel TXT dispõe de um mecanismo de políticas para a criação e implementação de listas de códigos executaveis aprovados ou sabidamente confiáveis. Este mecanismo é chamado de *Launch Control Policy* (LCP). Um mecanismo de proteção de segredos também é implementado através de métodos auxiliados por *hardware*, que removem dados residuais em reinicializações impróprias (ataques por *reset*). Por fim, o Intel TXT também tem a habilidade de produzir credenciais com as medições da plataforma para realizar atestação para usuários, ou sistemas locais ou remotos, de forma a completar o processo de verificação de confiança e suportar atividades de conformidade e auditoria.

Quando utilizado em ambiente virtualizado, o Intel TXT também faz uso extensivo da Tecnologia de Virtualização Intel (Intel VT) para prover proteção contra DMAs não autorizadas e garantir o isolamento de dados no sistema.

2.2.4. ARM *TrustZone*

A abordagem da arquitetura ARM para atingir as metas propostas pelo TCG também parte do conceito de uma plataforma confiável, a ARM *TrustZone*, que é uma arquitetura de *hardware* que estende o quesito de segurança a todo o *design* do sistema, permitindo que qualquer parte do sistema seja protegida. A tecnologia *TrustZone*, que está disponível nos controladores Cortex-A, Cortex-M23 e Cortex-M33, fornece uma infraestrutura básica que permite aos projetistas de SoCs (*System-on-Chip*) escolher uma gama de componentes que podem auxiliar em funções específicas dentro de um ambiente seguro. O objetivo

[4]Conjunto de todos os componentes de *hardware*, *firmware* e/ou *software* que são críticos para a segurança do sistema, no sentido de que os erros ou vulnerabilidades que ocorrem dentro do TCB podem prejudicar as propriedades de segurança do sistema inteiro. Em contrapartida, partes de um sistema fora do TCB não podem se comportar de forma a que obtenham mais privilégios do que os concedidos de acordo com a política de segurança.

principal da arquitetura é habilitar a construção de um ambiente programável que assegure a confidencialidade e integridade de quase todos os ativos a serem protegidos de ataques específicos, podendo ser utilizada para construir um conjunto de soluções de segurança que não são possíveis com os métodos tradicionais [ARM 2009].

Com a arquitetura ARM *TrustZone* o sistema pode ser isolado em dois estados lógicos: um estado seguro e um estado inseguro (Figura 2.2). Estes estados também são sinalizados para todos os dispositivos periféricos, através do barramento do sistema, permitindo-lhes tomar decisões de controle de acesso com base no estado atual do sistema. O mecanismos responsável pela troca de contexto entre os dois estados é chamado de *monitor*.

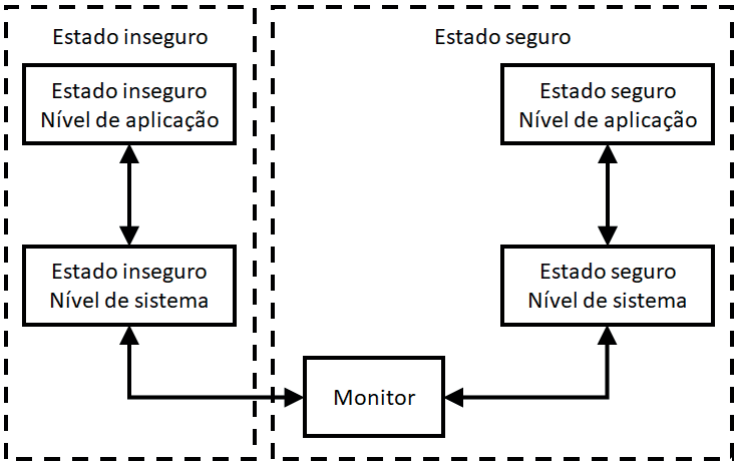

Figura 2.2. Modos de execução na arquitetura ARM *TrustZone* (adaptado de [ARM 2009]).

Quando uma aplicação está sendo executada em um estado seguro, ela pode isolar partes da memória para seu próprio uso, impedindo que aplicações sendo executadas em um estado inseguro acessem esses locais. Isto é garantido pelo controlador de memória que utiliza as premissas da arquitetura *TrustZone*, que fornece controle de acesso para regiões de memória com base no estado atual do sistema através do *TrustZone Address Space Controller*. Este particionamento de memória pode ser definido de forma fixa, ou programável em tempo de execução.

Uma aplicação em estado seguro também pode forçar que certas interrupções ou exceções sejam capturadas somente em um estado seguro, sendo que este controle também fica a cargo do controlador de interrupções. Além disso, o sistema também pode bloquear o acesso a determinados dispositivos para aplicações que não estejam sendo executadas em um estado seguro, garantindo a exclusividade de uso desses dispositivos somente a aplicações seguras [Lesjak et al. 2015].

Os sistemas que suportam *TrustZone* vêm, ainda, com *hardware* complementar para suporte a chaves seguras não-voláteis, *Real Time Clock* seguro e aceleradores criptográficos.

Para a consolidação de um ambiente seguro, um sistema operacional e um mecanismo de *boot* seguros devem ser desenvolvidos, utilizando os recursos do *TrustZone*. A combinação do isolamento por *hardware TrustZone*, *boot* seguro e sistema operacional seguro constituem um ambiente de execução confiável (*Trusted Execution Environment*

- TEE), com as propriedades de segurança do TEE suportando a confidencialidade e a integridade de múltiplas aplicações confiáveis [ARM 2009].

2.2.5. AMD *Secure Processor*

Também conhecido como *Platform Security Processor* (PSP), o AMD *Secure Processor* é um subsistema de segurança em *hardware* dedicado integrado ao chip do processador, que executa de modo independente dos núcleos do processador principal. Esse subsistema proporciona um ambiente para tratamento de dados sensíveis isolado do sistema principal.

Conforme [Arthur and Challener 2015], para ser uma uma base confiável em *hardware*, podendo ser usada para o estabelecimento da cadeia de confiança, o PSP faz uso de um microcontrolador ARM *TrustZone* de 32 *bits* mas, apesar da utilização da arquitetura *TrustZone*, o PSP não é um núcleo virtual, mas um processador real, fisicamente separado, integrado ao SoC, que dispõe de uma SRAM dedicada e acesso direto ao coprocessador criptográfico. Além disso, o PSP dispõe de acesso aos recursos de memória do sistema e uma DRAM criptografada, com isolamento implementado em *hardware*.

O PSP também contém um coprocessador criptográfico composto de um gerador de números aleatórios, mecanismos para processamento de algoritmos criptográficos (AES, RSA e outros) e um bloco para armazenamento de chaves. Este bloco é composto de duas áreas de armazenamento: uma usada por *software* privilegiado, cuja leitura não é possível; e outra onde chaves podem ser carregadas, utilizadas e descartadas, em operações convencionais tanto de *software* executando no PSP ou no sistema operacional convencional. Também há uma lógica implementada em *hardware* para inicialização segura do núcleo da CPU e uma chave única da plataforma, que é distribuída para o bloco de armazenamento do coprocessador criptográfico durante o processo de inicialização.

Para garantir uma inicialização confiável, o PSP implementa o *Hardware Validated Boot* (HVB), que é uma forma de *boot* seguro não-modificável, implementado na ROM do SoC, que verifica a integridade da BIOS. A ROM faz a validação de uma chave de inicialização e então usa essa chave para validar o *firmware* do PSP que, por sua vez, é carregado e inicia a execução da aplicação do sistema.

2.2.6. Intel *Software Guard Extensions*

O *Intel Software Guard Extensions*, ou *Intel SGX*, é uma extensão ao conjunto de instruções da arquitetura *x86* que permite que aplicações possam ser executadas em uma área protegida de memória, chamada de *enclave*, que irá conter o código e os dados da aplicação. Um enclave é uma área protegida no espaço de endereçamento da aplicação, conforme mostrado na Figura 2.3, que garante a confidencialidade e integridade dos dados, evitando que esses dados sejam acessados por *malwares* que estejam em execução, e até mesmo por outros *softwares* com alto privilégio de execução, como monitores de máquinas virtuais, BIOS, e o próprio sistema operacional [McKeen et al. 2013, Jain et al. 2016].

Para garantir a confidencialidade e integridade dos dados, a arquitetura SGX adicionou novas instruções, uma nova arquitetura de processador e um novo modelo de execução, que incluem o carregamento do enclave em uma área protegida de memória, o acesso de recursos via mapeamento de tabelas de páginas e o escalonamento de aplicações dentro de enclaves. Após a aplicação ser carregada dentro de um enclave, ela fica protegida de todo

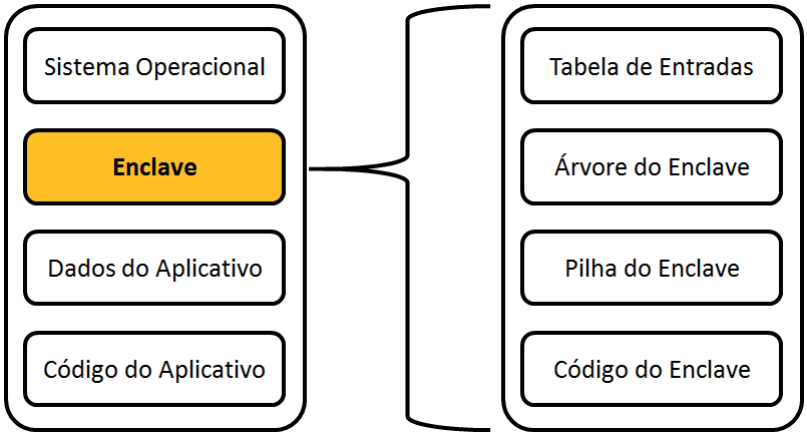

Figura 2.3. Enclave dentro do Espaço de Endereçamento Virtual da Aplicação (adaptado de [McKeen et al. 2013]).

e qualquer acesso externo ao enclave, inclusive de outras aplicações que estejam em outros enclaves. As tentativas de alterações não autorizadas de conteúdo dentro de um enclave são detectadas e impedidas, ou sua execução é abordada. Enquanto os dados do enclave estão tramitando entre os registradores e os outros blocos do processador, o acesso não autorizado é evitado utilizando os mecanismos de controle de acesso internos do próprio processador. Quando os dados são escritos na memória, eles são automaticamente cifrados e a integridade é mantida, evitando sondagens de memória ou outras técnicas para visualizar, modificar ou substituir os dados contidos em um enclave [Costan and Devadas 2016].

A cifragem de memória é feita utilizando os algoritmos padrão de criptografia, contendo proteções contra ataques de repetição. O fato de conectar os módulos de memória DRAM em outro sistema apenas dará acesso aos dados em sua forma cifrada; além disso, a chave de criptografia é armazenada em registradores dentro da CPU, não sendo acessível a componentes externos à mesma, e é alterada aleatoriamente em cada evento de hibernação ou reinício do sistema [Intel 2016b]. Maiores detalhes sobre a arquitetura, funcionalidades e implementação do SGX serão abordados na Seção 2.3.

2.2.7. Comparativo entre os mecanismos apresentados

Esta seção apresentou de forma sucinta uma coletânea dos principais mecanismos de segurança baseados em *hardware* disponíveis para sistemas computacionais modernos. A Tabela 2.1 apresenta um comparativo entre as principais características dos mecanismos abordados neste texto. Nota-se que a tecnologia Intel SGX abrange as principais características necessárias para o desenvolvimento de aplicações seguras. Por outro lado, o AMD *Secure Processor* consegue as mesmas características através da utilização do *TrustZone* e do TPM em um SoC.

2.3. Intel *Software Guard Extensions* (SGX)

O Intel SGX, introduzido na Seção 2.2.6, traz um novo conceito na construção de aplicações seguras, possibilitando o isolamento da parte sensível da aplicação dentro de uma estrutura denominada *enclave*. O *Intel Software Guard Extensions Developer Guide* [Intel 2016a] descreve o enclave como uma entidade de *software* monolítica que reduz a

Tabela 2.1. Comparativo entre os mecanismos de proteção baseados em *hardware*.

Tecnologia	Armazenamento seguro	Atestação	Isolamento de memória	Acelerador criptográfico
Cripto-processadores				✓
TPM	✓	✓		✓
Intel TXT	✓	✓	✓	
ARM TrustZone			✓	✓
AMD Secure Processor	✓	✓	✓	✓
Intel SGX	✓	✓	✓	✓

Trusted Computing Base (TCB) de um aplicativo para um sistema de tempo de execução confiável, com o domínio não confiável controlando a ordem em que as funções dentro do enclave são chamadas.

Em uma análise em alto nível, a arquitetura SGX oferece um modelo de programação muito semelhante ao utilizado para o desenvolvimento de aplicações regulares nos diversos sistemas operacionais, com o uso de *Dynamic Linked Library*, *Shared Objects*, ou similares, apenas com algumas diferenças em como a aplicação é projetada e desenvolvida para fazer uso dos recursos providos pelo SGX, já que a utilização indevida desses recursos pode levar a vulnerabilidades que podem ser exploradas posteriormente.

O código-fonte de um enclave não tem muitas diferenças quando comparado com o código-fonte de uma aplicação regular. No entanto, o código do enclave é carregado de uma maneira especial, de forma que uma vez que o enclave foi inicializado, o seu código e o restante da aplicação não podem ler diretamente os dados que residem no ambiente protegido ou alterar o comportamento do código dentro do enclave sem detecção. Por esta razão, é essencial que o desenvolvedor identifique quais componentes e recursos da aplicação devem ser protegidos, para que somente estes residam dentro de enclaves.

Normalmente, uma biblioteca compartilhada contém seções de dados e código correspondentes às variáveis e/ou objetos e funções e/ou métodos implementados, com o sistema operacional alocando um *heap* quando o processo que usa a biblioteca compartilhada é carregado, e uma pilha para cada *thread* gerada dentro do processo. Da mesma forma, uma biblioteca de enclaves contém seções de código e dados que serão carregadas na memória protegida (*Enclave Page Cache* - EPC) quando o enclave for criado. Existem também os metadados, que não são carregados no EPC, pois são utilizados pela aplicação que irá inicializar o enclave para determinar como carregá-lo corretamente no EPC. Os metadados definem um número de contextos de *threads* confiáveis, que inclui a pilha confiável e um *heap* criado na inicialização do enclave, sendo que ambos são necessários para suportar um ambiente de execução confiável. Além disso, os metadados compõem, juntamente com a assinatura do enclave, um certificado essencial para assegurar a autenticidade e a integridade do mesmo.

Mesmo que um enclave possa ser fornecido como um arquivo de biblioteca compartilhada, definir quais porções de código e dados serão colocadas dentro do enclave e quais permanecerão fora deste, na porção não protegida da aplicação, é um aspecto chave do desenvolvimento do enclave, sendo este o primeiro passo para projetar um aplicativo habili-

tado para fazer uso da arquitetura SGX. Esta é uma tarefa que fica a cargo do desenvolvedor da aplicação, já que este é quem melhor conhece o aplicativo a ser desenvolvido.

2.3.1. Organização de Memória do Enclave

O código e os dados do enclave são armazenados em uma parte reservada da memória, chamada *Processor Reserved Memory* (PRM), que é um subconjunto da memória principal que não pode ser acessado diretamente por outras aplicações. O controlador de memória da CPU também rejeita transferências via DMA para o PRM, de forma a protegê-lo do acesso através de dispositivos periféricos.

O PRM é uma área contígua de memória, tendo o seu tamanho e seu endereço de início como um valor inteiro e potência de dois, o que permite que seus endereços possam ser verificados por um *hardware* simples e barato. O PRM é um detalhe da microarquitetura que pode mudar em implementações futuras do SGX.

O conteúdo dos enclaves e as estruturas de dados associadas a estes são armazenados em um subconjunto do PRM, chamado de *Enclave Page Cache* (EPC). O EPC, por sua vez, é subdividido em páginas de 4 KBytes, que podem ser atribuídas a diferentes enclaves. Isso permite que se tenha múltiplos enclaves em execução no sistema ao mesmo tempo, o que é uma necessidade para ambientes multiprocessados.

O EPC é gerenciado pelo mesmo *software* que gerencia o restante da memória física do computador, que pode ser um hipervisor ou um *kernel* do sistema operacional. O gerenciador de memória física utiliza as instruções do SGX para alocar páginas não utilizadas por enclaves e para liberar páginas previamente alocadas [Costan and Devadas 2016].

Como o EPC está contido dentro do PRM, quaisquer *softwares* que não utilizem enclaves não podem ter acesso a este, o que garante um isolamento do enclave, mas também cria alguns obstáculos para um software carregar trechos de código e dados iniciais para um enclave recém criado. Esses obstáculos são contornados pelo uso de instruções do próprio SGX que permitem carregar conteúdo de outras páginas de memória para as páginas do EPC. Assim, quem é responsável por carregar as páginas de memória para o EPC é o *software* não confiável. Tendo isso em vista, este processo é validado pelo SGX, que pode recusar qualquer operação que comprometa as garantias de segurança dos enclaves, como, por exemplo, a tentativa de atribuir a mesma página no EPC para dois enclaves diferentes.

O armazenamento do EPC na memória principal é protegido cifrando os dados, de forma a possibilitar uma defesa contra ataques à memória, tanto por *software* quanto por *hardware*. Para isso, uma unidade de *hardware* chamada *Memory Encryption Engine* (MEE) cuida da criptografia e da integridade dos dados quando estes estão sendo transferidos entre a memória e o processador, sendo que a região de memória onde um MEE opera é chamado de *Região MEE*. Um processador que suporta a arquitetura SGX e implementa o EPC em uma região de memória cifrada também dará suporte para a BIOS reservar um intervalo de memória chamado *Processor Reserved Memory* (PRM), que poderá ser protegido por uma ou mais regiões MEE.

O processador bloqueia qualquer acesso ao PRM vindo de agentes externos, tratando esses acessos como referências não existentes na memória. Já o acesso a páginas de memória dentro de um enclave, utilizando instruções de memória como MOV, é verificado

pelo hardware seguindo o fluxograma descrito na Figura 2.4. Inicialmente verifica-se se o processo está sendo executado dentro de um enclave e se o mesmo pertence ao enclave que ele está querendo acessar; em caso afirmativo, o acesso é liberado; caso contrário, o acesso é bloqueado, tratando-o como uma referência a uma posição de memória inexistente. Da mesma forma, as tentativas de acesso às páginas de memória dentro de um enclave por parte de um processo que não está sendo executado em um enclave, também são tratadas como referências inexistentes [McKeen et al. 2013, Intel 2014, Costan and Devadas 2016].

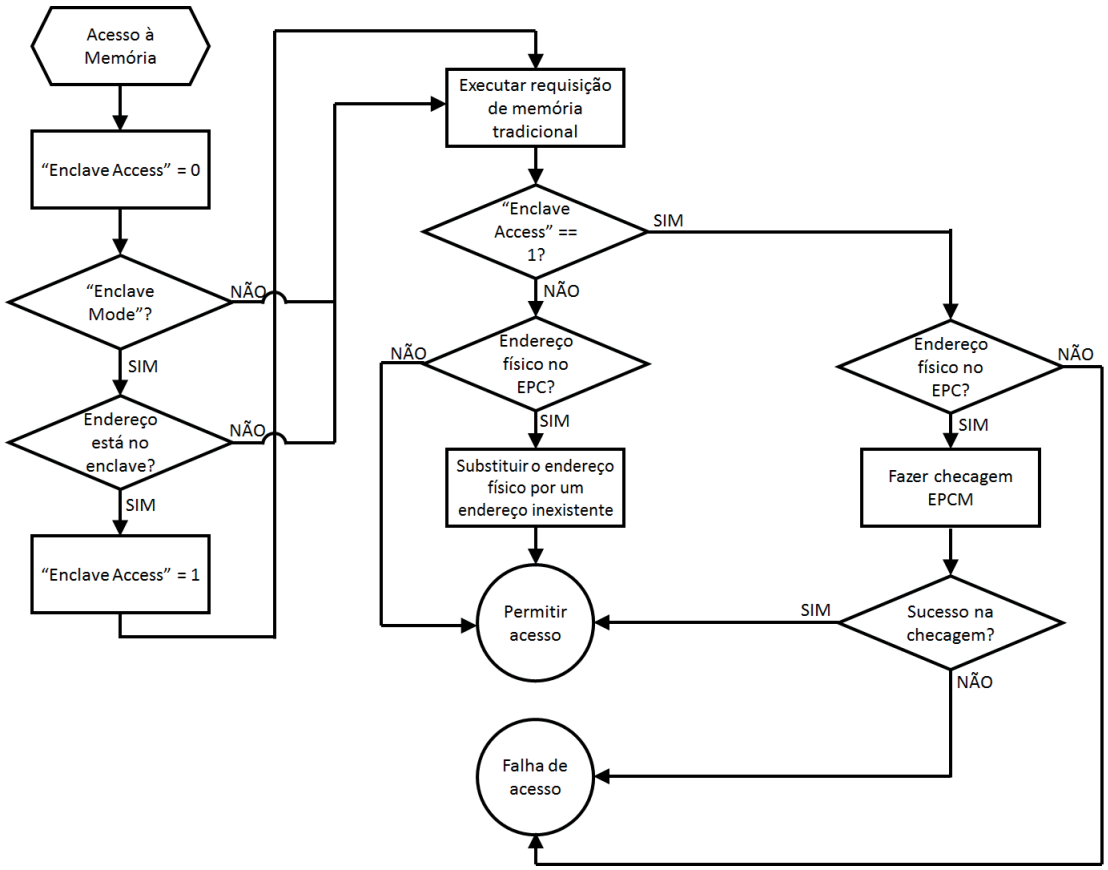

Figura 2.4. Fluxograma de acesso à memória (adaptado de [McKeen et al. 2013]).

Para realizar suas verificações de segurança, o SGX grava algumas informações sobre as decisões de alocação para cada página EPC no EPCM *(Enclave Page Cache Map)*, que é uma matriz com uma entrada para cada página EPC. Ela permite que a computação do endereço da entrada EPCM de uma página seja efetuada com apenas uma operação de troca *bit* a *bit* e uma adição. O conteúdo do EPCM é utilizado apenas pelas verificações de segurança da SGX; em condições normais de funcionamento, o EPCM não gera qualquer comportamento visível ao *software*, sendo que a sua utilização é transparente para os desenvolvedores. A organização do PRM é mostrada na Figura 2.5.

De acordo com [Costan and Devadas 2016], o EPCM basicamente armazena três informações para identificar o proprietário de cada página EPC:

- Um campo de um único *bit* (*VALID*) indica se a página EPC foi alocada (*bit* 1)

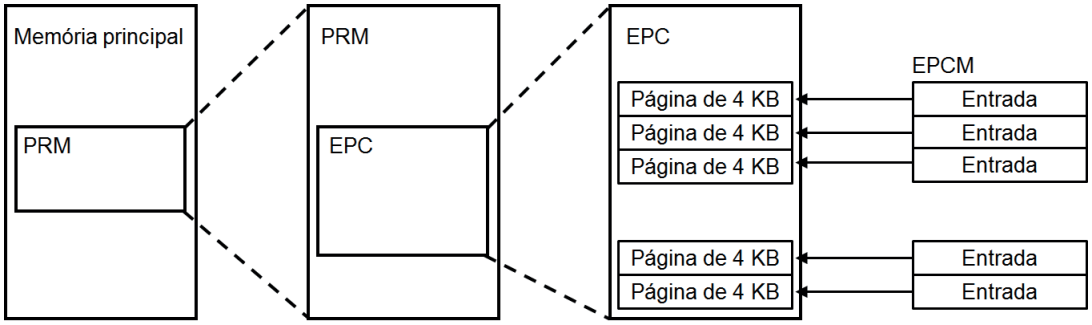

Figura 2.5. Os dados do enclave são armazenados no EPC, que é um subconjunto do PRM, que por sua vez é uma área contígua de memória que não pode ser acessada por outros softwares ou por dispositivos via DMA (adaptado de [Costan and Devadas 2016]).

ou está livre (*bit* 0), o que evita que dados sejam sobrescritos em páginas EPC já alocadas.

- Um segundo campo (*ENCLAVESECS*) identifica a qual enclave a página EPC pertence, de forma a impedir que um enclave acesse informações confidenciais de outro enclave. Isso impede que os enclaves se comuniquem via memória compartilhada utilizando páginas EPC, já que cada página EPC irá pertencer a um único enclave. Por outro lado, existe a possibilidade de os enclaves compartilharem dados residentes em uma memória não confiável, que está em uma região fora do EPC.

- Por fim, a instrução utilizada para alocar uma página EPC também determina o seu uso pretendido, que é registrado no campo de tipo de página (*PT*) da entrada EPCM correspondente. As páginas que armazenam o código e dados de um enclave são consideradas como tendo um tipo regular (*PT_REG*). As páginas dedicadas ao armazenamento das estruturas de dados de suporte da arquitetura SGX são marcadas com tipos especiais.

Um exemplo de página EPC com um tipo especial é o *SGX Enclave Control Structure* (SECS), marcada com o tipo PT_SECS, que armazena metadados do enclave. As páginas PT_SECS não são mapeadas no espaço de endereçamento dos enclaves, e são utilizadas exclusivamente pela CPU.

O SECS pode ser considerado como um sinônimo da identidade de um enclave, já que o primeiro passo para se criar um enclave é alocar uma página no EPC para armazenar seu SECS, e a última etapa para a destruição de um enclave é desalocar a página no EPC que contém o seu SECS. A entrada no EPCM que identifica um enclave possui uma página EPC que aponta para o SECS do enclave, sendo que o endereço virtual do SECS é utilizado para identificar o enclave quando este invoca uma instrução SGX.

Como as instruções do SGX utilizam endereços SECS para identificar os enclaves, o *software* não confiável deve criar entradas na sua tabela de páginas apontando para os SECS dos enclaves que gerencia. Apesar disso, o *software* não confiável não pode acessar as páginas SECS, já que estas são armazenadas no PRM, e tampouco os enclaves podem acessá-las, visto que o SGX impede explicitamente que o código do enclave acesse as

páginas SECS. Esta limitação de acesso às páginas SECS existe para que a arquitetura SGX possa armazenar informações confidenciais no SECS e possa assumir que nenhum *software* potencialmente mal-intencionado irá acessar essas informações. Um exemplo de informação confidencial armazenada no SECS é a medição do enclave (descrita na Seção 2.3.3): se o *software* fosse capaz de alterar a medição do enclave, o esquema de atestação da arquitetura SGX (descrito na Seção 2.3.6) não proporcionaria garantias de segurança.

2.3.2. Ciclo de Vida do Enclave

O processo de criação de um enclave consiste em várias etapas: inicialização da estrutura de controle do enclave; alocação das páginas de memória no EPC (*Enclave Page Cache*) e carregamento do conteúdo do enclave para essas páginas; medição do conteúdo do enclave; e criação de um identificador para o enclave. Antes de iniciar a criação do enclave, o processo já estará residindo na memória principal, estando livre para qualquer inspeção e análise e, após ele ser carregado para dentro do enclave, seus dados e código estarão protegidos de quaisquer acessos externos. O ciclo de vida do enclave é apresentado na Figura 2.6, que mostra também as instruções responsáveis pelo gerenciamento do enclave.

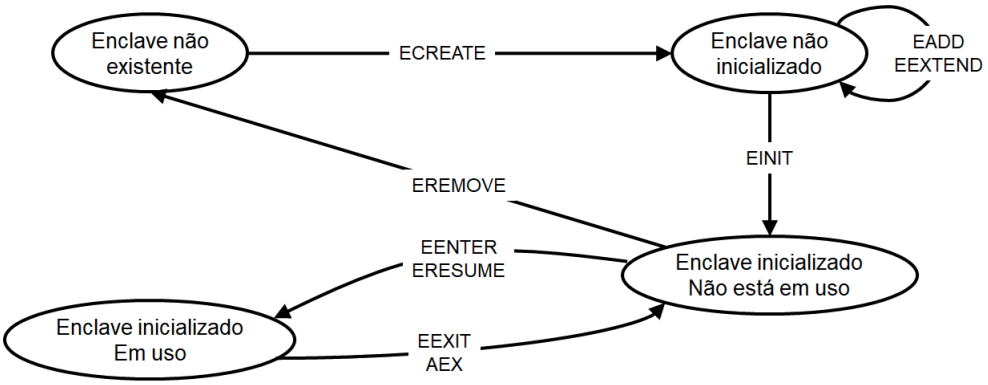

Figura 2.6. Instruções de gerenciamento do ciclo de vida do enclave e diagrama de transição de estados (adaptado de [Costan and Devadas 2016]).

A instrução *ECREATE* inicia a criação do enclave e inicializa o *SGX Enclave Control Structure* (SECS), que irá conter informações globais sobre o enclave. Cada página de memória é adicionada ao enclave utilizando a instrução *EADD* e a instrução *EEXTEND* adiciona a medição criptográfica do conteúdo do enclave. Finalmente, a instrução *EINIT* completa o processo de criação do enclave e cria a sua identidade, permitindo então que ele seja utilizado.

Após esse processo, uma questão importante é o controle da transferência de execução ao entrar e sair do enclave. No procedimento de entrada no enclave, é necessário limpar quaisquer valores em cache que possam se sobrepor à região protegida pelo enclave, além de checar todos os endereços de memória protegidos pelo enclave, identificar a instrução dentro do enclave para a qual o processador deve transferir a execução e habilitar o modo de execução em enclave. Já ao sair de um enclave, novamente deve-se limpar os valores em cache que se referem a endereços de memória protegidos pelo enclave, de forma a evitar que outro *software* possa acessar essas informações. Para entradas e saídas efetuadas programaticamente no enclave, o SGX oferece as instruções *EENTER* e

EEXIT, respectivamente. Se a saída do enclave ocorrer devido a algum evento ou falha, o processador executará uma rotina chamada *Asynchronous Exit* (AEX), que irá salvar o estado do enclave, limpar os registradores e armazenar o endereço da instrução que gerou a falha, permitindo que a execução seja posteriormente retomada, invocando a instrução *ERESUME*.

Por fim, o enclave é destruído através da instrução *EREMOVE*, que irá liberar todas as páginas EPC utilizadas pelo enclave, garantindo que nenhum processador lógico está executando instruções dentro das páginas EPC a serem removidas. O enclave é completamente destruído quando a página EPC que contém a sua estrutura *SECS* é liberada [McKeen et al. 2013, Intel 2014, Costan and Devadas 2016].

2.3.3. Medição e Assinatura do Enclave

O Intel *Software Guard Extensions Developer Guide* [Intel 2016a] define três principais atividades para estabelecer a confiança em um enclave:

- **Medição:** Um enclave é instanciado em um ambiente confiável e é feita uma gravação precisa e protegida de sua identidade.
- **Atestação:** Demonstra a outras entidades que o enclave foi instanciado de maneira correta.
- **Selagem:** Permite que os dados pertencentes a um ambiente confiável sejam vinculados a ele de tal forma que só podem ser restaurados quando o ambiente confiável é restaurado.

Nesta seção, será abordada a medição do enclave, sendo que a selagem e a atestação serão tratados nas seções seguintes.

Cada enclave possui um certificado auto-assinado pelo autor do enclave, também conhecido como *Enclave Signature* (SIGSTRUCT). A assinatura do enclave contém informações que permitem que a arquitetura Intel SGX detecte se alguma parte do enclave foi adulterada, de forma a permitir que um enclave possa provar que ele foi corretamente carregado no EPC e pode ser confiável. No entanto, o *hardware* apenas verifica a medição do enclave quando este é carregado. Assim, qualquer pessoa pode modificar um enclave e assiná-lo com sua própria chave. Para evitar esse tipo de ataque, a assinatura do enclave também identifica o autor do enclave, contendo vários campos essenciais para que o enclave seja atestado por entidades externas:

- ***Enclave Measurement*:** Um *hash* de 256 *bits* único, que identifica o código e os dados iniciais a serem colocados dentro do enclave, bem como a ordem e a posição na qual eles devem ser colocados, e as propriedades de segurança dessas páginas. Uma alteração em qualquer uma dessas variáveis resultará em uma medida diferente. Quando as páginas de código/dados do enclave são carregadas para o EPC, a CPU efetua a medição do enclave e armazena esse valor no registrador MRENCLAVE. Em seguida, a CPU compara o conteúdo do MRENCLAVE com o valor contido no SIGSTRUCT desse enclave e, somente se eles forem idênticos, a CPU permitirá que o enclave seja inicializado.

- **Chave Pública do Autor do Enclave:** Após um enclave ser inicializado com êxito, a CPU registra um *hash* da chave pública do autor do enclave no registrador MRSIGNER. O conteúdo do MRSIGNER servirá como identidade do autor do enclave. Assim, os enclaves que foram autenticados com a mesma chave devem ter o mesmo valor colocado no registo MRSIGNER.

- ***Security Version Number* do Enclave (ISVSVN):** O autor do enclave atribui um SVN (*Security Version Number*) a cada versão de um enclave. O SVN reflete o nível da propriedade de segurança do enclave, e deve monotonicamente aumentar com melhorias das propriedades de segurança. Depois que um enclave é inicializado com sucesso, a CPU registra o SVN, que pode ser usado durante a atestação. Diferentes versões de um enclave com as mesmas propriedades de segurança devem ter o mesmo SVN. Por exemplo, uma nova versão de um enclave com correções de *bugs* não relacionados à segurança deve ter o mesmo SVN que a versão anterior.

- ***Product ID* do Enclave (ISVPRODID):** Permite que o autor marque seus enclaves com identificadores de produto com a mesma identidade do autor. Depois que um enclave é inicializado com êxito, o *Product ID* é registrado pela CPU, que pode ser usado durante a atestação.

O desenvolvedor deve fornecer o SVN e o *Product ID* de um enclave, bem como um par de chaves para gerar a assinatura de enclave. A CPU deriva a identidade do autor do enclave a partir de sua chave pública, já a chave privada é utilizada para assinar o enclave. O cálculo da medida do enclave deve ser realizado com base no código e nos dados iniciais a serem colocados dentro do enclave, a ordem esperada e a posição em que eles devem ser colocados e as propriedades de segurança dessas páginas. O código e os dados iniciais a serem colocados no interior do enclave, bem como as propriedades de segurança dessas páginas são gerados pelo compilador, enquanto a sua colocação no enclave é controlada pelo carregador de enclave. Assim, o cálculo de medição deve seguir o comportamento esperado do carregador de enclave no que diz respeito à forma de colocar o código e os dados iniciais dentro do enclave.

A chave de assinatura do desenvolvedor é parte da identidade do enclave e é fundamental para proteger seus segredos. Um intruso que comprometa a chave de assinatura privada de um ISV (*Independent Software Vendor*) poderá ser capaz de escrever um enclave malicioso que ateste com êxito a identidade dos enclaves legítimos e/ou escrever um *malware* que usa o enclave malicioso para comprometer dados selados em plataformas individuais [Intel 2016a].

2.3.4. Interface do Enclave: Chamadas ECALL e OCALL

A arquitetura SGX requer que toda a funcionalidade dentro de um enclave seja ligada estaticamente, em tempo de compilação. Isso cria um *trade-off* de desempenho/tamanho que os desenvolvedores devem analisar cuidadosamente, pois impacta no tamanho do TCB. Ao usar a funcionalidade de biblioteca estática, os desenvolvedores têm duas opções: fornecer uma camada para executar funções fora do enclave, adicionando uma sobrecarga de desempenho na aplicação; ou incluir a implementação da biblioteca como parte do enclave, provocando um aumento no tamanho do TCB. A utilização da primeira opção é

representada pela Figura 2.7, onde tem-se a divisão entre os componentes confiável e não confiável da aplicação, e ambos tem uma camada chamada *edge routines*, ou rotinas de borda, que são funções que podem ser executadas dentro ou fora de um enclave, com a função de ligar uma chamada do aplicativo com uma função dentro do enclave ou uma chamada do enclave com uma função no aplicativo.

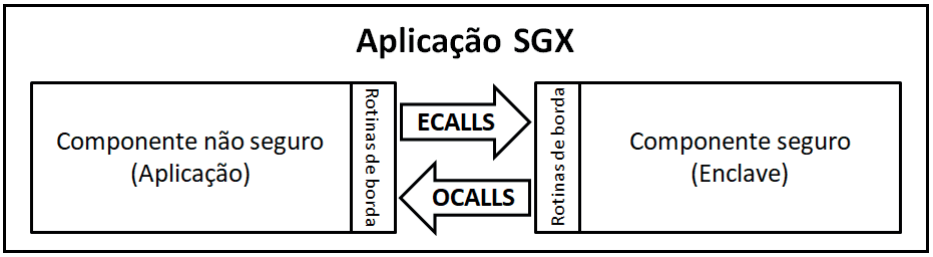

Figura 2.7. Divisão da aplicação em dois componentes, confiável e não confiável, e a comunicação entre os dois componentes (adaptado de [Intel 2016a]).

Particionar um aplicativo em dois componentes, confiável e não confiável, tem implicações adicionais do ponto de vista da segurança. Pode-se dizer que uma porção de código e dados menor normalmente implica uma menor probabilidade de ter defeitos no produto final, e também resulta em uma análise de segurança mais simples e um *software* mais seguro. Outro ponto a ser considerado é que, quanto menor for o tamanho do enclave, mais rápido este será capaz de efetuar o *backup* dos dados fora do enclave quando ocorrer um evento de hibernação do sistema, além de também carregar o enclave mais rapidamente quando o sistema é restaurado. Assim, embora seja possível mover a maior parte do código do aplicativo para um enclave, na maioria dos casos isso não seria desejável, visto que o tamanho de TCB deve ser um fator a considerar ao projetar o que será colocado dentro de um enclave, com o desenvolvedor tendo o objetivo de minimizar o tamanho deste [Intel 2016a].

2.3.4.1. Chamadas para o Enclave (ECALLs)

Após definir os componentes confiável (enclave) e não confiável (aplicativo) de uma aplicação habilitada para a arquitetura SGX, o desenvolvedor deve definir cuidadosamente a interface entre esses componentes. O código confiável é executado nos seguintes cenários:

- Quando o componente não confiável faz uma chamada explícita para uma função pertencente ao enclave, através de um ECALL. Este procedimento é o mesmo que um aplicativo regular chamar uma função em uma biblioteca compartilhada.

- Após o retorno de uma chamada feita a partir do enclave para o aplicativo externo (OCALL). O retorno de um OCALL é semelhante ao que acontece quando uma biblioteca compartilhada envia o retorno de uma chamada efetuada por outra biblioteca compartilhada, por exemplo, o retorno de uma biblioteca *Standard C* para executar uma operação de E/S. Quando um OCALL retorna, a função confiável que fez a chamada para fora do enclave continua a sua execução.

- A arquitetura SGX suporta interrupções durante a execução do enclave, no entanto, garante que, ao retornar a para dentro do enclave, sua execução continuará como se a interrupção nunca tivesse ocorrido, da mesma forma que deve ocorrer quando ocorre uma interrupção durante a execução de uma função em uma biblioteca regular.

Um enclave deve disponibilizar um conjunto de métodos e/ou funções que estarão disponíveis para que o aplicativo faça uso (ECALLs) e descrever quais serviços o aplicativo deve fornecer (OCALLs), sendo que essas funções devem ser definidas pelo desenvolvedor.

As funções ECALLs expõem a interface do enclave que o aplicativo não confiável pode utilizar e, por isso, deve-se limitar o número de ECALLs para reduzir a superfície de ataque ao enclave. O desenvolvedor deve estar ciente de que o enclave não tem controle sobre quais ECALLs são executadas ou em que ordem estas são chamadas. Desta forma, um enclave não deve depender de ECALLs sendo executadas em uma determinada ordem.

Em contrapartida, as funções ECALL e OCALL somente podem ser chamadas após a inicialização do enclave, significando que as alocações de endereço necessárias foram executadas com êxito, todos os dados globais (incluídos os relativos à segurança) foram inicializados com êxito, o *trusted thread context* foi inicializado com todos os seus dados de segurança, e as funções de inicialização foram executadas de forma completa.

As entradas e saídas de dados no enclave podem ser observadas e modificadas por aplicações ou códigos executados em um ambiente não confiável. Desta forma, o desenvolvedor do enclave nunca deve confiar em qualquer dado proveniente de um domínio de *software* não confiável, e deve sempre verificar os parâmetros de entrada das funções ECALLs e os valores retornados pelas funções OCALLs. Após os dados terem sido verificados e ter sido identificada a sua origem e/ou destino (entidade remota, usuários, etc.), deve-se decidir pela aplicação ou não de criptografia para proteger os dados que em algum momento estarão expostos ao domínio de *software* não confiável.

Deve-se ter também um cuidado especial para tratar os ponteiros e as passagens de parâmetros por referência a funções ECALL, já que o aplicativo não confiável pode passar um ponteiro referenciando um endereço de memória dentro do limite do enclave, o que pode fazer com que o enclave sobrescreva seus dados ou código de forma indevida, podendo até mesmo ocasionar o vazamento de informações confidenciais do enclave.

2.3.4.2. Chamadas Externas ao Enclave (OCALLs)

Os enclaves não podem acessar diretamente os serviços fornecidos pelo sistema operacional (*system calls*). Para isso, um enclave deve executar uma chamada OCALL para uma rotina no aplicativo não confiável, o que acrescenta uma sobrecarga de desempenho, mas não afeta a confidencialidade dos dados. Porém, a comunicação com o sistema operacional pode requerer a divulgação de dados ou a importação de dados vindos do aplicativo não confiável que precisam ser tratados adequadamente. Além disso, as chamadas OCALLs, necessárias para operações como sincronização de *threads* e operações de E/S, são expostas ao domínio não confiável da aplicação e, portanto, podem ser associadas a alguns riscos de segurança. Desta forma, um enclave deve ser projetado de tal forma que evite um vazamento de

informações que permitiriam a um invasor, que está examinando as funções não confiáveis que são chamadas pelo enclave, obter informações sobre os dados confidenciais do mesmo.

O enclave também deve tratar as situações em que uma chamada de função OCALL não é executada por completo, para evitar situações em que, por exemplo, um invasor intercepte uma solicitação do enclave para gravar dados em disco e emita uma resposta informando que a operação foi concluída com êxito, quando na verdade a operação não foi realizada. O retorno de uma chamada OCALL ocorre no mesmo contexto que estava sendo executado antes da função OCALL ser chamada, mantendo, assim, o fluxo de execução do enclave.

Ao executar uma chamada OCALL, o domínio não confiável da aplicação também pode efetuar chamadas ECALL, retornando a execução para o enclave durante a execução dessa chamada. Uma vez fora do enclave, um intruso pode tentar encontrar vulnerabilidades invocando funções ECALL recursivamente para o enclave, assim, o desenvolvedor deve limitar quais chamadas ECALL são permitidas durante a execução de um OCALL, ou até mesmo impedindo que as chamadas ECALL ocorram.

Uma chamada OCALL expõe somente os dados passados a ela por parâmetro. O valor de retorno da função e os parâmetros de retorno passados a ela por referência pertencem ao domínio confiável da aplicação e, por isso, não estão acessíveis fora desse domínio. Se o valor de retorno de uma função OCALL é um ponteiro, apenas a referência deste estará dentro do enclave; os valores referenciados por este ponteiro devem ser previamente verificados para serem utilizados no domínio confiável da aplicação.

2.3.5. Selagem de Dados

Quando um enclave é instanciado, o desenvolvedor deve identificar quais dados e/ou estados devem ser protegidos e que devam ser preservados (“selados”) após o enclave ser destruído. O enclave pode ser encerrado quando a aplicação completa a sua execução e encerra o enclave, quando a aplicação é encerrada (de maneira forçada ou não), e quando o computador entra em hibernação ou é desligado.

Em geral, os dados fornecidos a um enclave são perdidos quando o enclave é destruído mas, se estes dados precisam ser preservados durante um desses eventos para uso futuro, estes devem ser armazenados fora dos limites do enclave antes de destruí-lo. Para proteger e preservar os dados, existe um mecanismo que permite ao enclave recuperar uma chave exclusiva para esse enclave. Essa chave só pode ser gerada por esse enclave naquela plataforma específica. O enclave usa essa chave para cifrar dados para a plataforma ou para decifrar dados já existentes na plataforma. Estas operações de cifragem e decifragem são chamadas de *sealing* e *unsealing*, respectivamente[Intel 2016a].

Ao selar dados, o enclave precisa especificar as condições que precisam ser atendidas quando os dados devem ser recuperados e, para isto, existem duas opções disponíveis. A primeira opção envolve a criação de um selo para o enclave corrente, utilizando a medida do enclave (MRENCLAVE) para a criação do selo. Assim, somente um enclave com a mesma medida será capaz de acessar os dados que forem selados desta maneira. Isso significa que, se o código do enclave sofrer alterações, isso refletirá em uma mudança na medição do enclave e a chave de criptografia irá ser alterada também, impedindo a

recuperação dos dados previamente selados.

A segunda opção é utilizar a assinatura do autor do enclave (MRSIGNER), juntamente com o *Product ID* para gerar o selo do enclave. Assim, somente um enclave com o mesmo valor no registrador MRSIGNER e o mesmo *Product ID* será capaz de acessar os dados que foram selados dessa maneira. Desta forma, permite-se que um enclave seja atualizado pelo autor, mas não requer um processo complexo de atualização para desbloquear dados selados na versão anterior do enclave (que terá uma valor de MRENCLAVE diferente) e também permite que enclaves do mesmo autor compartilhem dados selados.

Os desenvolvedores também podem definir o *Security Version Number* (SVN) quando codificam o enclave, que também é armazenado na CPU quando o enclave é instanciado. Um enclave deve fornecer o SVN em sua solicitação para obter a chave de criptografia da CPU, não sendo possível especificar um SVN superior ao especificado na assinatura do enclave (ISVSVN). No entanto, o enclave pode especificar um SVN anterior ao ISVSVN do enclave, dando-lhe a capacidade de acessar dados selados por uma versão anterior do enclave, o que facilitaria as atualizações de *software*, por exemplo.

Além disso, as aplicações que executarão em enclaves não devem ser distribuídas com os dados confidenciais, mas sim, após a instalação, contatar um provedor de serviços para solicitar esses dados. Este processo é descrito na Figura 2.8 [Anati et al. 2013].

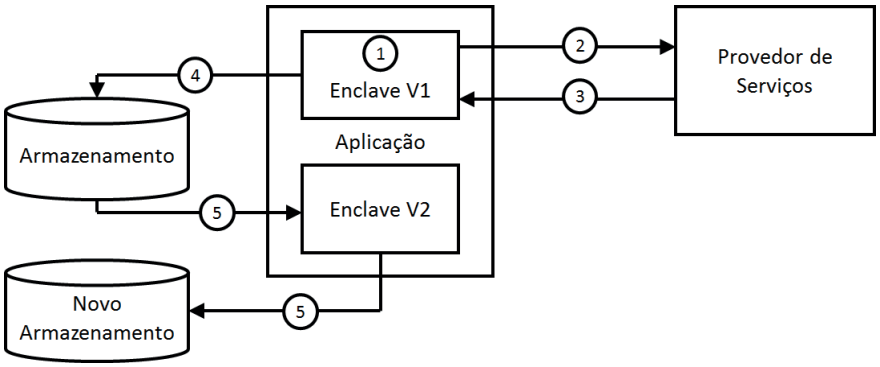

Figura 2.8. Fluxograma referente à integridade no armazenamento de dados (adaptado de [Anati et al. 2013]).

No passo 1, o *software* inicia a criação do enclave para que possa ser executado de forma segura. No passo 2, o enclave contata o provedor de serviços para efetuar o *download* dos dados; nesta etapa, o provedor irá gerar uma chave que identificará a máquina e o enclave que está executando a aplicação. Posteriormente, no passo 3, o provedor estabelece uma linha de comunicação segura com a aplicação e envia os dados para o enclave. Já no passo 4, o enclave utiliza uma chave de criptografia baseada em *hardware* para codificar e armazenar os dados confidenciais em um sistema de armazenamento permanente, garantindo que os dados serão acessados somente quando o enclave for restaurado. No passo 5 tem-se a situação de uma atualização de *software*, sendo que neste caso o enclave irá acessar os dados e gerar uma nova chave de criptografia, de forma que somente a nova versão do *software* consiga acessar os dados novamente, impedindo que versões anteriores da aplicação tenham acesso a esses dados.

Todo esse processo tem como finalidade proteger os dados confidenciais até mesmo

de versões anteriores da aplicação que possam conter brechas que possibilitem o acesso indevido aos mesmos. Desta forma, quando essas brechas forem detectadas, a empresa responsável pelo desenvolvimento do *software* pode corrigi-las e disponibilizar uma atualização, podendo impor que apenas os usuários que disponham da última versão do *software* possam acessar os dados e o provedor [Intel 2014].

2.3.6. Atestação

Atestação é o processo de comprovar que um *software* foi corretamente estabelecido em uma plataforma. No caso da Intel SGX, é o mecanismo pelo qual uma terceira entidade estabelece que uma determinada entidade de *software* está em execução em uma plataforma habilitada para Intel SGX e protegida dentro de um enclave, antes de provisionar esse *software* com segredos e dados protegidos. A atestação depende da capacidade de uma plataforma produzir uma credencial que reflita com precisão a assinatura de um enclave, que inclui informações sobre as propriedades de segurança do enclave. A arquitetura Intel SGX fornece os mecanismos para suportar duas formas de atestação. Existe um mecanismo para criar uma afirmação básica entre enclaves em execução na mesma plataforma, que suporta atestação local ou intra-plataforma, e outro mecanismo que fornece a base para a atestação remota, com os enclaves executando em dispositivos diferentes.

A atestação intra-plataforma é importante, pois os desenvolvedores podem escrever enclaves que podem cooperar uns com os outros para executar alguma função de nível mais alto. Para isso, os desenvolvedores precisam de um mecanismo que permita a um enclave provar sua identidade e autenticidade para outra parte dentro da plataforma local.

Um enclave pode pedir ao *hardware* para gerar uma credencial, também conhecida como relatório ou *EREPORT*, que inclui uma prova criptográfica de que o enclave existe na plataforma. Esse relatório pode ser fornecido a outro enclave para verificar se este foi gerado na mesma plataforma. O mecanismo de atestação utilizado para o certificado de enclave intra-plataforma utiliza um sistema de chaves simétricas, onde apenas o enclave que verifica a estrutura de relatório e o *hardware* de enclave que cria o relatório conhecem a chave. O processo de atestação é efetuado em três etapas, como mostra a Figura 2.9 [Anati et al. 2013].

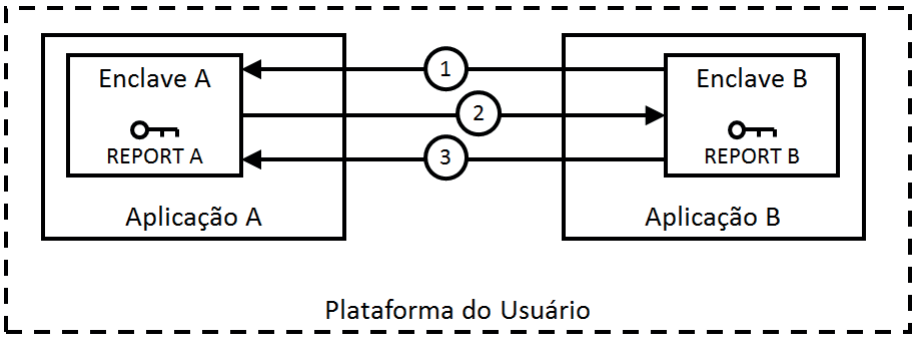

Figura 2.9. Autenticação intra-plataforma (adaptado de [Anati et al. 2013]).

No primeiro passo, o enclave A obtém a identificação do enclave B, utilizando uma comunicação aberta. No passo 2 o enclave A cria a estrutura *REPORT* utilizando a identificação do enclave B e envia esta estrutura ao enclave B, ainda utilizando uma

comunicação aberta. No terceiro passo, o enclave B utiliza os dados contidos no *REPORT* enviado por A de forma a verificar que o enclave A está sendo executado no mesmo dispositivo que B e, confirmando estes dados, o enclave B também pode criar uma estrutura *REPORT* e enviar esta ao enclave A. Assim, os dois enclaves estão autenticados e podem trocar mensagens de maneira segura.

Já a autenticação inter-plataforma, ou remota, requer o uso de criptografia assimétrica e de um enclave especial, chamado de *Quoting Enclave*, que irá verificar as estruturas *REPORT* dos outros enclaves da máquina utilizando a autenticação intra-plataforma e então substituir o *message authentication code* (MAC) dessas estruturas com uma assinatura criada através de uma chave assimétrica, utilizando o *Intel Enhanced Privacy ID* (EPID) e tendo como saída uma outra estrutura chamada *QUOTE*. O processo de autenticação remota é descrito na Figura 2.10 [Anati et al. 2013].

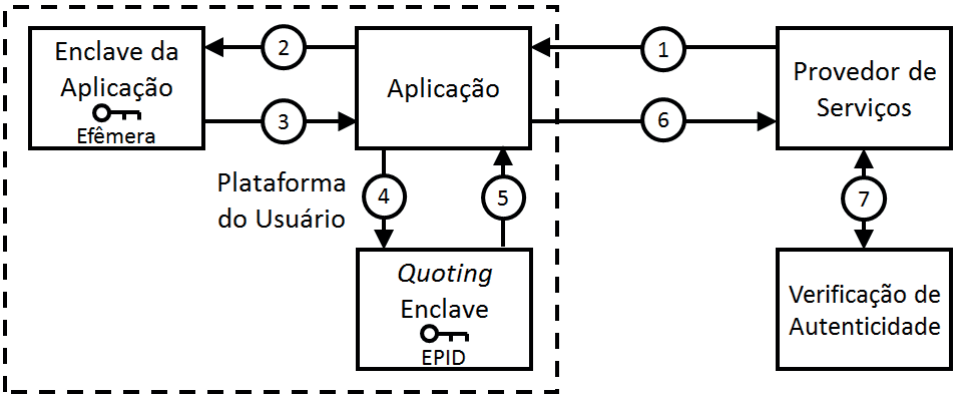

Figura 2.10. Autenticação remota (adaptado de [Anati et al. 2013]).

Primeiramente, o provedor de serviços solicita que a aplicação prove que está executando os componentes necessários dentro de um ou mais enclaves. No segundo passo, a aplicação encaminha a solicitação do provedor de serviços, juntamente com a identidade do *Quoting Enclave*, para o enclave da aplicação. No passo 3, o enclave da aplicação gera uma chave pública efêmera que deverá ser utilizada pelo provedor de serviços para as próximas comunicações e então gera a estrutura *REPORT*, enviando esta para a aplicação. No passo 4, a aplicação encaminha a estrutura *REPORT* para que o *Quoting Enclave* a assine. No passo 5 o *Quoting Enclave* verifica a estrutura *REPORT*, cria a estrutura *QUOTE*, assinando-a com a sua chave EPID, e a devolve à aplicação. No passo seguinte (6), a aplicação envia a estrutura *QUOTE* para o provedor de serviços. Finalmente, o provedor de serviços pode utilizar um verificador de autenticação para analisar a resposta recebida (7) e verificar se a aplicação satisfaz os requisitos.

A arquitetura Intel SGX também provê um enclave especial conhecido como o *Quoting Enclave*, que verifica os relatórios que foram criados para o MRENCLAVE e, em seguida, converte e assina esses relatórios usando uma chave assimétrica específica do dispositivo, a chave *Intel EPID*. A saída desse processo é chamada uma *quote*, e pode ser verificada fora da plataforma. Somente o *Quoting Enclave* instanciado tem acesso à chave Intel EPID, assim a chave da CPU nunca é exposta fora da plataforma.

O Intel EPID é um esquema de assinatura de grupo, que permite que as plataformas

assinem criptograficamente objetos ao mesmo tempo em que preservam a privacidade do assinante, com cada assinante de um grupo tendo sua própria chave privada para assinatura, e os verificadores utilizando a mesma chave pública do grupo para verificar assinaturas individuais, impedindo que os usuários possam ser identificados de forma exclusiva [Intel 2016a].

Uma entidade remota pode fornecer dados confidenciais para um enclave após a atestação remota ter sido realizada, com a transferência dos dados sendo conduzida através de um canal seguro. O estabelecimento de canal seguro deve estar vinculado ao processo de atestação remota, para evitar que o servidor remoto forneça dados confidenciais para uma entidade diferente do enclave que foi atestado.

Existem diversas maneiras de estabelecer um canal confiável para a comunicação remota entre em enclave e um provedor de serviços. Uma dessas maneiras é o enclave autenticar o servidor para garantir que receberá os dados de uma entidade confiável e, após isso, o enclave pode gerar um par de chaves público/privada para aquela comunicação, enviando a chave pública ao servidor. Após o servidor validar o enclave que enviou a chave, o próprio servidor pode gerar uma chave de criptografia simétrica E, cifrá-la com a chave pública P e enviar $P(E)$ através do canal de comunicação para a aplicação remota. Após receber os dados, o enclave pode decifrar $P(E)$ utilizando a sua chave privada e, assim, tanto o enclave quanto o servidor remoto irão possuir a mesma chave de criptografia E. O canal de comunicação não precisa ser protegido, já que os dados que trafegarão neste canal estarão cifrados.

O servidor remoto também deve verificar a identidade do assinante do enclave (MRSIGNER) antes de encaminhar a este quaisquer dados confidenciais, já que na instanciação do enclave apenas é verificado o registrador MRENCLAVE. Dessa forma, qualquer pessoa poderia modificar um enclave e assiná-lo novamente. Além disso, o provedor de serviços também deve verificar os outros atributos do enclave, para evitar que sejam divulgados dados confidenciais a enclaves de teste ou que estão em modo de depuração, por exemplo.

Após estabelecer um canal seguro para a comunicação, os dados confidenciais podem ser provisionados para o enclave, cifrando os dados seguros S com uma chave E e enviando, assim, $E(S)$ para o enclave, que utilizará a mesma chave E para efetuar a decifragem de $E(S)$, obtendo novamente os dados S. No entanto, pode ser inconveniente exigir que o enclave se conecte a uma entidade remota toda vez que ele necessitar os mesmos dados e, para evitar este tipo de situação, o enclave pode armazenar os dados obtidos em uma memória não volátil, selando estes dados (conforme descrito na Seção 2.3.5) de forma que ficarão acessíveis somente ao enclave que os selou, e somente na plataforma em que eles foram selados [Intel 2016a].

2.4. Casos de Uso da Arquitetura SGX

Apesar da arquitetura SGX ser relativamente nova, já que foi de fato lançada no mercado em outubro de 2015, juntamente com os processadores Intel Core de 6^a geração (*Skylake*), desde o seu anúncio vários trabalhos foram sendo desenvolvidos de forma a validar a proposta. Um desses trabalhos, descrito por [Jain et al. 2016], apresenta a proposta de emulação das instruções adicionadas à SGX, utilizando um emulador de processador *open-source*, o QEMU. O emulador desenvolvido não se restringe apenas à simulação de

hardware, mas também de componentes do sistema operacional, além de fornecer uma biblioteca que permite a depuração e o monitoramento do desempenho das aplicações.

Em [Hoekstra et al. 2013] são apresentados três protótipos de aplicações que se valem dos recursos fornecidos pela SGX. O primeiro protótipo é de uma aplicação do tipo OTP (*One-Time Password*), que tem por finalidade prover uma chave de acesso que poderá ser utilizada apenas uma vez, por exemplo em transações financeiras. O segundo protótipo apresenta um ERM (*Enterprise Rights Management*), que tem como finalidade manter a integridade e controlar o acesso a documentos sigilosos, sendo este protótipo desenvolvido para o Departamento de Segurança Interna dos EUA. O terceiro protótipo desenvolvido se refere a um sistema de videoconferência, que faz uso da SGX e do PAVP (*Protected Audio and Video Path*), de forma a manter o sigilo dos dados de áudio e vídeo em trânsito.

Outros trabalhos também fazem uso da arquitetura SGX para prover segurança a aplicações que são executadas em ambientes de nuvem pública. Um exemplo é apresentado em [Baumann et al. 2015], que descreve um protótipo que permite encapsular aplicações legadas dentro de um enclave, sem a necessidade de reescrevê-las. Esta abordagem pode ser utilizada tanto para aplicações simples, como também para máquinas virtuais executadas em ambientes *IaaS*. [Brenner et al. 2017] também apresenta o uso do SGX para encapsular a execução de micro serviços em ambientes na nuvem. Outros trabalhos que fazem uso da arquitetura SGX em ambientes distribuídos são descritos em [Schuster et al. 2015] e [Shih et al. 2016].

[Richter et al. 2016] utilizam a arquitetura SGX para isolar módulos do *kernel* do Linux em enclaves, evitando que uma falha em um módulo do *kernel* se propague ao restante do sistema. Para isso, partes do módulo são migradas para o *user space*, visto que os enclaves não podem ser executados em modo núcleo. Já [Tsai et al. 2017] e [Tian et al. 2017] apresentam um sistema operacional, constituído por uma biblioteca (ou seja, um *Library OS*), que é executado dentro de um enclave.

Os trabalhos [Brekalo et al. 2016] e [Mofrad et al. 2017] propõem o uso do SGX para auxiliar na segurança de bancos de senhas e na criação de uma biblioteca de criptografia, respectivamente. O ponto em comum nos dois trabalhos é a utilização de enclaves para a geração e manutenção das chaves de criptografia a serem utilizadas. Por sua vez, [Karande et al. 2017] faz uso do SGX para manter a integridade e confidencialidade dos dados em *logs* gerados pelo sistema Linux.

Por fim, [Lind et al. 2017] apresentam um *framework* que utiliza diretivas de pré-compilação para particionar automaticamente aplicações escritas em linguagem C em seus componentes confiável e não confiável.

2.5. Limitações, Questões em Aberto e Vulnerabilidades do SGX

Apesar da arquitetura SGX prover mecanismos eficientes para garantir a segurança dos dados de uma aplicação, ainda há algumas questões a serem consideradas com maior atenção.

Uma das limitações da arquitetura SGX diz respeito ao uso da CPU para determinar a chave utilizada para a selagem dos dados. Isso implica que, no caso de substituição da CPU, seja por opção do usuário ou por problemas técnicos, o enclave não estará mais apto

a abrir os dados previamente selados por ele. Essa é uma questão também importante quando se trata de questões de balanceamento de carga em servidores, seja na utilização de múltiplos servidores ou de servidores com múltiplos processadores. Em virtude desta limitação, ainda não há disponíveis no mercado processadores multi-*socket* com arquitetura SGX. O Intel *Software Guard Extensions Developer Guide* [Intel 2016a] também coloca que, na necessidade de efetuar a migração de dados selados para outro computador, isso deverá ser efetuado utilizando o processo de atestação remota entre ambos os enclaves.

Outro ponto, descrito por [Hoekstra et al. 2013], diz respeito à entrada de dados das aplicações. Os usuários, inevitavelmente, terão que informar dados em algum momento para as aplicações que estão protegidas em seus enclaves, mas os dispositivos de entrada de dados (como teclados, câmeras de vídeo, microfones, entre outros) não estão preparados para fornecer esses dados ao sistema de forma segura, o que possibilita sua interceptação antes que eles cheguem à segurança do enclave. Vale ressaltar também que, assim como os dispositivos de entrada, tem-se também os dispositivos de saída que, da mesma forma, não estão preparados para fornecer a segurança adequada no tratamento dos dados provenientes dos enclaves.

[Davenport and Ford 2014] também apontam uma preocupação quanto ao uso do SGX como um aliado à execução de *malwares*, visto que a Intel não fez nenhuma restrição a quais aplicações poderiam se utilizar da segurança dos enclaves. [Schwarz et al. 2017] implementaram um ataque de canal lateral ao enclave através de um *malware* que, por executar dentro de outro enclave, fica protegido pelo isolamento provido pela arquitetura SGX e não pode ser identificado por anti-virus ou analisado pelo sistema operacional. Procedimentos forenses também ficam limitados devido à proteção da memória.

O Intel SGX não considera ataques por canal lateral nem ataques de engenharia reversa em seu modelo de ameaças. O Intel *Software Guard Extensions Developer Guide* [Intel 2016a] ressalta que cabe aos desenvolvedores construir enclaves resistentes a estes tipos de ataque. O *malware* de [Schwarz et al. 2017] é um ataque de canal lateral à memória *cache* do tipo *Prime and Probe*, capaz de extrair uma chave de um processamento RSA ocorrendo dentro do enclave da vítima. Este é o tipo de ataque mais comum ao enclave SGX encontrado até o momento. [Moghimi et al. 2017] utiliza este mesmo método para extrair chaves AES de um processamento do enclave e [Brasser et al. 2017] conseguiu, além de extrair uma chave RSA, detectar sequências específicas do genoma humano durante a indexação genômica ocorrendo dentro do enclave.

Nestes ataques o enclave da vítima e o *malware* executam em paralelo na mesma máquina física. Apesar do enclave estar protegido criptograficamente na EPC, na *cache* os dados estão em claro. Além disso, é o *software* do sistema operacional que gerencia a tradução dos endereços da *cache*, que por sua vez é compartilhada com os demais *softwares* em execução no computador. Desta forma, apesar do atacante não conseguir acesso direto ao conteúdo do *cache* usado pelo enclave, ele pode aplicar a técnica conhecida como *Prime and Probe* para inferir segredos do enclave.

Para obter sucesso com a técnica *Prime and Probe*, pelo menos dois requisitos devem ser atendidos: o *malware* deve ser capaz de sobrescrever determinados endereços da *cache* usados pela vítima, e também deve ser capaz de realizar medição de tempo com resolução boa o suficiente para distinguir *hits* e *misses* na *cache*. Em linhas gerais, no

primeiro momento o atacante preenche a *cache* monitorada com seus dados. Então aguarda que a vítima execute algum código cujo acesso à memória é dependente da informação secreta (a chave por exemplo). Na sequencia o atacante faz uma requisição a dados seus e mede o tempo para recuperá-los. Este tempo determina se o dado se encontrava ou não na *cache* (*hit* ou *miss*). No caso de *miss*, sabe-se que aquele endereço foi utilizado pela vítima. Então o atacante se aproveita da dependência do acesso à memória para inferir os *bits* correspondentes ao segredo.

Em nível de *software*, no entanto, várias bibliotecas criptográficas estão sendo robustecidas especificamente para evitar ataques ao *cache*. Para cada acesso à memória dependente de segredos, o enclave pode gerar um conjunto de acessos à memória que irá se manifestar em mudanças em todos os conjuntos da *cache* monitorados, escondendo o endereço de memória efetivamente acessado do adversário. Outras abordagens buscam eliminar as dependências entre o segredo e o acesso à memória das implementações, o que também é efetivo para evitar este tipo de ataque.

[Costan and Devadas 2016] ressalta outras vulnerabilidades, como ataques por monitoramento do consumo de energia, cujas variações podem gerar informações suficientes para que o atacante infira a sequência de instruções sendo executadas, ou ainda, ataques passivos de tradução de endereçamento que podem dar informações do padrão de acesso à memória com a granularidade de páginas. A Intel menciona que o modelo de ameaças ao SGX considera que o atacante pode ter acesso à *flash* que armazena o *firmware* do computador, mas segundo [Costan and Devadas 2016], a documentação oficial não aborda as implicações decorrentes deste fato. No entanto, até o presente momento, não foram encontrados trabalhos demonstrando o uso destas vulnerabilidades para efetivamente descobrir segredos do enclave.

Por fim, além da Intel ter de ser considerada confiável, como, por exemplo, no provimento da PROVISIONKEY utilizada em enclaves que implementam a atestação, [Costan and Devadas 2016] questiona a real necessidade do *Launch Enclave* (LE), que é um enclave emitido pela Intel e responsável por aprovar todos os enclaves antes da inicialização. Segundo os autores, o LE poderia funcionar como um mecanismo de licenciamento, permitindo à Intel se posicionar como um intermediário na distribuição de todo *software* SGX.

2.6. SGX na Prática: SGX SDK

Esta seção tem o objetivo de descrever os detalhes para a construção de aplicações utilizando a arquitetura Intel SGX. Para isto, faz-se o uso do SDK (*Software Development Kit*) do SGX para o sistema operacional Linux, com implementações em linguagem C [Intel 2016b]. Os exemplos apresentados nesta seção estão disponíveis na íntegra em https://github.com/newtoncw/sbseg-sgx, já as bibliotecas necessárias para a execução dos exemplos, bem como as instruções para sua instalação, estão disponíveis em https://github.com/01org/linux-sgx.

2.6.1. A Linguagem de Definição do Enclave e a Ferramenta *Edger8r*

A definição da interface do enclave (funções ECALLs e OCALLs) é feita através da configuração dos arquivos EDL (*Enclave Definition Language*), que definem também os

tipos de dados que serão suportados pelo enclave. A Listagem 2.1 traz um exemplo de configuração de um arquivo EDL.

Listagem 2.1. Exemplo de configuração de arquivo EDL (adaptado de [Intel 2016b]).

```
1  enclave {
2      from "file1.edl" import *;
3      from "file2.edl" import foo, bar;
4
5      include "string.h"
6      include "mytypes.h"
7
8      struct mysecret {
9          int key;
10         const char* text;
11     };
12
13     enum boolean { FALSE = 0, TRUE = 1 };
14
15     trusted {
16         include "trusted.h"
17         public void set_secret([in] struct mysecret* psecret);
18         void some_private_func(enum boolean b);
19     };
20
21     untrusted {
22         include "untrusted.h"
23         void ocall_print();
24         int another_ocall([in] struct mysecret* psecret) allow(some_private_func);
25     };
26 };
```

O arquivo EDL é dividido em duas seções: a seção `trusted` define as ECALLs, enquanto a seção `unstrusted` define as OCALLs. Como descrito nas seções anteriores, uma ECALL define um ponto de entrada para o enclave, e uma OCALL define uma transferência do controle do enclave para a aplicação, para a realização de *systems calls* ou operações de E/S. As OCALLs também podem ser utilizadas para efetuar a transferência de dados entre o enclave e a aplicação, sendo que estas são opcionais. Em contrapartida, deve ser definida pelo menos uma chamada ECALL pública, já que estas representam as entradas no enclave. No exemplo apresentado na Listagem 2.1, tem-se apenas a função definida na linha 17 como ponto de entrada para o enclave. A função definida na linha 18 pode ser chamada apenas por uma função OCALL. Na seção `untrusted` estão contidas as definições das funções OCALLs, sendo que a função definida na linha 23 não pode efetuar chamadas de funções ECALL. Já a função definida na linha 24 pode invocar a função ECALL definida na linha 18.

Um arquivo EDL pode, opcionalmente, importar funções de outros arquivos EDL, fazendo isso de uma forma global ou seletiva. Para importar todas as funções de outro arquivo EDL utiliza-se a sintaxe definida na linha 2 da Listagem 2.1 e, para importar determinadas funções de outro arquivo EDL, é necessário informar a lista de funções a serem importadas, separando seus nomes por vírgula, como definido na linha 3.

Pode-se também efetuar a inclusão de arquivos de cabeçalho contendo definições de funções em C. As definições contidas nos arquivos de cabeçalho podem ser inseridas apenas para o contexto confiável da aplicação, fazendo sua inclusão na seção `trusted` (linha 15), apenas para o contexto não confiável, incluindo na seção `untrusted` (linha 21), ou para ambos os contextos, sendo incluídas na seção `enclave` (linhas 5 e 6). Além disso, é possível definir os tipos de dados que poderão ser utilizados pelo enclave, como

exemplificado nas linhas 8 a 13.

O arquivo também EDL provê suporte à definição de atributos de direção para os parâmetros: `[in]`, `[out]` ou `[user_check]`. O atributo `[in]`, quando utilizado em um ECALL, representa que o parâmetro será passado da aplicação para o enclave. Quando ele é utilizado em um OCALL, define que o parâmetro será retornado do enclave para a aplicação. Já o atributo `[out]` em uma função ECALL define que o parâmetro é retornado do enclave para a aplicação, e quando utilizado em um OCALL, ele define que o parâmetro é passado da aplicação para o enclave. Os atributos `[in]` e `[out]` podem ser utilizados de forma combinada, permitindo que o parâmetro seja transferido em ambas as direções.

Por fim, o atributo `[user_check]` é utilizado em casos onde as demandas de comunicação de dados entre o enclave e a aplicação exigem que as restrições impostas pelos parâmetros `[in]` e `[out]` sejam ignoradas. Isto pode ocorrer, por exemplo, quando um *buffer* é muito grande para residir totalmente na memória do enclave, sendo necessário fragmentá-lo em blocos menores e processá-lo através de uma série de chamadas ECALL, ou em situações onde a aplicação passa um ponteiro como parâmetro do ECALL. A utilização do atributo `[user_check]` implica em maiores riscos de segurança e, em virtude disso, devem ser adicionadas verificações adicionais no código para evitar o vazamento de informações confidenciais do enclave.

O arquivo EDL é utilizado pela ferramenta *Edger8r* para gerar a interface de comunicação entre o enclave e a parte não confiável da aplicação, ou seja, as rotinas de borda (ou *proxies*) entre o código confiável e o código confiável. A execução de *Edger8r* gera dois arquivos-fonte e dois arquivos cabeçalho para cada arquivo EDL, conforme apresentado na Figura 2.11. O arquivo-fonte `enclave_u.c` contém as rotinas que serão as portas de entrada para o enclave (funções ECALLs), efetuando as validações necessárias de acordo com os atributos de entrada e saída que foram definidos para cada um dos parâmetros. Já o arquivo-fonte `enclave_t.c` contém as rotinas que serão as portas de saída do enclave (funções OCALLs). Os arquivos de cabeçalho `enclave_u.h` e `enclave_t.h` devem ser incluídos no código não confiável da aplicação e no enclave, respectivamente, para que as rotinas de entrada e saída possam ser utilizados.

A ferramenta *Edger8r* pode ser executada como parte do processo de construção da aplicação ou de forma isolada. É importante destacar que sua execução deve ser feita em um ambiente protegido e livre de aplicações maliciosas, para garantir a integridade do código gerado. Além disso, o código contido no enclave é responsabilidade do desenvolvedor, portanto este deve revisar o código gerado pelo *Edger8r*.

2.6.2. O Arquivo de Configuração do Enclave

O *Enclave Configuration File* é um arquivo XML que faz parte do projeto, onde o programador pode definir alguns parâmetros do enclave. Este arquivo é utilizado pela ferramenta *SGX Sign* para efetuar a assinatura do enclave. Um exemplo deste arquivo de configuração é mostrado na Listagem 2.2. Todos os elementos contidos no arquivo são opcionais, caso algum dos elementos não esteja presente, ou se o arquivo não for criado, serão utilizados os valores padrões no processo de assinatura do enclave.

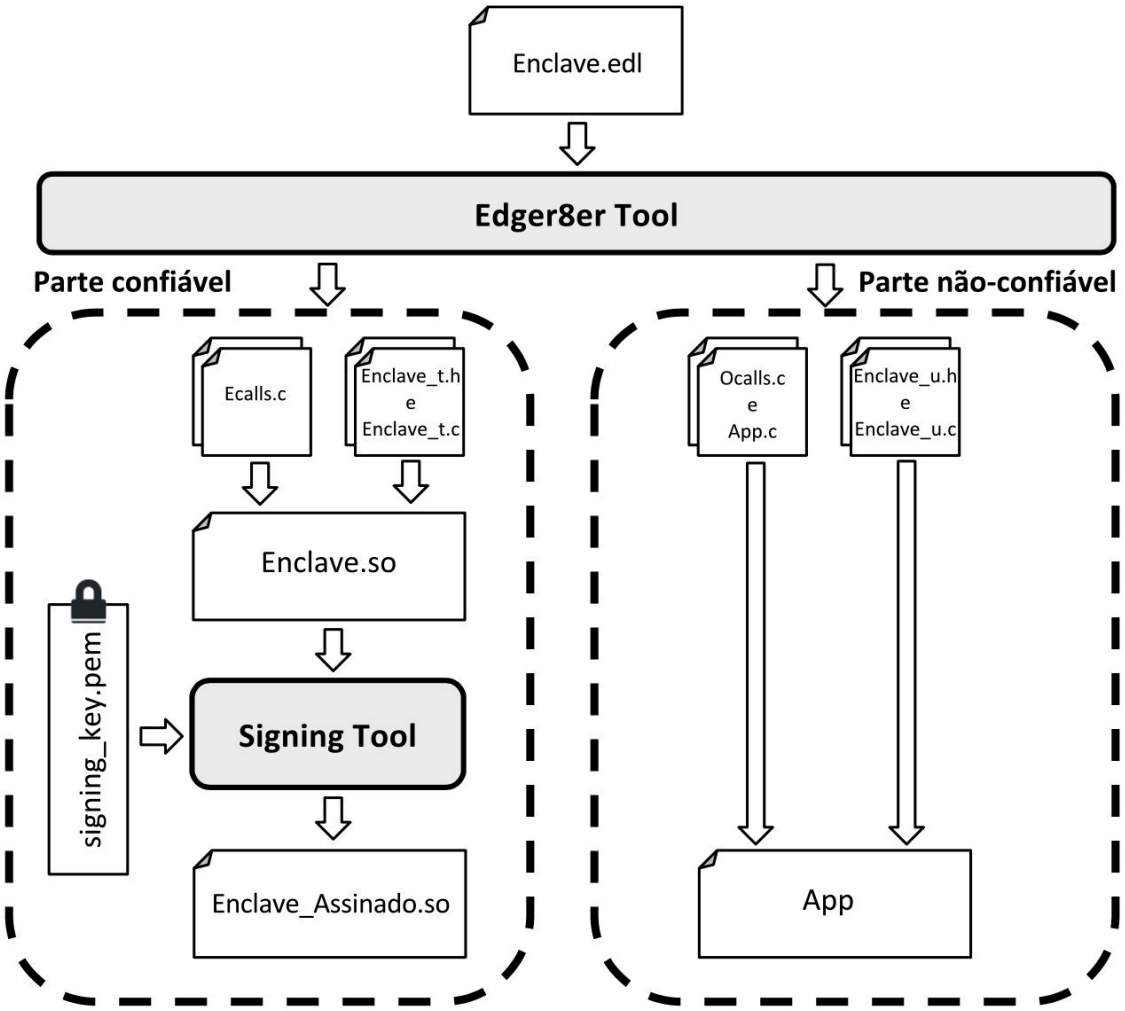

Figura 2.11. Processamento do arquivo EDL para a geração das interfaces das funções de borda do enclave.

Listagem 2.2. Arquivo de configuração do enclave (adaptado de [Intel 2016b]).

```
1  <EnclaveConfiguration>
2      <ProdID>1</ProdID>
3      <ISVSVN>1</ISVSVN>
4      <TCSNum>1</TCSNum>
5      <TCSPolicy>1</TCSPolicy>
6      <DisableDebug>0</DisableDebug>
7      <StackMaxSize>0x50000</StackMaxSize>
8      <HeapMaxSize>0x100000</HeapMaxSize>
9      <MiscSelect>0</MiscSelect>
10     <MiscMask>0xFFFFFFFF</MiscMask>
11 </EnclaveConfiguration>
```

Os elementos `ProdID` e `ISVSVN` representam, respectivamente, o ID do produto e o *Secure Version Number*, sendo atribuídos pelo desenvolvedor, e ambos têm seu valor padrão como zero. Cada *thread* em execução no enclave é associado à um *Thread Context Structures* (TCS). O elemento `TCSNum` define a quantidade de TCS , sendo que este deve ser maior que zero, e tem seu valor padrão como 1. O elemento `TCSPolicy`, quando atribuído com zero, indica que o TCS estará limitado ao contexto não confiável e, quando

atribuído com valor 1, indica que o TCS não estará limitado ao contexto não confiável, sendo este o seu valor padrão. O elemento `DisableDebug` indica a possibilidade de o enclave ser depurado; quando atribuído com zero, seu valor padrão, o enclave pode ser depurado e, caso atribuído com 1, a depuração não é permitida.

O elemento `StackMaxSize` define o tamanho máximo da pilha por *thread* e o elemento `HeapMaxSize` define o tamanho máximo do *heap* para o processo. Ambos devem ser definidos sempre com valores múltiplo de $4KB$, e seus valores padrão são $0x40000$ e $0x100000$, respectivamente. Caso não haja tamanho suficiente na pilha do enclave, uma chamada ECALL irá retornar o código de erro `SGX_ERROR_STACK_OVERRUN`, que indica que o valor do elemento `StackMaxSize` deve ser ajustado. Para evitar esse tipo de problema, os valores destes dois parâmetros podem ser ajustados utilizando o *Enclave Memory Measurement Tool*, que efetua a medição do uso real de memória por um enclave em tempo de execução, retornando o pico de uso da pilha e do *heap* de memória.

Por fim, os elementos `MiscSelect` e `MiscMask` são reservados para extensões futuras, tendo seus valores definidos com 0 e $0xFFFFFFFF$, respectivamente. Caso os valores sejam definidos de outra forma, o enclave pode não ser carregado corretamente.

2.6.3. Assinando o Enclave: A Ferramenta de Assinatura de Enclave

Conforme explicado na seção 2.3.3, assinar um enclave é um processo que envolve a criação de uma estrutura de assinatura com propriedades do enclave. A estrutura de assinatura do enclave é composta de 4 seções: cabeçalho, assinatura, corpo e *buffer* (os detalhes de cada seção podem ser encontrados em [Intel 2016b]. Esta estrutura permite à arquitetura SGX detectar quaisquer adulterações do enclave, e assim provar que o mesmo é legítimo e que foi carregado corretamente.

O *Software Development Kit* (SDK) fornecido pela Intel inclui uma ferramenta, chamada *Enclave Signing Tool*, que realiza o processo de assinatura. Esta ferramenta também avalia a imagem do enclave em busca de potenciais erros e problemas de segurança. Ao ser carregado, a assinatura é conferida para confirmar a integridade do enclave.

São admitidos dois métodos de assinatura: em passo único, usando a chave privada do usuário, ou em dois passos, usando uma ferramenta de assinatura externa. O método de dois passos protege a chave de assinatura em um ambiente separado e deve ser o método utilizado nas versões de produção da aplicação.

Para executar a ferramenta deve ser utilizado o comando com a seguinte sintaxe:

```
# sgx_sign <comando> [argumentos]
```

Os comandos possíveis são:

- `sign`: Assina o enclave em um passo único. Este comando deve receber os seguintes argumentos: `-enclave [arquivo]`, especificando o enclave a ser assinado; `-key [arquivo]`, especificando a chave de assinatura; e `-out [arquivo]`, especificando o arquivo retornado. Além destes argumentos, `-config [arquivo]`, que aponta o arquivo de configuração, também pode ser utilizado.

- `gendata`: É o primeiro passo da assinatura em dois passos. Gera o material a ser assinado pela ferramenta externa que consiste em seções de cabeçalho e corpo da estrutura de assinatura do enclave. A saída é um arquivo de 256 bytes. Este comando deve receber os argumentos `-enclave [arquivo]`, `-out [arquivo]` e, opcionalmente, `-config [arquivo]`.

- `catsig`: É o segundo passo da assinatura em dois passos. Gera o enclave assinado com a entrada de assinatura e a chave pública. A entrada de assinatura é gerada por uma ferramenta externa baseada nos dados gerados no passo anterior. Aqui são geradas as seções de assinatura e *buffer* da estrutura de assinatura do enclave.Este comando deve receber os argumentos `-enclave [arquivo]`, `-key [arquivo]`, `-out [arquivo]`, `-sig [arquivo]`, `-unisgned [arquivo]` e, opcionalmente, `-config [arquivo]`. O argumento `-sig` aponta o arquivo contendo a assinatura correspondente ao material de assinatura do enclave e o argumento `-unsigned` aponta o arquivo contendo o material gerado pelo comando `gendata`.

Quando um projeto de enclave é criado pela primeira vez, o usuário deve escolher entre usar uma chave de assinatura existente ou gerar uma chave automaticamente. Quando se optar por utilizar uma chave existente, ela deve estar no formato PEM e não deve estar cifrada. A assinatura em passo único utilizando uma chave existente é efetuada através do seguinte comando:

```
# sgx_sign sign -enclave enclave.so -config config.xml \
                -out enclave_assinado.so -key chave.pem
```

Os enclaves assinados em passo único só podem ser iniciado em modo *debug* ou *prerelease*, sendo que este processo de assinatura não pode ser utilizado para gerar enclaves em modo de produção.

Para o modo de produção, o SGX fornece um esquema de assinatura em dois passos. O primeiro passo ocorre ao final da compilação do enclave, quando a ferramenta de assinatura gera o material de assinatura do enclave. O comando para gerar o material de assinatura do enclave é o seguinte:

```
# sgx_sign gendata -enclave enclave.so -config config.xml \
                   -out enclave_hash.hex
```

O desenvolvedor deve levar o arquivo resultante (`enclave_hash.hex`) a uma plataforma de assinatura externa, onde a chave privada é armazenada, e assinar o arquivo de *hash* com o material de assinatura, e então levar o arquivo de assinatura resultante de volta para a plataforma de compilação.

No segundo passo, o desenvolvedor executa a ferramenta de assinatura com o comando `catsig` fornecendo as informações necessárias na linha de comando para adicionar o *hash* da chave pública e a assinatura à seção de metadados do enclave. Um exemplo do comando para o segundo passo de assinatura é dado a seguir:

```
# sgx_sign catsig -enclave enclave.so -config config.xml \
                  -out enclave_assinado.so -key chave_publica.pem \
                  -sig assinatura.hex -unsigned enclave_hash.hex
```

O arquivo `config.xml` pode ser omitido, sendo que, neste caso, as configurações padrão serão utilizadas.

O processo de assinatura em dois passos protege a chave de assinatura em uma plataforma separada. Este é o método de assinatura padrão, e o único método para assinar enclaves em modo de produção.

2.6.4. Bibliotecas Confiáveis e Não-Confiáveis do SDK

As bibliotecas confiáveis são bibliotecas estáticas desenvolvidas para serem ligadas ao enclave. Estas bibliotecas não podem usar instruções não suportadas pela arquitetura SGX e devem passar por um processo de revisão mais rigoroso que uma biblioteca convencional. Para desenvolver uma biblioteca confiável, deve-se gerar um arquivo `.lib` ao invés de um arquivo `.so`. Se um subconjunto da biblioteca fizer parte da interface do enclave com o domínio não-confiável, ele deverá ser declarado no arquivo EDL.

A ferramenta `sgx_edger8er` acrescenta automaticamente o nome do enclave nas funções de borda para evitar colisão dos nomes das funções, mas o desenvolvedor deve garantir a unicidade do nome das ECALLs. No entanto, quando dois enclaves importam a mesma ECALL de uma biblioteca confiável, o conjunto de funções de borda para cada enclave vão conter os mesmos nomes de funções *proxy* (de interface) não-confiáveis, o que irá gerar um erro. Para solucionar este problema, a ferramenta `sgx_edger8er` deve ser usada com a opção `-use-prefix` e acrescentar o prefixo com o nome do enclave nas ECALLs do código não-confiável.

A segurança do enclave depende na capacidade de se obter medição precisa de todo código e dados colocados dentro do enclave. Desta forma, enclaves não devem ser ligados dinamicamente em hipótese alguma. Ligações estáticas são permitidas.

A seguir serão relacionadas algumas das bibliotecas confiáveis e não-confiáveis disponibilizadas no SDK SGX, com a descrição de suas principais funcionalidades. Algumas delas são de uso obrigatório, ou seja, devem ser usadas nas aplicações envolvendo enclaves. A descrição detalhada de todas as funções disponíveis nas bibliotecas SGX e casos de uso podem ser encontrados em [Intel 2016b].

As seguintes bibliotecas são consideradas confiáveis; suas funções somente podem ser chamadas a partir de aplicações executando dentro do enclave:

- `libsgx_trts.a` (obrigatória): *Trusted Runtime System*. É uma biblioteca-chave do SDK. Fornece a lógica do ponto de entrada no enclave. Conta com funções auxiliares para determinar se um um dado endereço está, ou não, na região do enclave. Também é responsável pelo gerenciamento das interrupções e fornece um envelope para a instrução RDRAND, que retorna um número realmente aleatório gerado pelo hardware (`sgx_read_rand`).
- `libsgx_tstdc.a` (obrigatória): Biblioteca C padrão (*math*, *string*, etc). Algumas

funções não são suportadas, nos casos em que suas definições são inseguras (função `strcpy`), implicam em uso de instruções de CPU restritas, a implementação é muito grande para o enclave, precisam de informação do domínio não-confiável, ou a função não faz parte do padrão ou não é suportada por compilador específico.

- `libsgx_tstdcxx.a` (opcional): Biblioteca C++ em conformidade com o C++ 03 (incluindo STL). O suporte apresenta algumas limitações, como a impossibilidade de passar objetos pelos limites do enclave e de usar destrutores globais.

- `libsgx_tservice.a` (obrigatória): Esta biblioteca traz funcionalidades para dar suporte à manipulação segura e proteção de dados. Destacam-se as funcionalidades de criptografia (selagem e deselagem), estabelecimento de sessão Diffie-Hellman, suporte arquitetural ao enclave e funções relacionadas à atestação.

- `libsgx_tcrypto.a` (obrigatória): A biblioteca criptográfica conta com algoritmos como SHA256, Rijndael 128 CGM, CMAC 128, AES, ECC256 e ECDSA, embora com funcionalidades ainda bastante limitadas. Caso seja necessário usar outras funcionalidades, uma biblioteca criptográfica própria deve ser desenvolvida.

- `libsgx_tkey_exchange.a` (opcional): Biblioteca de troca de chaves confiável, que permite ao desenvolvedor trocar segredos entre servidores e enclaves. É usada em conjunto com a biblioteca de troca de chaves não-confiável.

As seguintes bibliotecas são consideradas não-confiáveis; suas funções somente podem ser chamadas a partir de aplicações executando fora do enclave:

- `libsgx_urts.so` (obrigatória): Fornece as funcionalidades necessárias para que as aplicações possam gerenciar os enclaves. Traz funções como `sgx_create_enclave` e `sgx_destroy_enclave`.

- `libsgx_uae_service.so` (obrigatória): Fornece acesso às funcionalidades dos enclaves arquiteturais (AEs), tanto aos enclaves quanto às aplicações não confiáveis.

- `libsgx_ukey_exchange.a` (opcional): Biblioteca de troca de chaves não-confiável.

Por fim, vale ressaltar que também existem bibliotecas como a `libsgx_trts_sim.a` e `libsgx_urts_sim.a`, que simulam as instruções SGX em software e podem ser usadas em plataformas sem suporte ao SGX, para fins de estudo e desenvolvimento.

2.6.5. Criando um Enclave

Esta seção demonstra detalhadamente como efetuar a criação de um enclave utilizando os métodos providos pelo SGX SDK, além de apresentar um exemplo simples da execução de chamadas ECALL e OCALL.

O projeto está dividido em três arquivos fonte (`app.c`, `lib_ocalls.c`, `lib_ecalls.c`), além do arquivo EDL para a geração das rotinas de borda. A Figura 2.12 relaciona as dependências dos arquivos fonte dentro da arquitetura da aplicação e esquematiza o seu fluxo de execução.

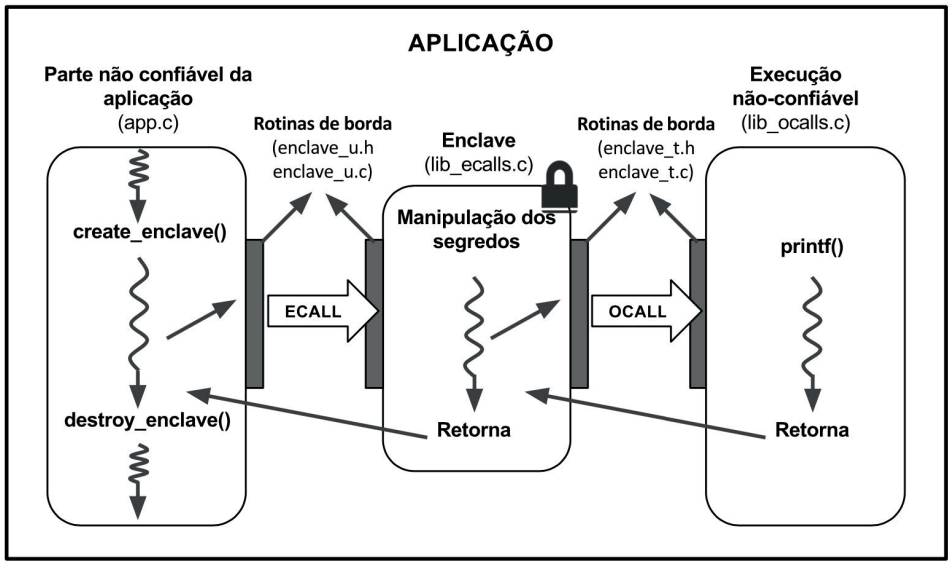

Figura 2.12. Composição e fluxo de execução da aplicação SGX.

O arquivo fonte `app.c` é apresentado na Listagem 2.3, contendo a função `main` e as operações necessárias para a criação do enclave, entrada no enclave, e destruição do mesmo.

Na linha 16 é efetuada a chamada da função `sgx_create_enclave`, responsável por carregar e inicializar o enclave. Para isto, utiliza-se a biblioteca do enclave (`ENCLAVE_FILENAME`), que deve estar assinada pela *SGX Sign Tool*, e um *token* que é criado na primeira inicialização do enclave, podendo ser armazenado para as inicializações seguintes do mesmo enclave, resultando em um ganho de desempenho. O parâmetro `SGX_DEBUG_FLAG` é provido pelo próprio SDK e pode ter seu valor atribuído no momento de compilação, sendo que o valor 0 indica que o enclave será inicializado em modo *release*, não permitindo que o mesmo seja depurado; já o valor 1 permite a depuração do enclave. O parâmetro `updated` retorna o valor 1 caso o `token` tenha sido atualizado na criação do enclave. Já o parâmetro `eid` retorna o identificador do enclave, necessário para a execução das funções ECALL.

Listagem 2.3. Arquivo fonte *app.c*.

```
1  #include "stdio.h"
2  #include "string.h"
3  #include "sgx_eid.h"
4  #include "sgx_urts.h"
5  #include "Enclave_u.h" //Gerado pelo Edger8r
6
7  #define ENCLAVE_FILENAME "enclave.signed.so"
8
9  int main(int argc, char *argv[]) {
10     char *data = "Hello World!";
11     sgx_enclave_id_t eid = 0;
12     sgx_launch_token_t token = {0};
```

```
13      sgx_status_t ret = SGX_ERROR_UNEXPECTED;
14      int updated = 0;
15
16      ret = sgx_create_enclave(ENCLAVE_FILENAME, SGX_DEBUG_FLAG, &token, &updated, &eid,
17          NULL);
18
19      if (ret != SGX_SUCCESS) {
20          printf("Erro na inicializacao do enclave ...\n");
21          return -1;
22      }
23
24      if (ecall_teste(eid, data) != SGX_SUCCESS) {
25          printf("ERRO na execucao da ecall ...\n");
26      } else {
27          printf("SUCESSO na execucao da ecall ...\n");
28      }
29
30      if(sgx_destroy_enclave(eid) != SGX_SUCCESS) {
31          printf("ERRO ao destruir o enclave.\n");
32          return -1;
33      }
34
35      return 0;
36  }
```

Na linha 24 é efetuada a chamada para a função ECALL (`ecall_teste`) passando por parâmetro o identificador do enclave. A função chamada neste ponto é gerada pela ferramenta *Edger8r*, ficando contida no arquivo fonte `Enclave_u.c` (conforme descrito no diagrama da Figura 2.11 da Seção 2.6.1) e contém as validações necessárias para efetuar a entrada no enclave. Após validados os parâmetros, a função `ecall_teste` contida no arquivo fonte `lib_ecalls.c`, e apresentada na Listagem 2.4, é chamada. Essa função recebe a mensagem passada por parâmetro e cria um *hash* de 256 *bits* a partir dessa mensagem, utilizando a função `sgx_sha256_msg` provida pelo SDK. Caso a criação do *hash* ocorra com sucesso, o mesmo é mostrado em tela, através da execução da função `ocall_print`.

Listagem 2.4. Arquivo fonte *lib_ecalls.c*.

```
1   #include "string.h"
2   #include "sgx_tcrypto.h"
3   #include "Enclave_t.h" //Gerado pelo Edger8r
4
5   void ecall_teste(char *c) {
6       sgx_status_t ret;
7       sgx_sha256_hash_t hash;
8
9       ret = sgx_sha256_msg((const uint8_t *)c, strlen(c), &hash);
10
11      if (ret == SGX_SUCCESS) {
12          ocall_print("sgx_sha256_msg SUCESSO");
13          ocall_print((char*)hash);
14      } else {
15          ocall_print("sgx_sha256_msg ERRO");
16      }
17  }
```

A função `ocall_print` está contida no arquivo fonte `lib_ocalls.c`, que é apresentado na Listagem 2.5. Essa função apenas faz a impressão na tela da mensagem recebida por parâmetro. A execução de uma função OCALL para realizar essa tarefa se faz necessária, já que o enclave não pode executar *system calls*. Da mesma forma que ocorre na execução de uma chamada ECALL, os parâmetros passados para a função também são

validados em uma função homônima contida no arquivo fonte `Enclave_t.c` gerado pela ferramenta *Edger8r*.

Listagem 2.5. Arquivo fonte *lib_ocalls.c*.

```
1 #include "stdio.h"
2 #include "Enclave_u.h" //Gerado pelo Edger8r
3
4 void ocall_print(char *c) {
5     printf("%s\n", c);
6 }
```

Quando o enclave não for mais necessário, deve-se destruí-lo executando a função `sgx_destroy_enclave`, que recebe por parâmetro o identificador do enclave a ser destruído.

Por fim, a Listagem 2.6 apresenta o conteúdo do arquivo EDL, que é utilizado pela ferramenta *Edger8r* para gerar as rotinas que irão efetuar as validações dos parâmetros passados, conforme os atributos utilizados. Nas ECALLs, o atributo `[in]` indica a entrada de parâmetro no enclave, enquanto na OCALLs ele é utilizado para indicar a entrada de parâmetro no universo não-confiável. Em ambos os casos, o atributo `[string]` indica que o parâmetro é uma cadeia de caracteres terminada com *NULL*. A rotina de borda primeiro determina o tamanho da string e, após isso, copia o seu conteúdo para o enclave (quando utilizada em uma ECALL), para inserir o terminador *NULL*. O atributo `[string]` pode ser utilizado em conjunto com o atributo `[in]` ou em uma combinação dos atributos `[in]` e `[out]`, nunca com o atributo `[out]` apenas.

Listagem 2.6. Arquivo EDL contendo as definições das funções ECALLs e OCALLs.

```
1 enclave {
2     trusted {
3         public void ecall_teste([in, string] char *c);
4     };
5
6     untrusted {
7         void ocall_print([in, string] char *c);
8     };
9 };
```

2.6.6. Selando Dados

Esta seção é mostra como pode-se efetuar a selagem de dados através de um enclave SGX, bem como a abertura posterior dos dados selados. O exemplo aqui apresentado é divido em dois arquivos fonte (`app.c` e `lib_ecalls.c`), não tendo nenhuma função OCALL disponível.

Este exemplo tem a seguinte sequência: primeiro é efetuada a criação do enclave, em seguida é realizada uma ECALL passando os dados que deverão ser selados. Os dados selados pelo enclave são retornados à aplicação, e, por sua vez, são passados para uma segunda ECALL que abrirá novamente estes dados e retornará os mesmos para a aplicação.

O arquivo fonte `app.c` é apresentado na Listagem 2.7, tendo uma estrutura semelhante ao apresentado anteriormente na Listagem 2.3 para efetuar a criação e destruição de um enclave.

Listagem 2.7. Arquivo fonte *app.c*.

```
1 #include "stdio.h"
```

```
2  #include "string.h"
3  #include "sgx_eid.h"
4  #include "sgx_urts.h"
5  #include "Enclave_u.h" // Gerado pelo Edger8r
6
7  #define ENCLAVE_FILENAME "enclave.signed.so"
8
9  int main(int argc, char *argv[])
10 {
11     char *data = "Hello World!!!";
12     uint8_t *sealed_data, *unsealed_data;
13     uint32_t data_size = strlen(data), sealed_data_size, unsealed_data_size, i;
14     sgx_enclave_id_t eid = 0;
15     sgx_launch_token_t token = {0};
16     sgx_status_t ret = SGX_ERROR_UNEXPECTED;
17     int updated = 0;
18
19     ret = sgx_create_enclave(ENCLAVE_FILENAME, SGX_DEBUG_FLAG, &token, &updated, &eid,
20         NULL);
21
22     if (ret != SGX_SUCCESS) {
23         printf("Erro na inicializacao do enclave ...\n");
24         return -1;
25     }
26
27     ret = ecall_get_sealed_data_size(eid, data_size, &sealed_data_size);
28
29     if (ret != SGX_SUCCESS) {
30         printf("Erro ao calcular o tamanho dos dados selados.\n");
31         return -1;
32     }
33
34     sealed_data = (uint8_t*) malloc(sealed_data_size);
35
36     ret = ecall_seal_data(eid, (uint8_t*)data, data_size, sealed_data,
37         sealed_data_size);
38
39     if (ret != SGX_SUCCESS) {
40         printf("Erro ao selar dados ...\n");
41         return -1;
42     }
43
44     printf("Dados selados: \n");
45     for(i = 0; i < sealed_data_size; i++)
46         printf("%02x ", sealed_data[i]);
47     printf("\n");
48
49     ret = ecall_get_unsealed_data_size(eid, sealed_data, sealed_data_size,
50         &unsealed_data_size);
51
52     if (ret != SGX_SUCCESS) {
53         printf("Erro ao calcular o tamanho dos dados abertos.\n");
54         return -1;
55     }
56
57     unsealed_data = (uint8_t*) malloc(unsealed_data_size);
58
59     ret = ecall_unseal_data(eid, sealed_data, sealed_data_size, unsealed_data,
60         unsealed_data_size);
61
62     if (ret != SGX_SUCCESS) {
63         printf("Erro ao abrir dados ...\n");
64         return -1;
65     }
66
67     printf("Dados abertos: \n%s\n", (char*)unsealed_data);
68
69     if(sgx_destroy_enclave(eid) != SGX_SUCCESS) {
70         printf("ERRO ao destruir o enclave.\n");
71         return -1;
```

```
72      }
73
74      return 0;
75  }
```

Após a criação do enclave, o primeiro passo é verificar qual será o tamanho dos dados após selados (linha 27). Isso se faz necessário pois deve-se alocar memória para que esses dados sejam armazenados. Esta operação é efetuada através de uma chamada ECALL, que recebe como parâmetro de entrada o tamanho dos dados a serem selados. Esta função, assim como as outras funções ECALL deste projeto, serão detalhadas mais adiante.

Após verificar qual será o tamanho dos dados selados, é feita a alocação de memória para armazená-los, e então efetua-se a chamada de uma segunda função ECALL (linha 36) que efetuará a selagem dos dados. Esta função recebe como parâmetros de entrada os dados a serem selados, o tamanho destes e o tamanho dos dados após selados, e como parâmetro de saída os dados selados. Após a selagem dos dados, os mesmos são impressos em tela *byte* a *byte*.

Por fim, de forma semelhante, é efetuado o processo para abrir os dados selados. Primeiramente, na linha 49, é executada uma chamada ECALL para saber qual será o tamanho dos dados após abertos (neste caso, esta etapa poderia ser omitida, pois neste exemplo o tamanho dos dados já é previamente conhecido), em seguida a memória necessária é alocada, e então uma segunda chamada ECALL é efetuada para abrir os dados (linha 59).

As funções ECALL estão definidas no arquivo fonte `lib_ecalls.c`, que é é apresentado na Listagem 2.8. A primeira ECALL, `ecall_get_sealed_data_size`, é responsável por calcular o tamanho dos dados após serem selados; para isso, recebe como parâmetro de entrada o tamanho dos dados abertos. Tal cálculo é efetuado pela função `sgx_calc_sealed_data_size`, pertencente ao SDK do SGX, que recebe dois parâmetros:

- `add_mac_txt_size`: Tamanho dos dados adicionais. Estes dados não serão cifrados, mas irão compor o cálculo do MAC (*Message Authentication Code*). Neste exemplo, é informado o valor 0 como parâmetro, pois não há dados adicionais.
- `txt_encrypt_size`: Tamanho dos dados que serão cifrados.

Listagem 2.8. Arquivo fonte *lib_ecalls.c*.

```
1  #include "sgx_tseal.h"
2  #include "Enclave_t.h" // Gerado pelo Edger8r
3
4  void ecall_get_sealed_data_size(uint32_t data_size, uint32_t *sealed_data_size) {
5      *sealed_data_size = sgx_calc_sealed_data_size(0, data_size);
6  }
7
8  void ecall_seal_data(uint8_t *data, uint32_t data_size, uint8_t *sealed_data,
9      uint32_t sealed_data_size) {
10
11     sgx_status_t ret = SGX_ERROR_UNEXPECTED;
12
13     ret = sgx_seal_data(0, NULL, data_size, data, sealed_data_size,
14         (sgx_sealed_data_t*)sealed_data);
15 }
```

```
16
17 void ecall_get_unsealed_data_size(uint8_t *sealed_data, uint32_t sealed_data_size,
18     uint32_t *unsealed_data_size) {
19
20     *unsealed_data_size = sgx_get_encrypt_txt_len((sgx_sealed_data_t*)sealed_data);
21 }
22
23 void ecall_unseal_data(uint8_t *sealed_data, uint32_t sealed_data_size,
24     uint8_t *unsealed_data, uint32_t unsealed_data_size) {
25
26     sgx_status_t ret = SGX_ERROR_UNEXPECTED;
27
28     ret = sgx_unseal_data((sgx_sealed_data_t*)sealed_data, NULL, 0, unsealed_data,
29         &unsealed_data_size);
30 }
```

A função `ecall_seal_data` é utilizada para efetuar a selagem dos dados, recebendo quatro parâmetros: os dados a serem selados, o tamanho dos dados a serem selados, os dados após a selagem (parâmetro de saída) e o tamanho dos dados após selados. Esta função faz uma chamada à função `sgx_seal_data`, disponível no SDK, passando a ela seis parâmetros:

- `additional_MACtext_length`: Tamanho dos dados adicionais que serão utilizados no cálculo do MAC. Neste exemplo não há dados adicionais, por este motivo é passado o valor 0.
- `*p_additional_MACtext`: Dados adicionais que serão utilizados no cálculo do MAC. Neste exemplo não há dados adicionais, por este motivo é passado o valor `NULL`.
- `text2encrypt_length`: Tamanho dos dados a serem selados.
- `*p_text2encrypt`: Dados a serem selados.
- `sealed_data_size`: Tamanho dos dados após a selagem.
- `*p_sealed_data`: Dados selados. Este é um parâmetro de saída da função.

A função `sgx_seal_data` sela os dados para o autor do enclave, ou seja, qualquer enclave produzido pelo mesmo autor estará apto a abrir os dados. O SDK também disponibiliza a função `sgx_seal_data_ex`, que funciona de forma semelhante, permitindo que seja escolhida entre a selagem de dados para o autor ou para o enclave, restringindo a abertura desses dados apenas para o enclave que os selou.

A função `ecall_get_unsealed_data_size` é responsável por calcular o tamanho dos dados após abertos. Para isso, recebe como parâmetros de entrada os dados selados e o tamanho destes, tendo como parâmetro de saída o tamanho dos dados após abertos. Este cálculo é feito através da função `sgx_get_encrypt_txt_len`, disponível no SDK do SGX, que recebe como único parâmetro os dados selados na forma da estrutura `sgx_sealed_data_t`.

Por fim, a função `ecall_unseal_data` efetua a abertura dos dados selados, tendo como parâmetros os dados selados, o tamanho destes, os dados abertos (parâmetro

de saída), e o tamanho dos dados após abertos. A abertura dos dados é feita através da função `sgx_unseal_data`, que requer os seguintes parâmetros:

- `*p_sealed_data`: Dados selados, na forma da estrutura `sgx_sealed_data_t`.
- `*p_additional_MACtext`: Parâmetro de saída que retorna os dados opcionais que fazem parte do cálculo do MAC. No caso do exemplo apresentado, é passado o valor `NULL` por não haver dados adicionais. Caso existam dados adicionais, deverá ser alocada memória para retornar estes dados.
- `*p_additional_MACtext_length`: Tamanho dos dados adicionais que fazem parte do cálculo do MAC. Na existência de dados adicionais, deve-se utilizar a função `sgx_get_add_mac_txt_len` para calcular o tamanho destes. Caso não existam dados adicionais, o valor passado deve ser 0.
- `*p_decrypted_text`: Parâmetro de saída que retorna os dados decifrados.
- `*p_decrypted_text_length`: Parâmetro que recebe o tamanho do *buffer* de dados depois de decifrados. Este parâmetro também retorna o tamanho real do *buffer* após decifrados.

Por fim, a Listagem 2.9 apresenta o arquivo EDL que contém as definições para as funções ECALL do enclave. Todas ECALLs são definidas com acesso público, permitindo que estas sejam chamadas pela aplicação.

Listagem 2.9. Arquivo EDL contendo as definições das funções ECALLs.

```
enclave {
    trusted {
        public void ecall_get_sealed_data_size(uint32_t data_size,
            [out] uint32_t *sealed_data_size);

        public void ecall_seal_data([in, size=data_size] uint8_t *data,
            uint32_t data_size, [out, size=sealed_data_size] uint8_t *sealed_data,
            uint32_t sealed_data_size);

        public void ecall_get_unsealed_data_size(
            [in, size=sealed_data_size] uint8_t *sealed_data,
            uint32_t sealed_data_size, [out] uint32_t *unsealed_data_size);

        public void ecall_unseal_data([in, size=sealed_data_size] uint8_t *sealed_data,
            uint32_t sealed_data_size,
            [out, size=unsealed_data_size] uint8_t *unsealed_data,
            uint32_t unsealed_data_size);
    };
};
```

Um ponto importante a ser observado é a utilização do atributo `[size]` em conjunto com os atributos `[in]` e `[out]` quando o parâmetro é um *buffer* de dados. Isso é necessário para que, ao se utilizar o atributo `[in]`, os dados sejam copiados para a área protegida de memória, para serem utilizados pelo enclave. Na utilização do atributo `[out]` os dados serão copiados para a área não protegida de memória, para que possam ser utilizados pela aplicação. Todo esse processo de cópia dos dados é executado pelas rotinas de borda criadas pela ferramenta *Edger8r*.

2.6.7. Atestação Entre Enclaves

Esta seção apresenta um exemplo de atestação local entre dois enclaves (ilustrado na Figura 2.9), permitindo a criação de um canal seguro de comunicação para efetuar o envio de uma mensagem. Este canal seguro é criado através de uma troca de chaves *Diffie-Hellman* entre os enclaves, sendo que a chave acordada é posteriormente utilizada para cifrar a mensagem a ser enviada. O processo de troca de chaves entre os enclaves se divide em 12 etapas representadas no diagrama da Figura 2.13, numeradas na ordem em que ocorrem.

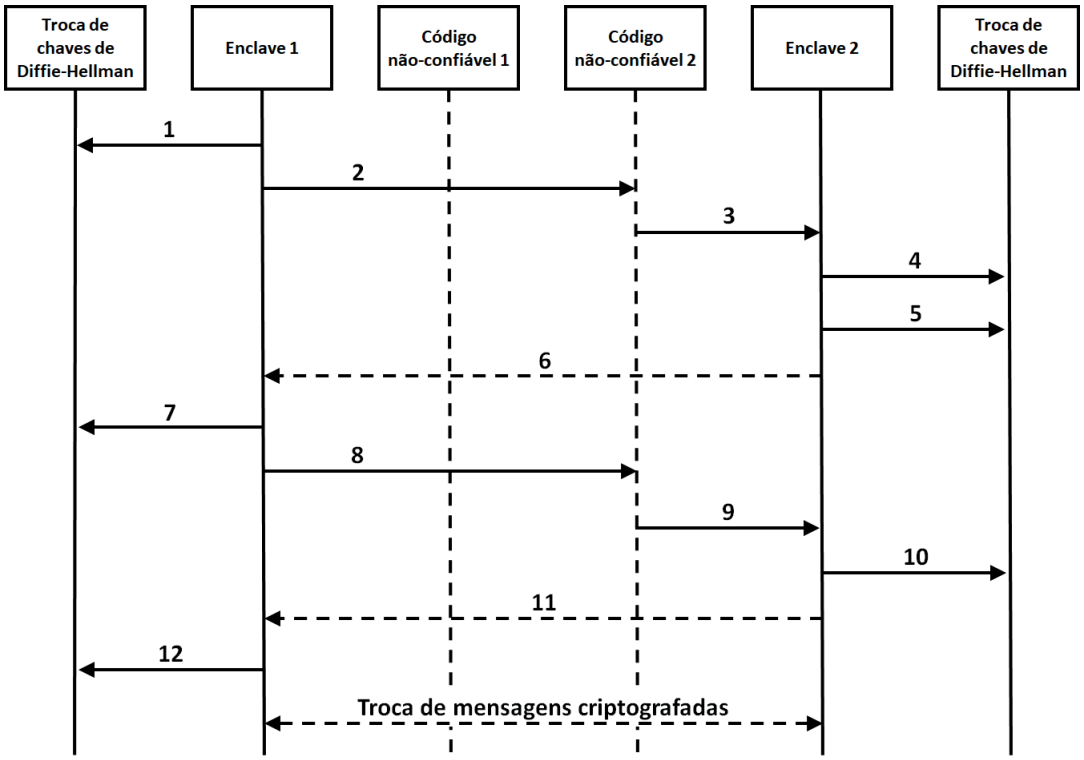

Figura 2.13. Etapas necessárias para efetuar a atestação local entre dois enclaves utilizando troca de chaves *Diffie-Hellman* (adaptado de [Intel 2016b]).

1. O enclave que deseja iniciar a atestação (identificado neste exemplo como Enclave 1) efetua uma chamada para a função `sgx_dh_init_session` de forma a iniciar a sessão para a troca de chaves, na condição de inicializador da requisição;

2. O Enclave 1 efetua uma chamada OCALL identificando a instância do enclave de destino (Enclave 2);

3. A aplicação efetua uma ECALL para o Enclave 2 para que este também inicie uma sessão para a troca de chaves;

4. O Enclave 2 inicia o processo de criar uma sessão para a troca de chaves, também utilizando a função `sgx_dh_init_session`, mas na condição de respondedor da requisição;

5. O Enclave 2 gera uma estrutura `sgx_dh_msg1_t`, através da função `sgx_dh_responder_gen_msg1`, que contém uma chave pública baseada em

uma curva elíptica NIST P-256, além da identificação do enclave destinatário da chave;

6. A estrutura `sgx_dh_msg1_t` é retornada ao Enclave 1. Nesta etapa conclui-se o passo 1 apresentado na Figura 2.9;

7. O Enclave 1 valida a estrutura `sgx_dh_msg1_t` através da função *sgx_dh_initiator_proc_msg1*, gerando também uma segunda estrutura, `sgx_dh_msg2_t`, que contém uma chave pública do Enclave 1, a sua estrutura *EREPORT* e um valor CMAC (*Cipher-based Message Authentication Code*), que será validada pelo Enclave 2 para atestar que o Enclave 1 está sendo executado na mesma plataforma;

8. O Enclave 1 executa uma chamada OCALL para enviar ao Enclave 2 a estrutura `sgx_dh_msg2_t` gerada;

9. A aplicação executa uma chamada ECALL para encaminhar as informações para o Enclave 2. Nesta etapa conclui-se o passo 2 apresentado na Figura 2.9;

10. O Enclave 2 valida a estrutura `sgx_dh_msg2_t` gerada pelo Enclave 1 através da função `sgx_dh_responder_proc_msg2` e, a partir desta, gera uma estrutura `sgx_dh_msg3_t` que irá conter a estrutura *EREPORT* do Enclave 2 e um valor CMAC calculado com base nesta estrutura. Nesta etapa também é gerada uma chave de 128 *bits*, definida como AEK, que será utilizada para cifrar as mensagens trocadas entre os enclaves;

11. A estrutura `sgx_dh_msg3_t` e a chave AEK gerada são retornadas ao Enclave 1. Nesta etapa conclui-se o passo 3 apresentado na Figura 2.9;

12. O Enclave 1 valida a estrutura `sgx_dh_msg3_t` através da função `sgx_dh_initiator_proc_msg3` para atestar que o Enclave 2 está sendo executado na mesma plataforma.

Após os enclaves confirmarem suas identidades e atestarem que ambos estão sendo executados na mesma plataforma, eles podem iniciar a troca de mensagens, cifrando-as com a chave AEK de 128 *bits* obtida no acordo de *Diffie-Hellman*. Para efetuar a cifragem das mensagens, utiliza-se a função `sgx_rijndael128GCM_encrypt`, que recebe os seguintes parâmetros:

- `*p_key`: Um ponteiro para a chave a ser utilizada no processo de criptografia. A chave deve conter 128 *bits*;
- `*p_src`: Um ponteiro para os dados que serão cifrados. Este valor pode ser nulo se houver dados adicionais (parâmetro `*p_aad`);
- `src_len`: Tamanho do *buffer* de dados que será cifrado. Este valor pode ser 0, desde que os parâmetros `*p_src` e `*p_dst` sejam nulos e o parâmetro `aad_len` maior que zero;

- `*p_dst`: Um ponteiro para um *buffer* onde os dados cifrados serão salvos. Este *buffer* deve ser previamente alocado;
- `*p_iv`: Um ponteiro para o vetor de inicialização utilizado no cálculo AES-GCM, com tamanho recomendado de 12 *bytes*;
- `iv_len`: Tamanho do ponteiro de inicialização;
- `*p_aad`: Um ponteiro para um *buffer* de dados adicional que será utilizado para efetuar o cálculo GCM MAC. Este parâmetro é opcional, podendo ser nulo, e o conteúdo deste *buffer* não será cifrado;
- `aad_len`: Tamanho do *buffer* de dados adicional. Caso o *buffer* adicional seja nulo, o valor deste parâmetro deve ser zero;
- `*p_out_mac`: Parâmetro de saída contendo o resultado do cálculo GCM MAC efetuado sobre os dados cifrados e os dados adicionais.

Para efetuar a decifragem dos dados utiliza-se a função `sgx_rijndael128GCM_decrypt`, que recebe os mesmos parâmetros que a função `sgx_rijndael128GCM_encrypt`. No processo de decifragem o parâmetro `*p_src` recebe os dados cifrados e o parâmetro `*p_dst` retorna os dados decifrados. Além disso, o parâmetro `*p_out_mac`, neste caso, recebe o valor GCM MAC calculado no processo de cifragem.

O projeto de exemplo para atestação local é divido em quatro arquivos fonte. Os arquivos `lib_ecalls1.c` e `lib_ecalls2.c` contêm as funções ECALLs dos enclaves 1 e 2, respectivamente. O arquivo fonte `lib_ocalls.c` contém as funções OCALLs de ambos os enclaves. Por fim, o arquivo fonte `app.c` contém o código da aplicação, responsável por instanciar os dois enclaves e efetuar a chamada à função ECALL do enclave 1, responsável por enviar a mensagem ao enclave 2.

A Listagem 2.10 apresenta o arquivo fonte com as funções ECALL do enclave 1. A função `Enclave1_ecall_send_message` é responsável por iniciar o processo de atestação e, após concluí-lo, enviar a mensagem cifrada ao enclave 2. Na linha 65 efetua-se a chamada à função `Enclave1_create_session`, que irá efetuar o processo de atestação. Dentro dessa função, na linha 24, é iniciado o processo de atestação, criando uma sessão para a troca de chaves (etapa 1 do diagrama da Figura 2.13). Após isso, é efetuada uma chamada OCALL (etapa 2, linha 29) que irá executar uma chamada ECALL para o enclave 2 (etapa 3), apresentada na Listagem 2.10 como `Enclave2_ecall_session_request`, que será responsável por iniciar a sessão no enclave 2 (etapa 4) e criar a estrutura `sgx_dh_msg1_t` (etapa 5) que será validada pelo Enclave 1.

Listagem 2.10. Arquivo fonte *lib_ecalls.c* do Enclave 1.

```
1 #include "sgx_eid.h"
2 #include "sgx_report.h"
3 #include "sgx_eid.h"
4 #include "sgx_ecp_types.h"
5 #include "sgx_dh.h"
6 #include "sgx_tseal.h"
```

```
7  #include "stdlib.h"
8  #include "string.h"
9  #include "Enclave1_t.h"
10
11 sgx_key_128bit_t session_dh_aek;
12 sgx_dh_session_t sgx_dh_session;
13
14 sgx_status_t Enclave1_create_session(sgx_enclave_id_t src_enclave_id,
15     sgx_enclave_id_t dest_enclave_id) {
16
17     sgx_status_t status = SGX_SUCCESS;
18     sgx_key_128bit_t dh_aek;
19     sgx_dh_msg1_t dh_msg1;  //Diffie-Hellman Message 1
20     sgx_dh_msg2_t dh_msg2;  //Diffie-Hellman Message 2
21     sgx_dh_msg3_t dh_msg3;  //Diffie-Hellman Message 3
22     sgx_dh_session_enclave_identity_t responder_identity;
23
24     status = sgx_dh_init_session(SGX_DH_SESSION_INITIATOR, &sgx_dh_session);
25     if(status != SGX_SUCCESS) {
26         return status;
27     }
28
29     status = Enclave1_ocall_session_request(dest_enclave_id, &dh_msg1);
30     if (status != SGX_SUCCESS) {
31         return status;
32     }
33
34     status = sgx_dh_initiator_proc_msg1(&dh_msg1, &dh_msg2, &sgx_dh_session);
35     if (status != SGX_SUCCESS) {
36         return status;
37     }
38
39     status = Enclave1_ocall_exchange_report(dest_enclave_id, &dh_msg2, &dh_msg3,
40         &dh_aek);
41     if (status != SGX_SUCCESS) {
42         return status;
43     }
44
45     status = sgx_dh_initiator_proc_msg3(&dh_msg3, &sgx_dh_session, &dh_aek,
46         &responder_identity);
47     if (status != SGX_SUCCESS) {
48         return status;
49     }
50
51     memcpy(&session_dh_aek, dh_aek, sizeof(sgx_key_128bit_t));
52
53     return SGX_SUCCESS;
54 }
55
56 void Enclave1_ecall_send_message(sgx_enclave_id_t src_enclave_id,
57     sgx_enclave_id_t dest_enclave_id, const char* message) {
58
59     sgx_status_t status;
60     uint32_t src_len = strlen(message);
61     uint8_t p_dest2[src_len];
62     sgx_aes_gcm_data_t* secure_message;
63     size_t message_size;
64
65     status = Enclave1_create_session(src_enclave_id, dest_enclave_id);
66     if(status != SGX_SUCCESS) {
67         ocall_print("Enclave1_create_session ERRO");
68         return;
69     }
70
71     message_size = sizeof(sgx_aes_gcm_data_t) + src_len;
72     secure_message = (sgx_aes_gcm_data_t*)malloc(message_size);
73     secure_message->payload_size = src_len;
74
75     status = sgx_rijndael128GCM_encrypt(&session_dh_aek, (uint8_t*)message, src_len,
76         secure_message->payload, secure_message->reserved,
```

```
77          sizeof(secure_message->reserved), NULL, 0, &(secure_message->payload_tag));
78      if(status != SGX_SUCCESS) {
79          ocall_print("sgx_rijndael128GCM_encrypt ERRO");
80          return;
81      }
82
83      status = Enclave1_ocall_send_request(dest_enclave_id, secure_message,
84          message_size);
85      if(status != SGX_SUCCESS) {
86          ocall_print("ERRO ao enviar mensagem");
87          return;
88      }
89  }
```

Após criada, a estrutura `sgx_dh_msg1_t` é retornada para o Enclave 1 (etapa 6) e então validada por ele (chamada na linha 34 da Listagem 2.10, correspondente à etapa 7 do diagrama da Figura 2.13). A mesma função que valida a estrutura `sgx_dh_msg1_t` gera uma segunda estrutura, `sgx_dh_msg2_t`, que contém as informações do Enclave 1, a qual é encaminhada ao Enclave 2, efetuando uma chamada OCALL (etapa 8, linha 39 da Listagem 2.10) e posteriormente uma chamada ECALL para o Enclave 2 (etapa 9). O Enclave 2 então valida esta estrutura (etapa 10, linha 43 da Listagem 2.11), gerando uma terceira estrutura, `sgx_dh_msg3_t`, e uma chave de 128 *bits* (`dh_aek`), que são retornadas para o Enclave 1 (etapa 11). Por fim, a última etapa de atestação é a validação da estrutura `sgx_dh_msg3_t` por parte do Enclave 1 (linha 45 da Listagem 2.10).

Finalizado o processo de atestação entre os enclaves, as mensagens trocadas entre eles são cifradas utilizando a chave AEK de 128 *bits* obtida durante o processo de atestação. Neste exemplo, a cifragem da mensagem é efetuada na linha 75 da Listagem 2.10, e a mensagem cifrada é encaminhada através de uma função OCALL (linha 83). Esta função OCALL é responsável por encaminhar a mensagem para o Enclave 2, efetuando uma chamada ECALL para este. O Enclave 2 recebe a mensagem e decifra ela (linha 59 da Listagem 2.11).

Listagem 2.11. Arquivo fonte *lib_ecalls.c* do Enclave 2.

```
1   #include "sgx_eid.h"
2   #include "sgx_report.h"
3   #include "sgx_eid.h"
4   #include "sgx_ecp_types.h"
5   #include "sgx_dh.h"
6   #include "sgx_tseal.h"
7   #include "string.h"
8   #include "stdlib.h"
9   #include "Enclave2_t.h"
10
11  sgx_key_128bit_t session_dh_aek;
12  sgx_dh_session_t sgx_dh_session;
13
14  void Enclave2_ecall_session_request(sgx_dh_msg1_t *dh_msg1) {
15      sgx_status_t status = SGX_SUCCESS;
16
17      if(!dh_msg1) {
18          ocall_print("dh_msg1 INVALIDA!");
19          return;
20      }
21
22      ocall_print("Enclave 2 sgx_dh_init_session");
23      status = sgx_dh_init_session(SGX_DH_SESSION_RESPONDER, &sgx_dh_session);
24      if(status != SGX_SUCCESS) {
25          ocall_print("ERRO!");
26          return;
```

```
27      }
28
29      ocall_print("Enclave 2 sgx_dh_responder_gen_msg1");
30      status = sgx_dh_responder_gen_msg1(dh_msg1, &sgx_dh_session);
31      if(status != SGX_SUCCESS) {
32          ocall_print("ERRO!");
33          return;
34      }
35  }
36
37  void Enclave2_ecall_exchange_report(sgx_dh_msg2_t *dh_msg2,
38      sgx_dh_msg3_t *dh_msg3, sgx_key_128bit_t *dh_aek) {
39
40      sgx_dh_session_enclave_identity_t initiator_identity;
41
42      ocall_print("Enclave 2 sgx_dh_responder_proc_msg2");
43      sgx_status_t status = sgx_dh_responder_proc_msg2(dh_msg2, dh_msg3, &sgx_dh_session,
44          dh_aek, &initiator_identity);
45      if(status != SGX_SUCCESS) {
46          ocall_print("ERRO!");
47          return;
48      }
49
50      memcpy(&session_dh_aek, dh_aek, sizeof(sgx_key_128bit_t));
51  }
52
53  void Enclave2_ecall_receive_message(sgx_aes_gcm_data_t* message,
54      size_t message_size) {
55
56      sgx_status_t status;
57      uint8_t p_dest[message->payload_size];
58
59      status = sgx_rijndael128GCM_decrypt(&session_dh_aek, message->payload,
60          message->payload_size, p_dest, message->reserved, sizeof(message->reserved),
61          NULL, 0, &(message->payload_tag));
62
63      if(status != SGX_SUCCESS) {
64          ocall_print("sgx_rijndael128GCM_decrypt ERRO");
65      } else {
66          ocall_print((char*)p_dest);
67      }
68  }
```

A mensagem cifrada pode ser encapsulada em uma estrutura `sgx_aes_gcm_data_t`, que irá conter o tamanho da mensagem (campo `payload_size`), um campo reservado para alinhar os dados em 16 *bytes* (campo `reserved`) , a mensagem cifrada e o conteúdo adicional autenticado, mas não cifrado (campo `payload`). Além destes, a estrutura também contém um quarto campo, `payload_tag`, que armazena o valor AES-GMAC calculado sobre os outros três campos da estrutura.

2.7. Considerações Finais

Neste trabalho foram apresentados alguns mecanismos de segurança baseados em *hardware*, que são utilizados para garantir a segurança de aplicações e a confidencialidade dos dados dos usuários, com enfoque principal na arquitetura Intel SGX – *Software Guard Extensions*, que permite a execução de aplicações em um ambiente seguro, denominado enclave, além de selagem de dados e atestação entre aplicações, de forma local ou remota.

Apesar de ser uma arquitetura relativamente nova, tendo sido disponibilizada no final do ano de 2015, já existem diversos trabalhos que fazem uso dela, em diferentes áreas

de aplicação, como descrito na Seção 2.4, além de questões ainda em aberto a respeito desta nova tecnologia, conforme abordado na Seção 2.5.

Por fim, os exemplos práticos apresentados na Seção 2.6 visam fornecer uma base de conhecimento para a utilização do SGX SDK para a construção de aplicações reais. Para priorizar a didática de apresentação, os exemplos apresentados abrem mão de algumas premissas de segurança como, por exemplo, gerar os dados a serem selados ou transferidos entre enclaves dentro do próprio enclave (visto que os dados gerados fora do enclave estão sujeitos a inspeções por outras aplicações). Para o desenvolvimento de aplicações seguras reais, essas premissas de segurança devem ser consideradas.

Referências

[Amin et al. 2008] Amin, M., Khan, S., Ali, T., and Gul, S. (2008). Trends and directions in trusted computing: Models, architectures and technologies. In *International Multiconference Of Engineers and Computer Scientist*, volume 1, pages 19–21.

[Anati et al. 2013] Anati, I., Gueron, S., Johnson, S. P., and Scarlata, V. R. (2013). Innovative technology for CPU based attestation and sealing. In *2nd Intl Workshop on Hardware and Architectural Support for Security and Privacy*, New York, NY. ACM.

[Anderson et al. 2006] Anderson, R., Bond, M., Clulow, J., and Skorobogatov, S. (2006). Cryptographic processors: A survey. *Proceedings of the IEEE*, 94(2):357–369.

[ARM 2009] ARM, A. (2009). Security technology building a secure system using TrustZone technology (white paper). *ARM Limited*.

[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 Eds.

[Baumann et al. 2015] Baumann, A., Peinado, M., and Hunt, G. (2015). Shielding applications from an untrusted cloud with Haven. *ACM Trans. Comput. Syst.*, 33(3):8:1–8:26.

[Bossuet et al. 2013] Bossuet, L., Grand, M., Gaspar, L., Fischer, V., and Gogniat, G. (2013). Architectures of flexible symmetric key crypto engines: A survey: From hardware coprocessor to multi-crypto-processor system on chip. *ACM Compututer Survey*, 45(4):41:1–41:32.

[Brasser et al. 2017] Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., and Sadeghi, A.-R. (2017). Software grand exposure: SGX cache attacks are practical. *arXiv preprint arXiv:1702.07521*.

[Brekalo et al. 2016] Brekalo, H., Strackx, R., and Piessens, F. (2016). Mitigating password database breaches with Intel SGX. In *1st Workshop on System Software for Trusted Execution*, SysTEX '16, pages 1:1–1:6, New York, NY, USA. ACM.

[Brenner et al. 2017] Brenner, S., Hundt, T., Mazzeo, G., and Kapitza, R. (2017). *Secure Cloud Micro Services Using Intel SGX*, pages 177–191. Springer International Publishing, Neuchâtel, Switzerland.

[Costan and Devadas 2016] Costan, V. and Devadas, S. (2016). Intel SGX explained. Cryptology ePrint Archive, Report 2016/086.

[Cox 2017] Cox, J. (2017). Hackers: We will remotely wipe iPhones unless Apple pays ransom. `https://motherboard.vice.com/en_us/article/hackers-we-will-remotely-wipe-iphones-unless-apple-pays-ransom`. Acessado em 01/04/2017.

[Davenport and Ford 2014] Davenport, S. and Ford, R. (2014). SGX: the good, the bad and the downright ugly. *Virus Bulletin*.

[FBI 2005] FBI (2005). Computer crime survey. `http://www.fbi.gov/publications/ccs2005.pdf`. Acessado em 01/02/2017.

[Greene 2012] Greene, J. (2012). Intel trusted execution technology, white paper. *Online: http://www. intel. com/txt*.

[Hoekstra et al. 2013] Hoekstra, M., Lal, R., Pappachan, P., Phegade, V., and Del Cuvillo, J. (2013). Using innovative instructions to create trustworthy software solutions. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP 2013, pages 11:1–11:1, New York, NY, USA. ACM.

[Intel 2014] Intel (2014). *Intel Software Guard Extensions Programming Reference*.

[Intel 2016a] Intel (2016a). *Intel Software Guard Extensions Developer Guide*. Intel Corporation.

[Intel 2016b] Intel (2016b). *Intel Software Guard Extensions SDK for Linux OS Developer Reference*. Intel Corporation.

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

[Karande et al. 2017] Karande, V., Bauman, E., Lin, Z., and Khan, L. (2017). SGX-Log: Securing system logs with SGX. In *2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 19–30, New York, NY, USA. ACM.

[Lesjak et al. 2015] Lesjak, C., Hein, D., and Winter, J. (2015). Hardware-security technologies for industrial IoT: TrustZone and security controller. In *41st Annual Conference of the IEEE Industrial Electronics Society*, IECON 2015, pages 002589–002595.

[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., Fetzer, C., and Pietzuch, P. (2017). Glamdring: Automatic application partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference*, pages 285–298, Santa Clara, CA. USENIX Association.

[McKeen et al. 2013] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. (2013). Innovative instructions and software model for isolated execution. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, New York, NY, USA. ACM.

[Mofrad et al. 2017] Mofrad, M. H., Lee, A., and Gray, S. L. (2017). Leveraging Intel SGX to create a nondisclosure cryptographic library. *arXiv preprint arXiv:1705.04706*.

[Moghimi et al. 2017] Moghimi, A., Irazoqui, G., and Eisenbarth, T. (2017). Cachezoom: How SGX amplifies the power of cache attacks. *arXiv preprint arXiv:1703.06986*.

[Richter et al. 2016] Richter, L., Götzfried, J., and Müller, T. (2016). Isolating operating system components with Intel SGX. In *1st Workshop on System Software for Trusted Execution*, SysTEX '16, pages 8:1–8:6, New York, NY, USA. ACM.

[Schuster et al. 2015] Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., and Russinovich, M. (2015). VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE Symposium on Security and Privacy*, pages 38–54.

[Schwarz et al. 2017] Schwarz, M., Weiser, S., Gruss, D., Maurice, C., and Mangard, S. (2017). Malware guard extension: Using SGX to conceal cache attacks. *arXiv preprint arXiv:1702.08719*.

[Shih et al. 2016] Shih, M.-W., Kumar, M., Kim, T., and Gavrilovska, A. (2016). S-NFV: Securing NFV states by using SGX. In *1st ACM International Workshop on Security in SDN and NFV*, New Orleans, LA.

[TCG 2008] TCG (2008). Trusted Platform Module (TPM) summary. `https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-Summary_04292008.pdf`. Acessado em 02/04/2017.

[TCG 2016] TCG (2016). Trusted Platform Module library - part 1: Architecture. `https://trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-1-Architecture-01.38.pdf`. Acessado em 02/04/2017.

[Tian et al. 2017] Tian, H., Zhang, Y., Xing, C., and Yan, S. (2017). SGXKernel: A library operating system optimized for Intel SGX. In *Computing Frontiers Conference*, CF'17, pages 35–44, New York, NY, USA. ACM.

[Tsai et al. 2017] Tsai, C.-C., Porter, D. E., and Vij, M. (2017). Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX Annual Technical Conference*, pages 645–658, Santa Clara, CA. USENIX Association.

[Vacca 2016] Vacca, J. R. (2016). *Cloud Computing Security: Foundations and Challenges*. CRC Press.

[Weafer 2016] Weafer, V. (2016). Report: 2017 threats prediction. Technical report, McAfee Labs. Acessado em 29/03/2017.