Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Desafios no Desenvolvimento de Aplicações Seguras Usando Intel SGX

Rodolfo de Andrade Marinho Silva

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Segurança da Informação

Andrey Elísio Monteiro Brito
(Orientador)

Campina Grande, Paraíba, Brasil
©Rodolfo de Andrade Marinho Silva, 01 de março de 2018

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

S586d Silva, Rodolfo de Andrade Marinho.
Desafios no desenvolvimento de aplicações seguras usando Intel SGX / Rodolfo de Andrade Marinho Silva. ¯ Campina Grande, 2018.
72 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) ¯ Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2018.
"Orientação: Prof. Dr. Andrey Elísio Monteiro Brito".
Referências.

1. Segurança da Informação. 2. Intel SGX. 3. Privacidade de Dados. I. Brito, Andrey Elísio Monteiro. II. Título.

CDU 004.056.53(043)

"DESAFIOS NO DESENVOLVIMENTO DE APLICAÇÕES SEGURAS USANDO INTEL SGX"

RODOLFO DE ANDRADE MARINHO SILVA

DISSERTAÇÃO APROVADA EM 01/03/2018

ANDREY ELÍSIO MONTEIRO BRITO, Dr., UFCG
Orientador(a)

FRANCISCO VILAR BRASILEIRO, Ph.D, UFCG
Examinador(a)

CHARLES BEZERRA DO PRADO, Dr., INMETRO
Examinador(a)

CAMPINA GRANDE - PB

Resumo

No decorrer das últimas décadas, uma quantidade de dados de usuários cada vez maior vem sendo enviada para ambientes não controlados pelos mesmos. Em alguns casos esses dados são enviados com o objetivo de tornar esses dados públicos, mas na grande maioria das vezes há a necessidade de manter esses dados seguros e privados, ou autorizar o seu acesso apenas em usos bem específicos. Considerando o caso onde os dados devem ser mantidos privados, entidades devem tomar cuidados especiais para manter a segurança e privacidade de tais dados tanto durante a transmissão quanto durante o armazenamento e processamento dos mesmos. Com esse objetivo, vários esforços vêm sendo feitos, inclusive o desenvolvimento de componentes de *hardware* que provêem ambientes de execução confiável, *TEEs*, como o *Intel Software Guard Extensions* (*SGX*). O uso dessa tecnologia, porém, pode ser feito de forma incorreta ou ineficiente, devido a cuidados não observados durante o desenvolvimento de aplicações.

O trabalho apresentado nessa dissertação aborda os principais desafios enfrentados no desenvolvimento de aplicações que façam uso de *SGX*, e propõe boas práticas e um conjunto de ferramentas (*DynSGX*) que ajudam a fazer melhor uso das capacidades da tecnologia. Tais desafios incluem, mas não são limitados a, particionamento de aplicações de acordo com o modelo de programação do SGX, colocação de aplicações em ambientes de computação na nuvem, e, sobretudo, gerência de memória.

Os estudos apresentados neste trabalho apontam que o mal uso da tecnologia pode acarretar em uma perda de performance considerável se comparado com implementações que levam em conta as boas práticas propostas. O conjunto de ferramentas proposto neste trabalho também mostrou possibilitar a proteção de código de aplicações em ambientes de computação na nuvem, com uma sobrecarga desprezível em comparação com o modelo de programação padrão de SGX.

// Abstract

During the last few decades, an increasing amount of user data have been sent to environments not controlled by data owners. In some cases these data are sent with the objective to turn them public, but in the vast majority of times, these data need to be kept safe and private, or to be allowed access only in very specific use cases. Considering the case where data need to be kept private, entities must take specific measures to maintain the data security and privacy while transmitting, storing and processing them. With this objective many efforts have been made, including the specification of hardware components that provide a trusted execution environment (TEEs), like the Intel Software Guard Extensions (SGX). The use of this technology , though, can be made in incorrect or ineffective ways, due to not taking some considerations into account during the development of applications.

In this work, we approach the main challenges faced in the development of applications that use SGX, and propose good practices and a toolset (DynSGX) that help making better use of the capabilities of this technology. Such challenges include, but are not limited to, application partitioning, application colocation in cloud computing environments, and memory management.

The studies presented in this work show that the bad use of this technology can result in a considerable performance loss when compared to implementations that take into account the good practices proposed. The toolset proposed in this work also showed to enable protecting application code in cloud computing environments, having a negligible performance overhead when compared to the regular SGX programming model.

Agradecimentos

Primeiramente, a Deus, agradeço pelos dons recebidos, que me possibilitaram pensar e desenvolver este trabalho de pesquisa.

Agradeço à minha família pelo apoio e incentivo à minha formação, dados durante toda a minha vida.

A Elaine, pelo companheirismo e paciência em momentos cruciais no decorrer do mestrado, especialmente durante a reta final.

Agradeço também ao LSD, pelos agradáveis ambiente e infraestrutura providos, e pela grande interdisciplinaridade de projetos existentes, que contribuíram para enxergar como meu trabalho se insere no mundo real.

Aos amigos do projeto SecureCloud (Marcus, Lília, Gabriel, Amanda, Fábio, Leandro, Pedro, Dalton, Matteus, Jefferson, Ionésio e Léo), agradeço por colaborarem diretamente com o desenvolvimento deste mestrado, seja provendo algumas dúvidas, seja provendo soluções para outras dúvidas, e aos demais companheiros de sala (Igor, Iury, Roberto, e Armstrong) pelas discussões sobre os mais variados temas, necessárias para refrescar um pouco a mente após longos períodos de estudo.

Por fim, mas não menos importante, agradeço a meu orientador, Andrey, pelos conselhos dados, pela confiança em meu trabalho, e, sobretudo, pela constante empolgação com o tema sendo pesquisado.

A todos vocês, o meu muito obrigado!

Conteúdo

1 Introdução **1**

1.1 Contextualização 1

1.1.1 Segurança e privacidade 1

1.1.2 Ambiente de execução confiável 2

1.1.3 Intel SGX 3

1.2 Problema investigado 4

1.3 Contribuições e relevância 4

1.4 Metodologia 5

1.5 Organização da dissertação 6

2 Estado da arte e trabalhos relacionados **7**

2.1 Base de computação confiável 7

2.2 Primitivas criptográficas 8

2.3 Segurança de dados baseada em *software* 9

2.3.1 Sistemas criptográficos 10

2.3.2 Isolamento de recursos 11

2.3.3 Considerações 12

2.4 Segurança de dados baseada em *hardware* 12

2.5 Trabalhos relacionados 13

2.6 Considerações 14

3 Intel SGX **16**

3.1 Introdução ao SGX 16

3.2 Modelo de ameaça 17

3.3 Modelo de memória 17
3.3.1 Paginação da EPC 18
3.4 Modelo de programação 19
3.5 Funcionalidades importantes 20
3.5.1 Medição de enclaves 20
3.5.2 Atestação de enclaves 20
3.5.3 Selagem de dados 24
3.6 Ciclo de vida de aplicações SGX 24
3.7 Intel SGX SDK e PSW 26
3.8 Casos de uso de SGX 26
3.9 Limitações do Intel SGX 28
3.9.1 Privacidade de código 28
3.9.2 Ligação estática de bibliotecas 28
3.9.3 Memória disponível 28
3.9.4 Vulnerabilidades 29
3.10 Considerações 30

4 Características-chaves em aplicações SGX 31
4.1 Gerência de memória 31
4.2 Particionamento de dados 35
4.3 Particionamento de aplicações 39

5 DynSGX 44
5.1 Introdução 44
5.2 Componentes 44
5.3 TCB do DynSGX 45
5.4 Modelo de programação 45
5.5 Ciclo de vida de aplicações 47
5.6 Ligação distribuída de código 48
5.7 Requisitos 49
5.8 Vulnerabilidades 50
5.9 Avaliação 52

5.9.1 Preparação dos experimentos 52
5.9.2 Experimentos 53
5.9.3 Discussão 55
5.10 Considerações 56

6 Conclusão **58**
6.1 Sumário 58
6.2 Limitações e trabalhos futuros 59

A Exemplo de aplicação SGX **67**
A.1 Desenvolvimento do arquivo EDL 67
A.2 Desenvolvimento do enclave 68
A.3 Desenvolvimento da parte insegura da aplicação 70

Lista de Símbolos

API - *Application Programming Interface*

ASLR - *Address Space Layout Randomization*

BIOS - *Basic Input/Output System*

CH - Criptografia Homomórfica

CPU - *Central Processing Unit*

DRAM - *Dynamic Random Access Memory*

DRM - *Digital Rights Management*

EDL - *Enclave Description Language*

EPC - *Enclave Page Cache*

EPCM - *Enclave Page Cache Map*

IAS - *Intel Attestation Service*

JSON - *JavaScript Object Notation*

MAC - *Message Authentication Code*

PRM - *Processor Reserved Memory*

PSW - *Platform Software*

RA - *Remote Attestation*

SDK - *Software Development Kit*

SEV - *Secure Encrypted Virtualization*

SGX - *Software Guard eXtensions*

SME - *Secure Memory Encryption*

SO - Sistema Operacional

SPID - *Service Provider Identification*

TCB - *Trusted Computing Base*

TEE - *Trusted Execution Environment*

VM - *Virtual Machine*

Lista de Figuras

3.1 Hierarquia de memória do Intel SGX 18

3.2 Atestação local 21

3.3 Atestação remota 22

3.4 Ciclo de vida de aplicações SGX 25

4.1 Sobrecarga média de acesso à memória SGX comparado com acesso à memória em C puro, com um intervalo de 95% de confiança. 34

4.2 Tempos médios para calcular a soma dos elementos de um *array* com tamanho de 1GB, variando o tamanho das partições utilizadas no cálculo das somas parciais, com um intervalo de 95% de confiança. 38

4.3 Tempos médios para gerar e calcular a soma de N elementos, variando N entre, 100 e 100000000, com um intervalo de 95% de confiança. 42

5.1 Ciclo de Vida de aplicações DynSGX. 48

5.2 Comparação de latência entre as implementações em C puro, Intel SGX, e DynSGX 54

Lista de Tabelas

4.1 Número de chamadas à função *enclave_sum* necessárias para computar a soma dos elementos de um *array* com 1GB de tamanho, de acordo com o tamanho, em MB, das partições. 37

5.1 Comparação de performance entre as implementações em C puro, SGX e DynSGX 56

Lista de Códigos Fonte

4.1 Pseudocódigo do experimento de acesso aleatório à memória. 32

4.2 Pseudocódigo do experimento de particionamento de dados. 36

4.3 Pseudocódigo do experimento de particionamento de aplicações, com todas as tarefas executadas no enclave. 41

4.4 Pseudocódigo do experimento de particionamento de aplicações, com as tarefas separadas entre o enclave e a parte insegura. 41

5.1 Exemplo de função C para somar dois números inteiros. 46

5.2 Código *Assembler* correspondente à função *sum_function*. 46

5.3 Exemplo de *JSON* contendo o mapeamento de elementos externos para seus respectivos endereços, que podem ser usados por outras funções. 48

5.4 Exemplo de função que usa um elemento externo (a função *strcmp*). 49

5.5 Exemplo de função vulnerável a *buffer overflow*. 51

5.6 Exemplo de função vulnerável a formatação não verificada de *strings*. . . . 51

A.1 Arquivo EDL contendo a declaração das funções que cruzam a fronteira entre as partes segura e insegura da aplicação SGX. 68

A.2 Arquivo contendo a implementação das funções a serem executadas totalmente dentro do enclave da aplicação SGX. 69

A.3 Arquivo de configuração do enclave SGX. 70

A.4 Arquivo contendo a implementação da parte insegura da aplicação SGX. . . 71

Capítulo 1

Introdução

1.1 Contextualização

1.1.1 Segurança e privacidade

Vivemos em um mundo que está a cada dia mais conectado, onde pessoas estão constantemente enviando dados pessoais para ambientes que não podem ser controlados por elas. Ocasionalmente, esses dados são destinados a tornarem-se públicos. Na maioria dos casos, porém, eles devem ser mantidos privados e/ou seguros – tais dados serão doravante denominados **dados sensíveis**. Nesse sentido, desenvolvedores de aplicações devem encontrar formas de proteger os dados de seus usuários contra roubo ou modificações indevidas a todo custo.

Além disso, desenvolvedores e empresas frequentemente hospedam suas aplicações em ambientes de nuvem pública, com o objetivo de aumentar sua disponibilidade e escalabilidade, ou até mesmo para diminuir os custos com infraestrutura física [AFG$^+$10]. Em tais ambientes, múltiplas aplicações de diferentes donos podem residir no mesmo servidor físico, possibilitando que usuários maliciosos explorem vulnerabilidades de segurança existentes para obter dados sensíveis de outros usuários.

Para entender os requisitos de aplicações que lidam com dados sensíveis em cenários como os supracitados, precisamos primeiramente definir os conceitos de segurança e privacidade. Segurança diz respeito a proteções contra acesso, modificação e remoção de dados sensíveis por pessoas não autorizadas. Privacidade, por sua vez, diz repeito à proteção contra

o uso de dados sensíveis para propósitos não autorizados.

Considerando esses conceitos, aplicações que lidam com dados sensíveis devem considerar os seguintes requisitos:

- Confidencialidade – impedir o acesso a dados sensíveis por pessoas não autorizadas;
- Disponibilidade – garantir que dados sensíveis possam ser recuperados por pessoas autorizadas sempre que requisitados;
- Integridade – impedir a modificação de dados sensíveis por pessoas não autorizadas;
- Privacidade – impedir o uso de dados sensíveis para propósitos não autorizados.

Há situações em que desenvolvedores também desejam manter o código de suas aplicações privado e seguro, seja pelo uso de código proprietário ou por algum outro motivo qualquer. Nessas situações, o código de suas aplicações devem ser tratados como dados sensíveis.

1.1.2 Ambiente de execução confiável

O crescente volume de dados sensíveis sendo enviados, armazenados e processados em ambientes de computação na nuvem, por um número cada vez maior de dispositivos móveis, e as demandas de segurança e privacidade de tais dados levaram um fórum de operadores de redes móveis (OMTP, do inglês *Open Mobile Terminal Platform*) a definir um padrão para um ambiente de execução confiável (TEE, do inglês *Trusted Execution Environment*).

Inicialmente, um TEE foi definido pela OMTP como um conjunto de componentes de *hardware* e *software* que facilitam o suporte de aplicações que satisfaçam os requisitos de um dos dois níveis de segurança definidos pela mesma. O primeiro nível de segurança prevê proteção apenas contra ataques de *software*, enquanto o segundo nível de segurança prevê proteção contra ataques de *hardware* e de *software* [OMT09].

Atualmente, o padrão TEE é definido pela *GlobalPlatform*, uma associação industrial que visa desenvolver padrões para processamento seguro e confiável em serviços digitais e outros dispositivos. A *GlobalPlatform* define TEE como uma área segura do processador principal em um *smart phone* ou qualquer dispositivo conectado a ele. O TEE garante

que dados sensíveis são armazenados, processados e protegidos em um ambiente isolado e confiável [Glo].

De forma a facilitar o desenvolvimento de serviços que garantam segurança e privacidade de dados sensíveis, várias implementações de TEE foram feitas [OMT09; Log; tru; MPP+08; AMD], entre as quais destacamos o Intel SGX, do inglês *Software Guard eXtensions*, objeto de estudo deste trabalho de mestrado.

1.1.3 Intel SGX

Com o passar do tempo, não só dispositivos móveis necessitavam de um TEE, mas sim todo um nicho de dispositivos que lidam com dados sensíveis. Um exemplo desses dispositivos são os computadores, sejam eles servidores usados em ambientes de computação na nuvem, sejam eles máquinas usadas por clientes. Visando este tipo de dispositivos, em 2013 a Intel definiu uma implementação própria de TEE conhecida como SGX.

Intel SGX é um conjunto de instruções incorporado às arquiteturas Intel 64 e IA-32 que permite a criação de contêineres baseados em *hardware*, chamados **enclaves**. Dados sensíveis que sejam carregados em um enclave são protegidos contra modificação ou acesso não autorizado; Código de aplicações que executem dentro de enclaves SGX são protegidos contra modificação [MAB+13].

Intel SGX está disponível em processadores da 6ª geração da família Intel Core i, com arquitetura *Skylake*, ou processadores mais novos da mesma família, bem como em processadores da família Xeon, a partir da 5ª versão. Tais processadores estão disponíveis no mercado desde o segundo semestre do ano de 2015.

Atualmente muitas pesquisas vêm sendo desenvolvidas e publicadas[1] sobre a aplicação de Intel SGX no mundo real [HLP+13], bem como vulnerabilidades encontradas na tecnologia, e ações necessárias para mitigar estas vulnerabilidades [Swa17].

Por ser uma tecnologia bastante promissora e já estar presente em um grande número de processadores atualmente em uso, iremos utilizá-la como objeto de estudo neste trabalho de dissertação.

[1] `https://software.intel.com/en-us/sgx/academic-research`

1.2 Problema investigado

No ano de 2016, estimou-se que o custo anual com crimes cibernéticos em alguns países chega a um total de USD 74 milhões (setenta e quatro milhões de dólares americanos) [HPE]. Além disso, com a crescente produção de dados sensíveis enviados para ambientes de computação na nuvem, cresce também a quantidade de ataques cibernéticos com o intuito de obter e usar tais dados de forma não autorizada. Levando em consideração esses fatos, desenvolvedores de aplicações que lidam com dados sensíveis devem cada vez mais procurar formas de garantir a segurança e a privacidade dos dados de seus clientes.

Uma forma de proteger os dados de usuários é através do uso de um TEE. Uma das implementações de um TEE, que está cada vez mais difundida em máquinas de usuários e em máquinas servidoras de computação na nuvem, é o Intel SGX. Assim como qualquer outra tecnologia, Intel SGX também possui vulnerabilidades, como falta de proteção contra ataques de canal lateral, que podem impactar as garantias de segurança providas por aplicações que façam uso da tecnologia, bem como limitações, como um pequeno espaço de memória disponível, que podem impactar o desempenho dessas aplicações devido ao seu mau uso.

Para tirar o melhor proveito de Intel SGX, desenvolvedores devem tomar várias decisões na hora de desenvolver suas aplicações, como: (i) qual porção da aplicação realmente precisa ser protegido pelas instruções do SGX, (ii) como gerenciar a memória consumida por suas aplicações, e (iii) como evitar ataques de canal lateral.

1.3 Contribuições e relevância

Este trabalho tem como objetivo abordar os problemas de vulnerabilidades e baixa eficiência de aplicações, oriundos de más práticas no uso de Intel SGX, investigando arquitetura, implementação e implantação de aplicações que façam uso dessa tecnologia. Com esse objetivo, são propostas várias práticas que devem ser consideradas durante o processo de desenvolvimento de aplicações que lidam com dados sensíveis. Tais práticas levam em consideração as vulnerabilidades e limitações do Intel SGX, bem como o ambiente onde essas aplicações serão executadas. Considerando essas práticas, desenvolvedores podem tomar decisões como: (i) como dividir suas aplicações no modelo de programação do Intel SGX; (ii) como par-

ticionar os dados a serem processados por suas aplicações; (iii) como gerenciar a memória consumida por suas aplicações; (iv) como garantir privacidade do código de suas aplicações.

Estas decisões tomadas pelo desenvolvedor trazem benefícios como: (i) maior proteção de dados sensíveis de clientes; (ii) diminuição da sobrecarga gerada pelo uso de Intel SGX; (iii) diminuição na sobrecarga gerada pelo consumo de memória excedente à disponível; (iv) proteção de código proprietário.

Este trabalho também introduz a ferramenta DynSGX [SBB17], desenvolvida ao longo da pesquisa do mestrado, que ajuda desenvolvedores a proteger o código de suas aplicações usando enclaves SGX, bem como possibilita uma melhor gerência de memória de enclaves ocupada por código das aplicações.

O trabalho é relevante por tratar de uma tecnologia que está se tornando padrão em processadores tanto de uso pessoal, quanto de servidores, e pela ausência de guias de desenvolvimento de aplicações SGX que indiquem as implicações de decisões tomadas neste processo.

1.4 Metodologia

A partir de levantamento bibliográfico do estado da arte, que serviu de base para investigações sobre limitações e vulnerabilidades do uso de Intel SGX para proteção de dados sigilosos, foi produzida uma lista de aspectos que devem ser considerados por programadores ao desenvolver uma aplicação SGX. Baseado nessa lista, foram desenvolvidos experimentos para avaliação do impacto em segurança e performance causado por cada um desses aspectos. Em seguida, os experimentos foram implementados para avaliação, considerando diferentes abordagens para cada um dos aspectos listados, que serão detalhadas no Capítulo 4. A implementação de todos os componentes, bem como da comunicação entre diferentes componentes, foi feita usando as linguagens C e C++.

Após a implementação, os experimentos foram executados em uma máquina que possui SGX habilitado, utilizando configurações comuns em soluções existentes no mercado e na literatura. Nesta máquina, foram comparadas diferentes abordagens usadas para os aspectos estudados. Essa comparação foi feita coletando métricas como: (i) consumo de memória, (ii) tempo de execução de tarefas, (iii) tamanho do *Trusted Computing Base* de uma apli-

cação, com implicações em segurança de dados, e (iv) garantias de segurança de código de aplicações.

Por fim, uma nova ferramenta foi proposta, com o objetivo de facilitar o desenvolvimento de aplicações que façam uso do SGX, levando em consideração as dificuldades e boas práticas discutidas neste trabalho.

1.5 Organização da dissertação

Este trabalho de dissertação é organizado da seguinte forma:

- O Capítulo 2 mostra as principais soluções para segurança de dados, sejam elas baseadas em *software* ou em *hardware*, bem como trabalhos relacionados à pesquisa apresentada nesta dissertação;
- No Capítulo 3 é detalhada a tecnologia Intel SGX, incluindo uma descrição geral da tecnologia, suas vantagens, modelo de programação, modelo de memória, limitações e vulnerabilidades no seu uso;
- Em seguida, no Capítulo 4 é detalhada a abordagem aqui proposta, onde descrevemos cada um dos aspectos analisados no desenvolvimento de aplicações seguras, bem como os experimentos executados para avaliação de boas práticas considerando cada um desses aspectos, e os resultados obtidos nesta avaliação;
- No Capítulo 5 introduzimos a ferramenta DynSGX, criada com o objetivo de facilitar o desenvolvimento de aplicações SGX, e adicionar garantias e funcionalidades não providas pela tecnologia Intel SGX isoladamente.
- Por fim, no Capítulo 6 é apresentada a conclusão do trabalho, sumariando os resultados e listando limitações e trabalhos futuros.

Capítulo 2

Estado da arte e trabalhos relacionados

Neste capítulo, mostramos e discutimos as diferentes soluções existentes para garantia de segurança e privacidade de dados. Primeiramente exploramos os conceitos de **TCB** e **primitivas criptográficas** e a importância de ambos no desenvolvimento de soluções de segurança e privacidade em sistemas computacionais. Após, são discutidas as soluções de **segurança baseadas em *software***, e em seguida, as soluções de **segurança baseadas em *hardware***. Mais à frente, são apresentados os trabalhos relacionados mais relevantes à esta pesquisa. Por fim, é apresentado um breve sumário de como este trabalho aborda o uso de uma das tecnologias atuais, com o objetivo de tirar melhor proveito da mesma.

2.1 Base de computação confiável

A base de computação confiável (TCB, do inglês *Trusted Computing Base*) é um dos principais conceitos utilizados no desenvolvimento de aplicações seguras. Ela foi primeiramente definida como sendo a combinação de *kernel* e processos confiáveis, *i.e.*, que podem violar as regras de controle de acesso de um sistema [Rus81]. Mais à frente, TCB foi descrito como sendo uma pequena porção de *software* e *hardware* na qual a segurança de um sistema computacional depende, e que se diferencia de uma porção muito maior, que pode funcionar de forma incorreta, sem afetar a segurança do sistema [LABW92].

A definição mais adotada atualmente, entretanto, por ser mais formal, é a de que o TCB de um sistema computacional é o conjunto de mecanismos de proteção contidos no mesmo, incluindo *software*, *firmware* e *hardware*, que combinados são responsáveis pela aplicação

de políticas de segurança computacionais [QZW+85].

Toda aplicação segura precisa definir um TCB, e confiar nele como bloco central para as garantias de segurança da aplicação. Em outras palavras, o TCB é a única porção de uma aplicação que, possuindo uma vulnerabilidade, poderia ser atacada para tentar obter controle indevido sobre a mesma.

Considerando que quanto maior uma aplicação, maior o número de *bugs* e vulnerabilidades esperado de se encontrar nela [McC04], uma boa prática na arquitetura e desenvolvimento de um TCB é mantê-lo o menor possível.

2.2 Primitivas criptográficas

No centro de quaisquer sistemas computacionais seguros estão as primitivas criptográficas. Elas são os blocos mais básicos de tais sistemas, e geralmente são criados para realizar uma tarefa bem específica, e de forma bastante confiável.

As principais tarefas que podem ser realizadas por primitivas criptográficas – lembrando que cada primitiva deve realizar apenas uma função – são:

Função *Hash* – Produz um valor reduzido da mensagem original. Geralmente é utilizado para verificar a integridade da mensagem.

Autenticação – Ato de verificar a identidade de alguma entidade.

Cifragem simétrica – Permite cifrar e decifrar dados usando uma mesma chave.

Cifragem assimétrica – Permite cifrar e decifrar dados com um par de chaves distintas (uma pública e uma privada).

Assinatura digital – Permite verificar a autoria de uma mensagem.

Geração de números (pseudo) aleatórios – Utilizado para prover aleatoriedade necessária para a segurança de várias operações.

Diferentes implementações de primitivas criptográficas podem prover diferentes níveis de segurança. Podemos, dessa forma, dizer que uma primitiva provê N *bits* de segurança, significando que são necessárias 2^N operações computacionais para quebrar a sua segurança.

Por constituir a base da construção de sistemas seguros, uma primitiva criptográfica precisa ser bastante confiável na realização de sua funçao. Isto é, ela precisa prover exatamente o nível de segurança prometido. Exemplificando, se uma primitiva diz prover um nível de segurança de X bits, mas pode ter sua segurança quebrada com $X - 1$ *bits* (necessita apenas da metade de operações computacionais que deveria necessitar para ser quebrada), todos os protocolos e sistemas contruídos baseados nela passam a ser considerados vulneráveis.

Antes de serem utilizadas, primitivas criptográficas devem ser amplamente testadas e ter seu nível de segurança formalmente provado. Este é um processo bastante lento. Por todos os motivos apontados, e também pela possibilidade da inserção de erros durante o processo de desenvolvimento, recomenda-se que desenvolvedores não tentem criar as suas próprias primitivas criptográficas, mas utilizem as que já estão disponíveis e acolhidas pela comunidade de criptólogos e de segurança e privacidade de dados [MVOV96].

Por fim, como dito anteriormente, primitivas criptográficas realizam uma única função, o que, geralmente, não é suficiente para suprir todas as necessidades de um sistema seguro. Dessa forma, é necessário combinar o uso de várias delas para a construção de protocolos e sistemas seguros. Por exemplo, para que duas partes se comuniquem de forma segura, somente cifrar o dado do lado do remetente e decifrá-lo no lado do destinatário não é o suficiente. É necessário, também, usar alguma estratégia para verificar a integridade do dado transmitido, uma vez que um atacante no meio do caminho poderia modificar a mensagem enviada.

2.3 Segurança de dados baseada em *software*

Há várias estratégias para prover segurança e pravacidade através do uso de *software*, sendo a maioria delas baseada no uso de criptografia, que foram desenvolvidas com o objetivo de prover segurança e privacidade de dados. É importante citar que o uso de técnicas de segurança de dados baseadas em *software* implica na adição de bibliotecas e APIs ao TCB de uma aplicação, tornando-a mais suscetível a vulnerabilidades inseridas durante o processo de desenvolvimento das mesmas. Além das estratégias baseadas no uso de criptografia, podemos destacar também a criação de ambientes isolados, como máquinas virtuais, e cônteineres. Detalhamos cada uma destas estratégias a seguir.

2.3.1 Sistemas criptográficos

Sistemas criptográficos são constituídos por um conjunto de primitivas criptográficas, geralmente sendo compostos de uma primitiva para geração de chaves, uma primitiva para cifrar dados, e uma primitiva para decifrar dados.

Um sistema criptográfico pode ser definido formalmente como a tupla ($\mathtt{P}, \mathtt{C}, \mathtt{K}, \varepsilon, \mathtt{D}$), onde:

- $\mathtt{P}$ é o conjunto de **textos puros** existentes;
- $\mathtt{C}$ é o conjunto de **cifrotextos** existentes;
- $\mathtt{K}$ é o conjunto de **chaves** existentes;
- ε é o conjunto de funções $E_k : P \to C$, com $k \in K$, que cifra um texto puro em um cifrotexto;
- $\mathtt{D}$ é o conjunto de funções $D_k : C \to P$, com $k \in K$, que decifra um cifrotexto em um texto puro;
- $\forall e \in K, \exists d \in K | D_d(E_e(p)) = p, \forall p \in P$.

Em relação à última propriedade citada, é importante apontar que existem sistemas criptográficos simétricos e assimétricos. Em um sistema criptográfico simétrico, temos que $e = d$, *i.e.*, a chave utilizada para cifrar um texto puro é a mesma utilizada para decifrar o cifrotexto gerado. Como exemplo de algoritmo de criptografia simétrica atualmente em uso, podemos citar o AES, do inglês *Advanced Encryption Standard* [Pub01].

Já em um sistema criptográfico assimétrico, também conhecido como criptografia de chave pública, temos que $e \neq d$, *i.e.*, há a necessidade da criação de duas chaves distintas, sendo uma mantida em segredo (**chave privada**), e uma tornada pública (**chave pública**) para a utilização por outras partes. Se uma das chaves é utilizada para cifrar um texto plano, o seu par deve ser utilizado para decifrar o cifrotexto. Além das funcionalidades de cifrar e decifrar textos, sistemas criptográficos assimétricos também podem ser utilizados para provar a origem de uma mensagem (**assinatura digital**), negociar uma chave simétrica de forma segura (*e.g.*, algoritmo Diffie-Hellman [Mer78]), e até mesmo para realizar processamento de dados utilizando apenas seu cifrotexto (*e.g.*, criptografia homomórfica). Algoritmos de

criptografia assimétrica, entretanto, demandam um processamento geralmente muito mais complexo do que algoritmos de criptografia simétricos. É interessante, portanto, combinar ambas as técnicas para alcançar os objetivos de segurança sem comprometer a performance do sistema.

A técnica de criptografia homomórfica, citada anteriormente, permite o processamento de dados na forma de cifrotexto. Isso possibilita, por exemplo, que seja realizada a busca por uma entrada cifrada em um banco de dados, também cifrado, sem permitir que nenhuma informação sobre a entrada e sobre o banco de dados seja extraída por algum atacante em controle da máquina onde a operação está sendo executada. Estudos anteriores [SMVB17], porém, mostraram que esta técnica é extremamente custosa em comparação a outras técnicas de segurança baseadas em *hardware*, sendo impraticável o seu uso em aplicações reais.

2.3.2 Isolamento de recursos

Outra forma de proteção de dados baseada em *software*, é o uso de técnicas de isolamento de recursos de uma máquina física, para que sejam utilizados, idealmente, por um único usuário ou aplicação.

Uma das técnicas de isolamento de recursos, amplamente utilizada na atualidade, é a de virtualização. Na técnica de virtualização, ambientes isolados, aqui chamados de **máquinas virtuais** (VMs, do inglês *Virtual Machines*), são criados com o objetivo de emular as funcionalidades de uma máquina física, e proteger estes ambientes contra outros que coexistam na mesma máquina hospedeira. Várias implementações de virtualização existem hoje no mercado [qem; Fou; kvm; vmw], e são utilizadas principalmente em ambientes de computação na nuvem, sejam elas privadas ou públicas.

Para a criação de VMs, é necessário o uso de um componente conhecido como hipervisor. Ele é responsável por criar e executar as VMs, assim como prover uma interface entre as VMs e o *hardware* real. Desta forma, o uso de virtualização para a proteção de dados sensíveis requer a adição deste componente ao TCB da aplicação como um todo.

Ao decorrer do tempo, várias vulnerabilidades foram encontradas em hipervisores, possibilitando que atacantes, a partir de VMs, ganhassem total controle sobre o hipervisor, o utilizassem para ganhar acesso à outras VMs residentes na mesma máquina [Kor09].

2.3.3 Considerações

Concluímos a seção apontando que soluções de segurança baseadas em *software* podem gerar um grande TCB nas aplicações que as utilizam, e consequentemente, podem aumentar a probabilidade de inserção de uma série de vulnerabilidades na aplicação. Além disso, outras soluções, apesar de já terem seu uso estabelecido na comunidade, como é o caso de criptografia assimétrica, ou estarem sendo bastante pesquisadas, como é o caso de criptografia homomórfica, acabam gerando um grande custo computacional, prejudicando o desempenho das aplicações que as utilizam.

2.4 Segurança de dados baseada em *hardware*

Como introduzido na Seção 1.1, o crescente volume de dados sensíveis sendo enviados e processados em ambientes de computação na nuvem levou à definição de um TEE, que facilitaria o desenvolvimento de soluções com garantias de segurança e privacidade de dados.

As soluções para segurança de dados baseadas em *hardware* geralmente se dão através da implementação de um TEE como parte de um microprocessador, com o objetivo de isolar dados de um processo em execução dos demais processos que executam na mesma máquina. Tais soluções, em geral, são computacionalmente mais eficientes que soluções baseadas em *software*, porém, para serem utilizadas, elas requerem o uso de *hardware* específico, que nem sempre está disponível nos ambientes onde as apicações serão executadas.

Como exemplos de implementações existentes no mercado, podemos citar as soluções que buscam separar o espaço de memória de uma máquina em duas áreas distintas, uma segura, e uma sem garantias de segurança, como é o caso do ARM TrustZone [ARM] e o AMD SME/SEV [AMD]. Nestas implementações, desenvolvedores de aplicações definem quais porções de dados devem ser considerados sensíveis, e o processador se encarrega de mantê-los em memória de uma forma criptograficamente segura. Em ambos os casos, todos os dados sensíveis, mesmo que pertencentes a diferentes aplicações são mantidos em um mesmo TEE. Uma outra implementação possível, que é a adotada pelo Intel SGX [MAB+13], busca separar o espaço de memória de uma máquina entre uma área insegura, e várias áreas isoladas seguras. Nesta solução, cada aplicação pode possuir uma área de memória segura para armazenar e processar dados sensíveis, e duas aplicações distintas não podem compartilhar

o mesmo espaço de memória seguro, alcançando total isolamento entre aplicações distintas.

Dentre as tecnologias para segurança baseada em *hardware* mencionadas, Intel SGX é a que se mostra como sendo mais promissora, e como tendo o maior número de pessoas envolvidas em pesquisas sobre a mesma, possivelmente devido à facilidade de acesso a processadores compatíveis com a tecnologia. Desta forma, SGX foi escolhida como objeto de estudo deste trabalho.

2.5 Trabalhos relacionados

Desde a publicação inicial [MAB$^+$13], feita em 2013, até a atualidade, diversas pesquisas sobre Intel SGX já foram divulgadas. Estas pesquisas são voltadas para diversos aspectos da tecnologia, que incluem seu uso em aplicações reais [KPA16; Fet16; SMVB17], ferramentas para facilitar o desenvolvimento de aplicações SGX [ATG$^+$16; TPV17; BPH15; SBB17], implementações de ataques de canal lateral [HCP17; MIE17; SWG$^+$17], e construção de enclaves mais seguros [GLS$^+$17; LPM$^+$17; KOA$^+$17].

Arnautov *et al.* propuseram a ferramenta SCONE, que tem como objetivo executar aplicações não modificadas dentro de enclaves SGX [ATG$^+$16]. Esta abordagem facilita a criação de aplicações seguras a partir de aplicações existentes, porém, ao inserir a aplicação por completo no enclave SGX, o seu TCB é aumentado. Esta solução vai de encontro com a recomendação da Intel, de que aplicações SGX devem manter apenas o mínimo de código possível dentro dos enclaves. Por outro lado, ela facilita o processo de criação de uma aplicação segura, pois não exige que o desenvolvedor aprenda a usar toda a API do Intel SGX para utilizar suas capacidades de segurança e privacidade.

Lind *et al.* propuseram a ferramenta Glamdring, que tem como objetivo facilitar o desenvolvimento de aplicações SGX, provendo o particionamento automático das mesmas [LPM$^+$17]. Para isso, o desenvolvedor deve utilizar anotações de código para marcar quais dados são considerados sensíveis na sua aplicação. Após a marcação feita manualmente pelo desenvolvedor, a ferramenta gera o código da aplicação particionado entre as partes segura e insegura, mantendo o mínimo possível de código dentro do enclave. Como veremos mais à frente, no Capítulo 4 esta estratégia não é a ideal em todos os cenários.

Por fim, é importante falar sobre os trabalhos sobre ataques de canal lateral. A Intel deixa

claro na documentação do SGX [Int] que a tecnologia, por si só, não é protegida contra ataques de canal lateral. Isto quer dizer que é possível deduzir informações sobre os dados sensíveis sendo processados através da observação do comportamento de outros componentes do sistema, como comportamento da *cache*, utilização de *CPU*, consumo de energia, *etc.* Götzfried *et al.* demonstraram ser possível realizar ataques à *cache* do processador para obter dados sensíveis do SGX [GESM17].

Apesar de ataques de canal lateral serem possíveis de se realizar em aplicações SGX, eles podem ser evitados através de boas práticas no desenvolvimento das aplicações seguras. Por exemplo, Gruss *et al.* propuseram um mecanismo capaz de evitar o mesmo tipo de ataques à *cache* do processador citado anteriormente [GLS+17].

2.6 Considerações

Devido ao crescente volume de dados sensíveis sendo enviados e processados em ambientes de computação na nuvem, faz-se necessário tomar medidas para prover proteções contra publicação e modificação de tais dados de forma indevida. Neste capítulo discutimos que tais medidas podem ser implementadas através do uso de técnicas baseadas em *hardware* ou *software*.

As soluções baseadas em *software* têm a vantagem de não depender de um *hardware* específico para serem utilizadas, porém, na maioria das vezes, incorrem no aumento do TCB das aplicações, e na perda de desempenho das mesmas.

As soluções baseadas em *hardware*, por sua vez, em geral têm um menor TCB, diminuindo o risco da adição de vulnerabilidades durante o processo de desenvolvimento das aplicações, e têm uma melhor performance em comparação a soluções baseadas em *software*, porém, necessitam ser executadas em ambientes com uma configuração de *hardware* específica disponível.

Uma das soluções baseadas em *hardware* é o Intel SGX. Ela pode ser usada para proteger código de aplicações contra modificação, e proteger dados de aplicações contra modificação e acessos não autorizados. Várias pesquisas sobre aplicações de Intel SGX, suas possíveis vulnerabilidades, e como mitigar estas vulnerabilidades, podem ser encontradas na literatura, porém elas não avaliam o impacto de vários aspectos na performance e nas garantias de

segurança de tais aplicações.

Considerando todos estes pontos, nos capítulos que seguem, detalhamos a tecnologia Intel SGX, apontamos como diversos fatores devem ser considerados durante o processo de desenvolvimento de aplicações seguras usando SGX, e propomos um conjunto de ferramentas que facilitam o desenvolvimento de aplicações SGX, bem como ajudam a superar alguns desafios do uso da tecnologia em ambientes reais.

Capítulo 3

Intel SGX

Nesse capítulo é detalhada a tecnologia Intel SGX. Primeiramente, são introduzidos os principais objetivos da tecnologia. Depois, é definido o modelo de ameaça considerado pela tecnologia. Em seguida, são apresentados alguns conceitos envolvidos no entendimento de seu funcionamento. Mais à frente, são considerados possíveis casos de uso para a tecnologia, e são apresentadas as principais limitações do SGX. Por fim, são feitas algumas considerações gerais sobre a tecnologia.

3.1 Introdução ao SGX

Intel SGX é uma tecnologia que tem como objetivo implementar um TEE. Ela busca garantir proteção contra modificação e divulgação de dados e código[2] de aplicações [MAB$^+$13]. Tal proteção é implementada em nível de *hardware*, onde o processador é responsável pelo controle de acesso a áreas de memória protegidas criptograficamente – tentativas de acesso ou modificação não autorizadas são tratadas como falha ou acesso a uma memória inexistente.

Intel SGX foi inicialmente definida no ano de 2013, e está comercialmente disponível desde o ano de 2015 em processadores da família Intel Core a partir da 6ª geração, baseados na microarquitetura Skylake, bem como em alguns processadores Intel Xeon v5.

Ao contrário de tecnologias semelhantes, como ARM TrustZone e AMD SME/SEV, onde considera-se uma separação única entre um mundo seguro e um mundo inseguro, com Intel SGX considera-se a existência de um mundo inseguro e vários mundos seguros, isolados

[2] O código de aplicações é protegido apenas contra modificação.

entre si, conhecidos como **enclaves**.

Desenvolvedores podem proteger as suas aplicações usando SGX, diretamente através do uso de um conjunto de instruções que foi incorporado às arquiteturas Intel 64 e IA-32, ou através do uso de um conjunto de APIs e bibliotecas, e um conjunto de enclaves arquiteturais, conhecidos, respectivamente, por *SGX SDK* e *SGX PSW*.

3.2 Modelo de ameaça

Antes de explicar como Intel SGX alcança os objetivos descritos na seção anterior, é necessário entender qual o modelo de ameaça considerado pela tecnologia. Intel SGX considera um modelo de ameaça onde um atacante é capaz de controlar e inserir código malicioso em programas com alto nível de privilégio, como sistemas operacionais e hipervisores, e até mesmo tenha acesso físico ao *hardware* onde os dados sigilosos se encontram.

Atacantes que consigam controle sobre um sistema operacional ou sobre um monitor de máquina virtual são capazes de obter e vazar informações sigilosas de usuários de diversas formas diferentes [GLX+17]. Já atacantes com acesso físico ao *hardware* podem realizar ataques de inicialização a frio, e recuperar chaves criptográficas que permanecem na DRAM, mesmo após a mesma ser removida da placa-mãe [HSH+09], e posteriormente usar essas chaves para obter dados sigilosos sem a devida autorização.

É importante ressaltar que Intel SGX não oferece proteção contra ataques de canal lateral [Int]. Vários ataques deste tipo já foram demonstrados, e provaram ter eficácia contra as proteções providas pelo SGX isoladamente [MIE17; GESM17; LSG+16]. Para evitar esse tipo de ataque, programadores devem tomar as devidas precauções [SCNS15; SLK+17; SLKP17].

3.3 Modelo de memória

Intel SGX utiliza uma hierarquia de memória conforme ilustrada na Figura 3.1.

Tendo o Intel SGX disponível no processador e habilitado, o BIOS reserva uma parte da memória principal (DRAM), chamada de *Processor Reserved Memory* (PRM), e a partir de então o acesso a essa área de memória passa a ser controlada pelo processador.

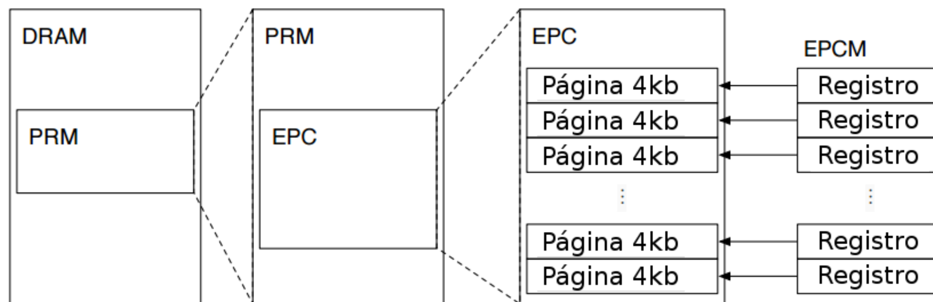


Figura 3.1: Hierarquia de memória do Intel SGX

Parte da PRM é destinada a uma área de memória conhecida como *Enclave Page Cache* (EPC). A EPC serve para armazenar páginas de enclaves SGX. Cada uma dessas páginas é associada a um único enclave SGX. O processador é responsável por garantir que essas páginas só sejam acessíveis pelo enclave associado a elas.

Por fim, para auxiliar o controle de acesso à memória, uma estrutura adicional precisa ser criada. Essa estrutura é conhecida como *Enclave Page Cache Map* (EPCM), e é utilizada para armazenar metadados – permissões de leitura, escrita e execução, tipo, endereço linear, estado, etc. – sobre as páginas que residem na EPC.

3.3.1 Paginação da EPC

De modo a aumentar o número de aplicações protegidas que podem ser suportadas de forma concorrente, a arquitetura SGX oferece instruções que permitem que sistemas operacionais façam paginação da EPC de forma segura. O processo de paginação da EPC também pode ser acionado caso o tamanho dos dados carregados em um enclave exceda o limite do tamanho da EPC.

O processo de paginação da EPC, em geral, é muito mais custoso que um processo de paginação normal de um sistema operacional. Isso se deve ao fato de que as seguintes medidas de segurança devem ser tomadas pelo SGX:

1. Uma página de enclave só pode ser removida da EPC depois que todas as traduções em cache para essa página tenham sido removidas de todos os processadores lógicos;

2. O conteúdo da página do enclave precisa ser cifrado antes de ser escrito na memória principal;

3. Ao recarregar a página na EPC, informações sobre o tipo, permissões, endereço virtual, conteúdo, e o enclave dono da página devem ser verificadas, de modo a garantir que as mesmas correspondem à página que foi removida;

4. Apenas a última versão da página removida pode ser recarregada.

3.4 Modelo de programação

Antes de desenvolver aplicações que fazem uso de Intel SGX, desenvolvedores devem ter em mente que nem toda a área de sua aplicação deve ser protegida. Na verdade, a Intel sugere que o mínimo possível da aplicação seja executada dentro de enclaves SGX – apenas a parte da aplicação que lida com dados sigilosos.

Dessa forma, aplicações SGX devem ser separadas em duas partes:

Parte segura/confiável – executada dentro de enclaves SGX. Deve ser usada para armazenamento e processamento de dados sensíveis/sigilosos. Por ser protegida, há um custo adicional de processamento em comparação com a execução de uma aplicação normal, relativo ao processo de cifrar e decifrar páginas dos enclaves. Código sendo executado na parte segura não é capaz de executar chamadas de sistema.

Parte insegura/não confiável – executada na memória principal da máquina. Serve como um invólucro para a parte segura da aplicação, podendo ser usado tanto para o processamento de dados que não sejam sigilosos, quanto como uma interface de comunicação com outras aplicações ou escrita de arquivos.

Para definir como as duas partes da aplicação SGX se comunicam entre si, um arquivo no formato EDL – *Enclave Description Language* – deve ser provido pelo desenvolvedor. Neste arquivo são definidas funções onde a parte insegura da aplicação envia dados e executa algum processamento dentro da parte segura (ECALLs), e funções onde a parte segura da aplicação envia dados e executa algum processamento na parte insegura (OCalls).

Juntamente com o *SGX SDK* é fornecida uma ferramenta conhecida como Edger8r. Ela é responsável por ler arquivos EDL e gerar interfaces de comunicação entre as partes segura e insegura de uma aplicação SGX.

3.5 Funcionalidades importantes

Intel SGX possui algumas funcionalidades que auxiliam no processo de estabelecer confiança em uma aplicação, bem como no compartilhamento seguro de dados sigilosos entre diferentes aplicações, ou entre diferentes versões de uma mesma aplicação. Essas funcionalidades são discutidas a seguir.

3.5.1 Medição de enclaves

A arquitetura Intel SGX é responsável por estabelecer identidades de enclaves. No jargão do Intel SGX, o processo de estabelecimento de identidade de um enclave é conhecido como **medição de enclave**. Associadas a cada enclave estão duas identidades distintas:

MRENCLAVE – Também referido como identidade de enclave, é o resultado da função de *hash* SHA-256 do registro de toda a atividade feita ao construir um enclave SGX. Em outras palavras, essa identidade é calculada baseada em todo o conteúdo carregado em um enclave, *i.e.*, conteúdos das páginas do enclave, posição relativa das páginas no enclave, e *flags* de segurança associadas às páginas.

MRSIGNER – Também referido como identidade de selagem, é a identidade de quem assinou o enclave SGX, tipicamente o desenvolvedor do enclave. Múltiplos enclaves podem ser assinados por um mesmo desenvolvedor, fazendo com que todos tenham a mesma identidade de selagem.

3.5.2 Atestação de enclaves

Atestação é o processo de provar que um determinado programa foi devidamente instanciado em uma plataforma. No caso do Intel SGX, o processo de atestação garante que um programa está sendo executado dentro de um enclave SGX, e que não foram feitas modificações no código do mesmo. O processo de atestação pode ser feito local ou remotamente.

Atestação local

Desenvolvedores podem escrever duas ou mais aplicações seguras, executadas em enclaves, que podem cooperar entre si, para realizar funções que não sejam de suas próprias compe-

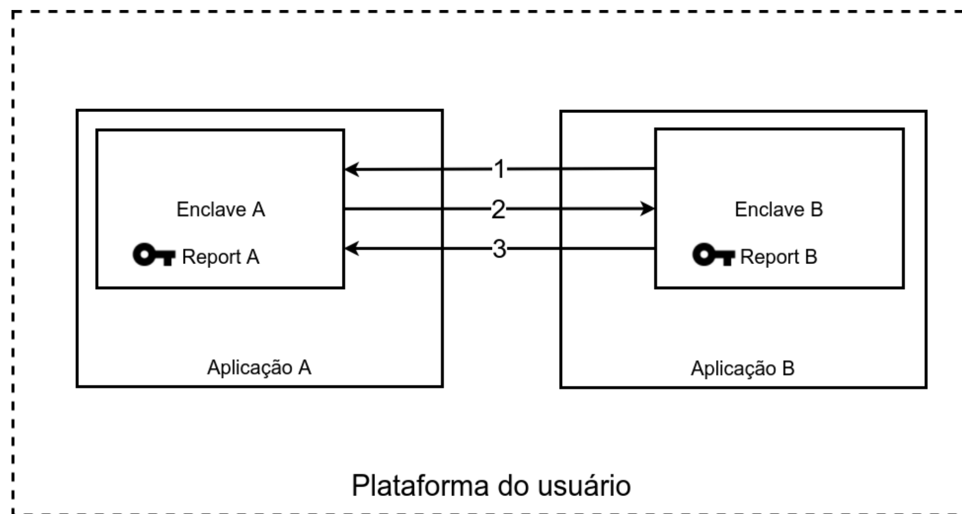


Figura 3.2: Atestação local

tências. Para isso, é provido um mecanismo que permite que enclaves se autentiquem entre si, através do uso das instruções EREPORT e EGETKEY.

O processo completo de atestação local de enclaves, ilustrado na Figura 3.2, acontece da seguinte forma:

1. Após um canal de comunicação ser estabelecido entre os enclaves A e B[3], o enclave A obtém o MRENCLAVE do enclave B.

2. O enclave A executa a instrução EREPORT, usando como parâmetro o MRENCLAVE do enclave B, criando assim uma estrutura, conhecida como *REPORT*, contendo o seu próprio MRENCLAVE e MRSIGNER, e assinada com uma chave, *Report Key*, acessível diretamente apenas pelo enclave B. Após obtido, o *REPORT* é enviado pelo enclave A para o enclave B.

3. Tendo recebido o *REPORT* do enclave A:

   (a) O enclave B executa a instrução EGETKEY para obter a sua chave, e com ela, verifica a assinatura do *REPORT* recebido do enclave A. Caso a assinatura seja verificada com sucesso, o enclave B pode ter certeza que o enclave A está sendo executado na mesma plataforma, que por sua vez, está de acordo com o modelo de segurança SGX.

[3]Toda comunicação de aplicações SGX passa pela parte insegura da aplicação.

(b) O enclave B pode então verificar se o MRENCLAVE do enclave A, contido no *REPORT* recebido, corresponde ao da aplicação com a qual ele deseja se comunicar. Em caso positivo, o enclave A pode agora ter certeza de estar se comunicando com a aplicação correta.

O mesmo processo pode ser repetido para que o enclave A confie no enclave B.

Atestação remota

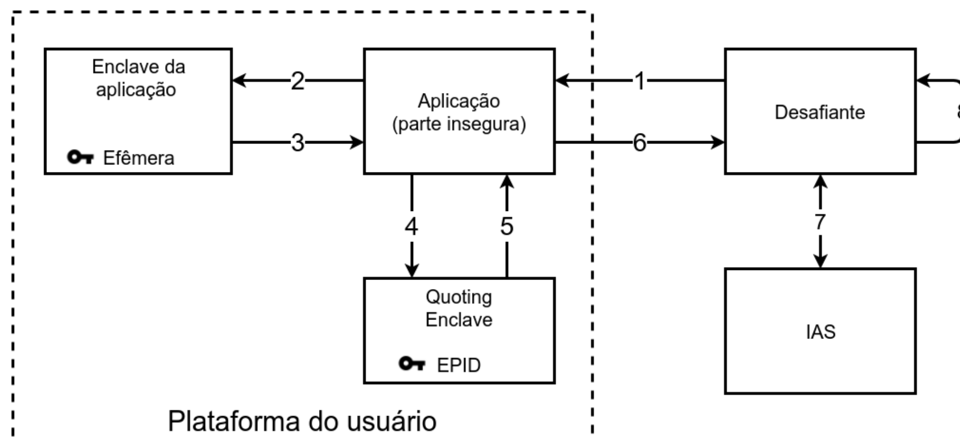


Figura 3.3: Atestação remota

O processo de atestação local usa um sistema de chaves simétricas, onde a chave só é acessível através da execução das instruções *EREPORT* e *EGETKEY*. No caso de uma atestação remota (RA, do inglês *Remote Attestation*), a aplicação que deseja ganhar confiança em uma aplicação SGX não é capaz de executar a instrução *EGETKEY*, uma vez que as partes são executadas em plataformas distintas. Para o processo de atestação remota, é necessário o uso de um sistema de chaves assimétricas.

Para permitir tal processo, um enclave especial, conhecido como *Quoting Enclave* (QE), é provido pela Intel. O QE é responsável por usar o processo de atestação local, e transformar o *REPORT* gerado no processo em uma outra estrutura, conhecida como *QUOTE*, assinada com uma chave assimétrica, acessível apenas pelo QE. Além disso, a Intel provê um serviço de atestação (IAS), que é capaz de verificar a autenticidade da assinatura de um *QUOTE*.

O processo completo de atestação remota, mostrado na Figura 3.3, acontece da seguinte forma:

1. Após estabelecida uma comunicação entre um desafiante e a plataforma onde se encontra a aplicação SGX[3], o desafiante pede uma prova de que a aplicação está sendo devidamente executada dentro de um enclave SGX. Junto a esse pedido, o desafiante envia a sua chave pública.

2. Recebendo tal pedido, o enclave procede criando uma estrutura, *MSG1*, que contém sua chave pública, e a envia para o desafiante.

3. Em seguida, o desafiante é capaz de derivar uma chave simétrica, *SMK*, através do método *Diffie-Hellman*, usando a sua chave privada e a chave pública do enclave. Tendo essa chave simétrica em mãos, o desafiante deve gerar uma outra estrutura, *MSG2*, e enviar para o enclave. A *MSG2* contém:

   (a) a chave pública do desafiante;

   (b) um identificador de registro do desafiante para com o IAS;

   (c) uma assinatura da concatenação de ambas as chaves públicas, usando a chave privada do desafiante;

   (d) um código de autenticação de mensagem (MAC) de toda a mensagem, usando a chave *SMK*, que pode ser verificada pelo enclave.

4. O enclave deve então gerar o seu *REPORT*, e enviar para o QE, para que ele gere o *QUOTE* do enclave.

5. O enclave gera um *MAC* do *QUOTE* recebido, usando a chave *SMK*, e a envia para o desafiante.

6. O desafiante verifica o *MAC*, para garantir que está se comunicando com o mesmo enclave ainda, e após verificar com sucesso, envia o *QUOTE* para o *IAS*.

7. O *IAS* verifica se a assinatura e a estrutura do *QUOTE* são válidos, ou seja, foram gerados por um enclave SGX, e envia um resposta para o desafiante.

8. Tendo confirmado a validade do *QUOTE*, o desafiante verifica se o *MRENCLAVE*, contido no *QUOTE*, é o esperado, completando assim o processo de atestação remota.

Após completar o processo de atestação remota, ambas as partes envolvidas podem usar a chave *SMK* para enviar mensagens entre si de forma segura.

3.5.3 Selagem de dados

Enquanto um enclave está instanciado, o *hardware* garante a integridade e confidencialidade de seus dados. Entretanto, ao encerrar o processo de um enclave, todo o seu conteúdo é destruído, e qualquer dado que esteja no enclave será perdido.

Se os dados do enclave seriam usados posteriormente, *e.g.*, em uma futura execução do enclave, é necessário que esses dados sejam armazenados fora da área protegida do enclave. Com este objetivo, o SGX provê uma funcionalidade para selar estes dados e armazená-los em disco de forma segura. A funcionalidade de selagem de dados pode ser feita de duas formas:

Selagem usando a Identidade de enclave – Utilizada quando apenas o mesmo código de enclave deve ser capaz de acessar os dados selados. Utiliza o MRENCLAVE para selar os dados. Com essa forma de selagem, não é possível transferir dados para aplicações diferentes, ou até mesmo para uma nova versão da mesma aplicação.

Selagem usando a Identidade de selagem – Utilizada quando se deseja transferir dados de um enclave para uma aplicação distinta ou para uma nova versão da mesma aplicação. Utiliza o MRSIGNER para selar os dados. Os dados selados serão acessíveis por qualquer enclave que tenha sido assinado pelo mesmo desenvolvedor, o que resulta em um mesmo MRENCLAVE. Também é útil quando se deseja atualizar uma aplicação, *e.g.*, quando deseja-se usar um código mais eficiente ou mais seguro na aplicação.

3.6 Ciclo de vida de aplicações SGX

Aplicações SGX devem ser iniciadas sem conter nenhum dado sigiloso na memória. Depois que o enclave SGX já está sendo executado, o processo de atestação de enclaves pode ser usado para obter confiança em uma aplicação SGX, e estabelecer um canal de comunicação seguro, para enviar dados para serem processados dentro do enclave da aplicação SGX.

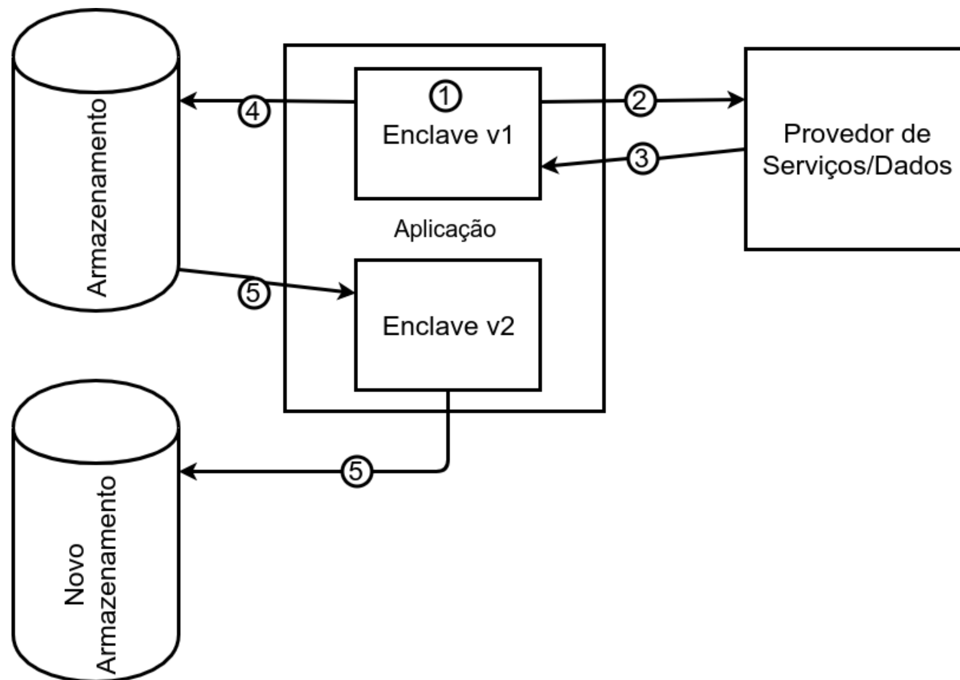


Figura 3.4: Ciclo de vida de aplicações SGX

A aplicação SGX pode cifrar os seus dados e armazená-los para um uso futuro, usando a funcionalidade de selagem de dados.

A Figura 3.4 ilustra os passos dados para completar esse ciclo.

1. Inicialização de enclave – A parte insegura da aplicação é responsável por criar o enclave. No processo de inicialização do enclave o seu MRENCLAVE é calculado.

2. Atestação – A aplicação SGX usa o processo de atestação remota para se tornar confiável e estabelecer um canal de comunicação seguro.

3. Provisionamento – A aplicação SGX recebe os dados sigilosos através do canal de comunicação seguro.

4. Selagem – O enclave usa a funcionalidade de selagem para armazenar seus dados sigilosos de forma que uma nova versão da aplicação possa acessá-los.

5. Atualização de aplicação – Uma aplicação SGX pode necessitar de uma atualização, e sua versão atualizada recebe os dados da versão anterior.

3.7 Intel SGX SDK e PSW

O SDK é um conjunto de ferramentas e APIs com funções de alto nível (em C/C++) que permite a criação e uso de enclaves SGX de forma mais fácil para o desenvolvedor. Juntamente com o SDK estão presentes duas ferramentas essenciais no ciclo de vida de aplicações SGX. São elas:

- sgx_edger8r – responsável por criar as interfaces de comunicação entre as partes segura e insegura da aplicação.
- sgx_sign – responsável por assinar um enclave SGX, usando uma chave privada, que resultará no MRSIGNER do enclave.

Já o PSW, distribuído junto do SDK, trata-se de um conjunto de enclaves especias da Intel, usados para instanciar e realizar o processo de medição enclaves. Entre os enclaves mais importantes estão:

- *Launch Enclave* (LE) – responsável por instanciar enclaves SGX.
- *Quoting Enclave* (QE) – responsável por gerar a estrutura conhecida como QUOTE, usada no processo de atestação remota de enclaves SGX.
- *Provisioning enclave* (PvE) – responsável por prover chaves necessárias ao processo de atestação de um enclave.

Ambos o SDK e o PSW estão disponíveis tanto para a plataforma *Windows* quanto para a plataforma *Linux*[4].

3.8 Casos de uso de SGX

Considerando aplicações do tipo cliente-servidor que usam SGX, podemos observar dois modelos: (i) proteção de dados no lado do cliente, ou (ii) proteção de dados no lado do servidor.

[4] https://software.intel.com/en-us/sgx-sdk/download

Máquina cliente como *host* do enclave SGX

Neste modelo de aplicação, o objetivo é ter uma aplicação na máquina cliente que **obtém** dados sensíveis de algum provedor de serviços. Antes de receber tais dados, a aplicação precisa provar para o provedor de serviços que está sendo executada em um enclave SGX.

Como exemplo de aplicação que pode tirar proveito de SGX para proteção de dados sigilosos, podemos citar uma aplicação de *Internet Banking*. Neste caso, a entidade financeira em posse dos dados de um cliente precisa se certificar que os dados sensíveis, *e.g.*, saldo e extrato de conta, só serão acessíveis pelo cliente a quem a conta pertence. Para alcançar este objetivo, a entidade financeira pode desenvolver uma aplicação que execute dentro de um enclave SGX, e que seja atestável pelo seu provedor de serviços.

Outro caso de uso deste modelo que podemos citar é o de gestão de direitos digitais (DRM, do inglês *Digital Rights Management*). Neste caso, um enclave SGX pode ser utilizado para evitar que cópias de conteúdos digitais sejam feitas sem a devida autorização. Uma grande empresa que atualmente está utilizando Intel SGX para este fim é a Netflix[5]. Ela utiliza as capacidades do SGX para proteger vídeos transmitidos com qualidade 4K.

Máquina servidora como *host* do enclave SGX

Neste modelo de aplicação, o objetivo é ter uma aplicação na máquina cliente que **envia** dados sensíveis para algum provedor de serviços. Antes de enviar tais dados, a aplicação precisa ter certeza que está se comunicando com um provedor de dados confiável, *i.e.*, está executando em um enclave SGX.

Um caso de uso que podemos citar como exemplo deste modelo é o de agregação de dados de *smart meters* [SMVB17]. Neste caso, *smart meters* coletam dados de consumo de energia de residências com um alto nível de detalhe. Estes dados podem ser usados para derivar informações sobre os indivíduos que ali residem[6]. Desta forma, é necessário prover um certo nível de proteção destes dados, para que eles só possam ser utilizados de forma agregada por distribuidores de energia.

[5] `https://help.netflix.com/pt/node/55763`

[6] `https://www.cnet.com/news/researchers-find-smart-meters-could-reveal-favorite-tv-shows/`

3.9 Limitações do Intel SGX

Como qualquer outra tecnologia, Intel SGX também possui limitações. As mais importantes delas são discutidas a seguir.

3.9.1 Privacidade de código

Apesar de dar garantias quanto à proteção do código de aplicações contra modificações não autorizadas, Intel SGX não garante a privacidade desse mesmo código. Em outras palavras, arquivos contendo todo o código de um enclave SGX são carregados na memória principal de forma não cifrada.

Há vários cenários onde desenvolvedores desejam manter a privacidade de seu código, portanto essa limitação deve ser levada em consideração por desenvolvedores, uma vez que atacantes poderiam facilmente usar ferramentas como o IDA [ida] para gerar um pseudo-código bastante similar à aplicação original.

3.9.2 Ligação estática de bibliotecas

Aplicações que usam bibliotecas de terceiros precisam ligar tais bibliotecas a seus enclaves estaticamente. Isso pode resultar em um enclave demasiadamente grande, e, consequentemente desperdiçar espaço da memória, que é um recurso bastante escaço para o SGX. Outra consequência desta limitação é que qualquer mínima modificação em uma única biblioteca pela aplicação requer que toda a aplicação SGX seja completamente recompilada e reimplantada.

3.9.3 Memória disponível

O espaço da memória principal usado pela PRM deve ser reservado pelo BIOS no momento em que o sistema está sendo inicializado. Além disso, toda a EPC deve residir dentro da PRM. Na versão atual do SGX, a parte da memória reservada para a PRM é limitada a um tamanho máximo de apenas $128\ MB$ por máquina. Caso o espaço necessário seja maior do que o disponível, há um grande aumento no tempo de processamento, devido ao custoso processo de paginação entre EPC e DRAM, conforme descrito na seção 3.3.1. Na seção 4.1

discutiremos melhor os impactos da gerência de memória protegida.

3.9.4 Vulnerabilidades

Conforme descrito na Seção 3.2, aplicações SGX podem ser vulneráveis a ataques de canal lateral. Esse tipo de ataque permite que um adversário colete estatísticas sobre execução da **CPU**, e as utilize para deduzir características sobre o programa em execução. Apesar de SGX não prover proteção contra esse tipo de ataque, é possível previní-lo, tomando os devidos cuidados[7].

É importante notarmos também que a solução de SGX é composta por componentes – *drivers*, bibliotecas, e instruções – complexos, e, portanto, dificilmente é completamente livre de erros, como qualquer outro *software*. Além disso, desenvolvedores de aplicações SGX podem cometer erros e escrever enclaves suscetíveis a vulnerabilidades como *buffer overflow* e formatação não verificada de *strings*.

Uma técnica de segurança que tem suas funcionalidades limitadas com o uso de SGX é a de aleatorização do leiaute do espaço de endereçamento (ASLR, do inglês *Address Space Layout Randomization*). Esta técnica torna aleatória a organização na memória de diversas partes de um processo, evitando assim que atacantes usem as vulnerabilidades mencionadas anteriormente para desviar a execução de um programa para uma função explorada carregada na memória. No caso do SGX, por ter um espaço de endereçamento reduzido, ataques de força bruta podem ser utilizados para descobrir o endereço desejado. Em experimentos realizados, foi verificado que em diferentes execuções, uma variável de um enclave SGX tem seu endereço modificado em apenas dois *bytes*, significando que a aleatorização tem aproximadamente apenas 65536 possibilidades. Essa quantidade de possibilidades é muito pequena, uma vez que atacantes podem aumentar a probabilidade de ataques com sucesso através da injeção de sequências de instruções **NOP** antes de um código malicioso.

[7] `https://github.com/chowes/sgx-side-channel/blob/master/sgx-side-channel.pdf`

3.10 Considerações

Intel SGX é uma tecnologia bastante promissora, oferecendo garantias de segurança e privacidade de dados de usuários contra ataques provenientes até mesmo de atacantes que tenham obtido privilégio indevidamente, ou que tenham acesso físico à máquina que hospeda os dados sigilosos. Como qualquer outra tecnologia, porém, também possui limitações que podem pôr em risco a sua aplicabilidade no mundo real. É necessário, portanto, fazer uma análise dos principais desafios a serem enfrentados durante o processo de desenvolvimento de aplicações que façam uso desta tecnologia, antes de pô-la em um ambiente de produção.

Capítulo 4

Características-chaves em aplicações SGX

Neste capítulo, detalhamos a abordagem utilizada neste trabalho, que envolve a avaliação do impacto de características de aplicações SGX na performance e na segurança providas pelas mesmas. Para cada uma das características avaliadas neste trabalho, realizamos vários experimentos. Para os experimentos dispomos de uma máquina Dell Optiplex 5040, com um processador Intel Core i7-7700, com 4 núcleos operando a $3{,}4\ GHz$, e $8\ GB$ de memória principal disponível.

De forma a mensurar o impacto de cada uma das características, coletamos e analisamos duas métricas: (i) o tamanho do TCB gerado – lembrando que quanto maior o TCB, maior a probabilidade de se inserir *bugs* e vulnerabilidades no mesmo – impactando diretamente a segurança de uma aplicação, e (ii) tempo de processamento das aplicações.

Em cada uma das seções a seguir, apresentamos as principais características que devem ser consideradas durante o desenvolvimento de aplicações SGX, incluindo uma descrição dos experimentos realizados, resultados obtidos nos experimentos, e uma discussão sobre os impactos de tais características nas aplicações em questão.

4.1 Gerência de memória

Conforme apontado na Seção 3.9, uma das limitações de SGX é a pequena área de memória protegida pela tecnologia, com apenas $128\ MB$ disponíveis. Nesta seção, avaliamos os

possíveis impactos na performance de aplicações SGX, caso elas venham a ter um consumo de memória que exceda o limite de memória disponível.

Código Fonte 4.1: Pseudocódigo do experimento de acesso aleatório à memória.

```
1  #define MB_SIZE 1024*1024
2
3  char tmp;
4
5  char perform_random_memory_access(int size_in_mb) {
6     size_t total_size = size_in_mb * MB_SIZE;
7     char *allocated_memory = (char *) malloc(total_size);
8
9     int i;
10    for (i=0; i<total_size; ++i) {
11       int pos = rand() % total_size; // A função rand() gera um nú
            mero aleatório.
12       tmp ^= allocated_memory[pos]; // Para evitar otimizações feitas
             pelo compilador.
13    }
14    free(allocated_memory);
15    return tmp;
16 }
```

Experimentos

Para avaliar o impacto do consumo de memória em aplicações SGX, realizamos um experimento que compara o desempenho de acesso aleatório à memória em uma aplicação escrita em C, sem garantias de segurança, e em uma aplicação que usa SGX, com garantias de segurança. As implementações de ambas aplicações são praticamente idênticas, diferindo apenas que a alocação e acesso aleatório de memória da aplicação SGX são feitos dentro de um enclave, enquanto na primeira implementação são feitos de forma insegura.

O código utilizado em ambas implementações é exibido no Código Fonte 4.1, e é descrito da seguinte forma:

1. A aplicação aloca um espaço de memória do tamanho passado como parâmetro,

associando-o à variável *allocated_memory*, que será utilizada como um *array* de elementos do tipo *char* (linha 7);

2. A aplicação gera um número inteiro aleatório pos, que indica o índice do *array* a ser acessado (linha 11);

3. A aplicação acessa o elemento presente no índice aleatório gerado (linha 12);

4. A aplicação repete os passos 2 e 3 pelo número de vezes que corresponde ao total de posições do *array* alocado (linhas 10-13);

5. A aplicação libera o espaço alocado para a variável *allocated_memory* (linha 14).

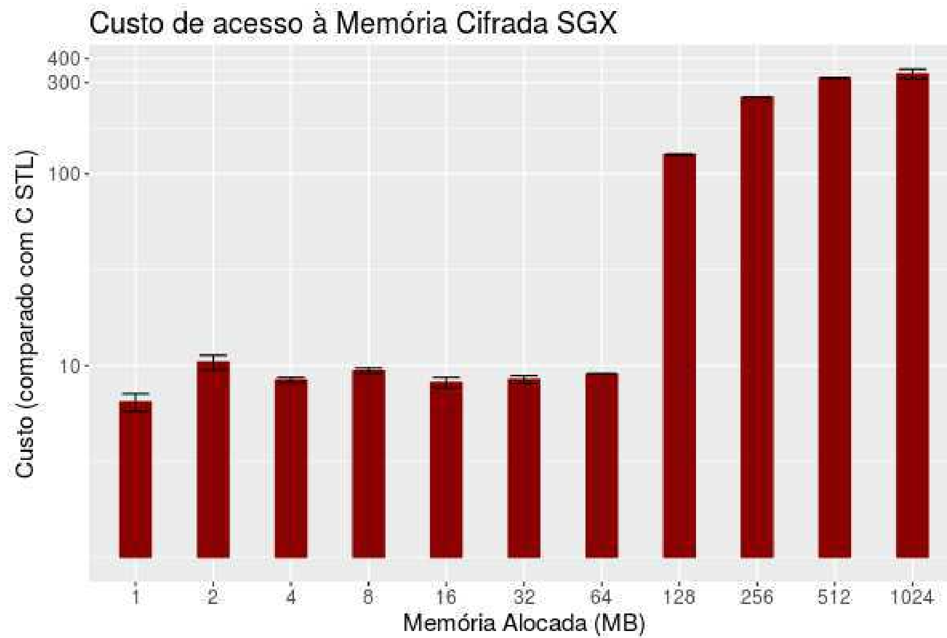
Neste experimento o espaço de memória utilizado pelas aplicações foi variado entre $1\ MB$ e $1024\ MB$, em uma escala logarítmica.

Tendo em vista que, neste experimento, apenas o tamanho da memória alocada e acessada aleatoriamente é variada, coletamos apenas a métrica do tempo de processamento tanto para a aplicação insegura, em C puro, e a aplicação segura, utilizando SGX.

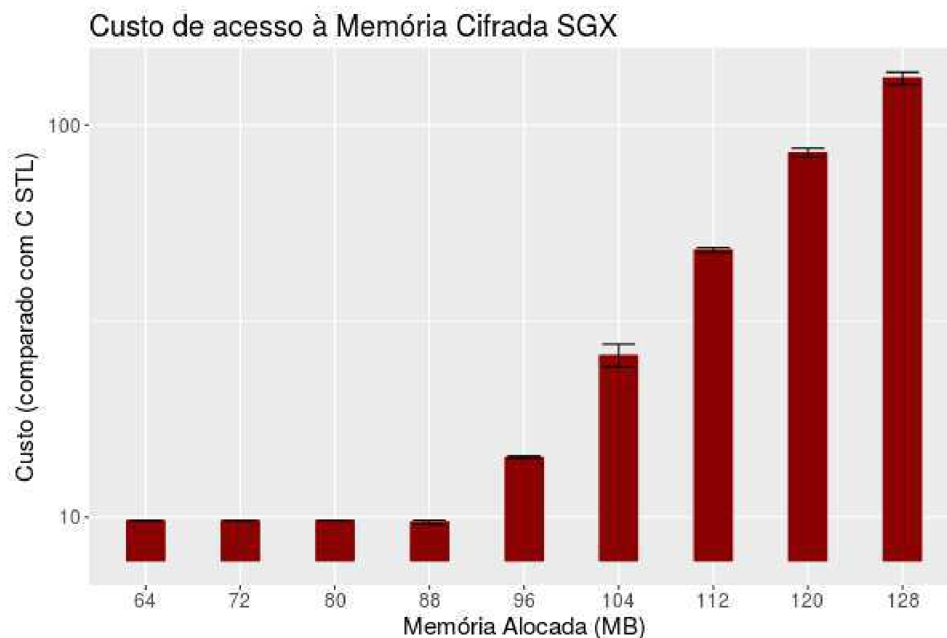
Resultados obtidos

Para cada uma das configurações descritas anteriormente, trinta repetições do experimento foram executadas. A Figura 4.1 mostra a relação entre os tempos de execução da aplicação SGX e da aplicação C pura, considerando cada uma das configurações determinadas anteriormente. Na Figura 4.1a, notamos que aplicações SGX que usam até $64\ MB$ de memória têm um custo médio de acesso à memória aproximadamente 10 vezes mais alto que aplicações sem garantias de segurança. Já aplicações que consumam $128\ MB$ de memória ou mais têm um custo médio de acesso à memória pelo menos 100 vezes, e chegando a ser 380 vezes mais alto que o de aplicações inseguras.

Para entender melhor o comportamento do custo de acesso à memória em aplicações SGX, repetimos o experimento descrito anteriormente, porém variando o espaço de memória utilizado entre $64\ MB$ e $128\ MB$, com incrementos de $8\ MB$. Na Figura 4.1b, podemos observar que aplicações que demandam até 88MB de memória têm um custo médio de acesso à memória 10 vezes mais alto que aplicações inseguras. Aplicações que ultrapassem $88\ MB$ têm um custo de acesso que cresce linearmente à medida que mais memória é demandada.



(a) Tamanho de memória variando entre 1MB e 1024MB



(b) Tamanho de memória variando entre 64MB e 128MB

Figura 4.1: Sobrecarga média de acesso à memória SGX comparado com acesso à memória em C puro, com um intervalo de 95% de confiança.

Discussão

Considerando os resultados obtidos nestes experimentos, podemos concluir que a quantidade de memória consumida e uma gerência de memória apropriada são características de

extrema importância no desenvolvimento de aplicações SGX, visto que um alto consumo de memória pode torná-las impraticáveis no mundo real. Dados os resultados obtidos nos experimentos aqui apresentamos, recomendamos que desenvolvedores busquem criar e utilizar suas aplicações de um modo que ela não consuma além do limite de memória disponível na EPC, sob o risco de obter uma sobrecarga demasiadamente alta no tempo de processamento de sua aplicação, caso essa recomendação não seja seguida.

É importante notar que, apesar de termos reservado um espaço de $128\ MB$ para a PRM, apenas cerca de $90\ MB$ ficam disponíveis para a EPC. Isso de deve ao fato de que parte da PRM é utilizada pela estrutura EPCM e pelos enclaves especiais da Intel.

4.2 Particionamento de dados

Existem aplicações que precisam processar um grande volume de dados, também conhecido como processamento de *Big Data*. Considerando esse tipo de aplicações, desejamos analisar qual a melhor forma para processar este grande volume de dados, desde que sejam independentes entre si. Nesta seção, procura-se, mais precisamente, saber se, e como, este grande volume de dados independentes deve ser particionado para ser processado dentro de enclaves SGX, considerando-se a disponibilidade de um único enclave. Em outras palavras, deseja-se identificar como particionar um grande volume de dados para processar suas partes independentes, sequencialmente, da forma mais eficiente possível.

Experimentos

A característica que avaliamos nesta seção envolve apenas variações na carga útil a ser processada, e não na aplicação SGX usada para processar estes dados. Desta forma, o tamanho do TCB não será uma métrica coletada e avaliada nestes experimentos, uma vez que o tamanho do TCB permanece constante em todas as configurações. A métrica coletada e avaliada nesta seção é o tempo de execução necessário para completar a tarefa de computar a soma de todos os elementos de um *array* de números inteiros com 1GB de tamanho.

Este experimento, foi dividido em duas partes. Em um primeiro momento, buscou-se avaliar o tempo de processamento do *array*, considerando-se partições com tamanhos de 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, e $1024\ MB$. Em um segundo momento, tendo em vista

as observações de perda de desempenho quando enclaves excedem o tamanho disponível na EPC, feitas na Seção 4.1, buscou-se avaliar também o tempo de processamento do *array*, considerando-se partições com tamanhos de 72, 80, 88, 96, 104, e 112 MB.

Código Fonte 4.2: Pseudocódigo do experimento de particionamento de dados.

```
1   // Na parte insegura da aplicação
2
3   #define MB_SIZE 1024*1024
4
5   long sum_array(int chunk_size_in_mb){
6      int array_size = 1024 * MB_SIZE;
7      int chunk_size = chunk_size_in_mb * MB_SIZE;
8      int *array = generate_random_array(array_size); // Cria um array
          com tamanho 1GB
9      long sum = 0;
10     int i;
11     for (i=0; i<array_size/chunk_size; ++i){
12        sum += enclave_sum(&array[i*chunk_size], chunk_size);
13     }
14     free(allocated_memory);
15     return sum;
16  }
17
18  // No enclave da aplicação
19
20  long enclave_sum(int *array, int array_size){
21     long sum = 0;
22     int i;
23     for (i=0; i<array_size; ++i){
24        sum += array[i];
25     }
26     return sum;
27  }
```

O código utilizado neste experimento é exibido no Código Fonte 4.2, e é descrito da seguinte forma:

1. A aplicação aloca um espaço de memória com tamanho de 1 GB, associando-o à

Tabela 4.1: Número de chamadas à função *enclave_sum* necessárias para computar a soma dos elementos de um *array* com 1GB de tamanho, de acordo com o tamanho, em MB, das partições.

Tam. da partição	**Num. de chamadas**
1	1024
2	512
4	256
8	128
16	64
32	32
64	16
72	15
80	13

Tam. da partição	**Num. de chamadas**
88	12
96	11
104	10
112	10
120	9
128	8
256	4
512	2
1024	1

variável *array*, que é um *array* de elementos do tipo *int* (linha 8);

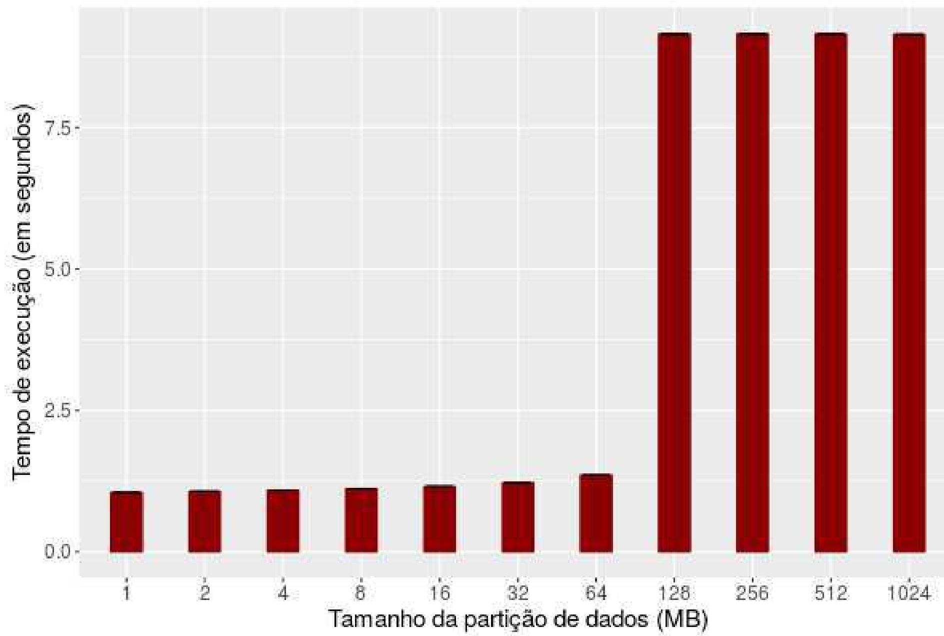
2. A aplicação envia para o enclave uma partição dos dados para calcular a soma dos elementos desta partição (linha 12);

3. O enclave da aplicação calcula a soma dos elementos da partição recebida (linhas 23-25);

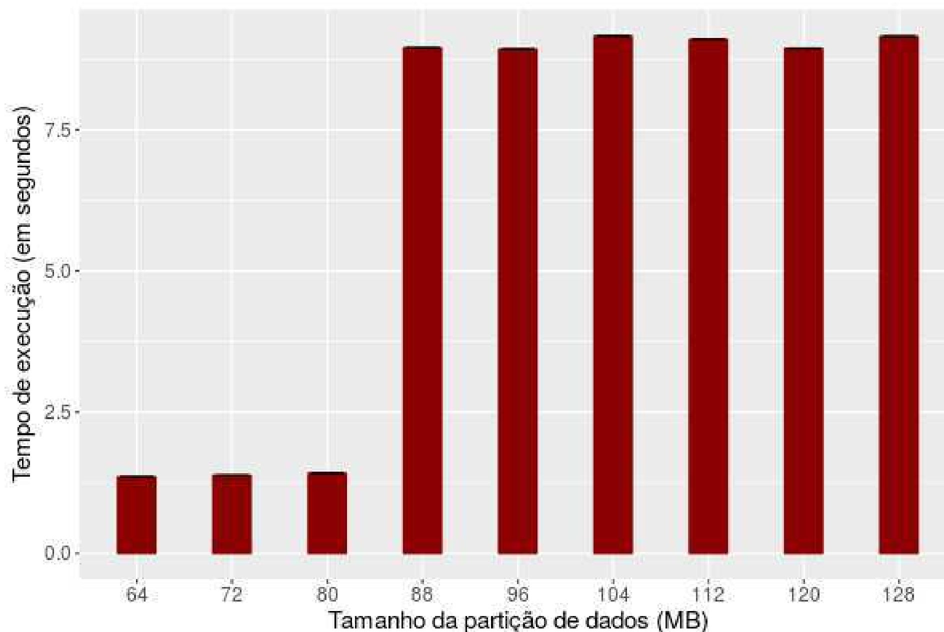
4. Repetimos os passos 2 e 3 até que a soma de todas as partições tenham sido computadas (linhas 11-13);

5. A aplicação libera o espaço alocado para a variável *array* (linha 14).

É importante salientar que o tamanho do *array* copiado para o enclave a cada chamada à função *enclave_sum* é exatamente o tamanho passado pelo parâmetro *array_size* da mesma função. O número de chamadas feitas à função *enclave_sum*, considerando cada um dos tamanhos de partição propostos, pode ser visto na Tabela 4.1.



(a) Tamanho da partição variando entre 1MB e 1024MB



(b) Tamanho da partição variando entre 64MB e 128MB

Figura 4.2: Tempos médios para calcular a soma dos elementos de um *array* com tamanho de 1GB, variando o tamanho das partições utilizadas no cálculo das somas parciais, com um intervalo de 95% de confiança.

Resultados obtidos

A Figura 4.2 mostra o tempo total de processamento do *array* com tamanho de $1\ GB$, em cada um dos cenários propostos. A partir dos resultados exibidos na Figura 4.2a, notamos

que, para o particionamento de dados feito com partições de até $64\ MB$, à medida que o tamanho das partições cresce, o tempo total de execução da tarefa também cresce – o tempo de execução usando partições com tamanho de $64\ MB$ é cerca de 28% mais alto que o tempo de execução usando partições com tamanho de $1\ MB$. Já para o particionamento de dados feito com partições de $128\ MB$ ou mais, é notável a sobrecarga gerada pelo consumo de memória superior ao disponibilizado na EPC, porém não há, estatísticamente, diferença entre o uso de diferentes tamanhos de partições que excedam este limite – o tempo de execução da tarefa usando partições de $1024\ MB$ é cerca de $0{,}6\%$ mais baixo que o tempo de execução usando partições de $128\ MB$.

Analisando os dados exibidos na Figura 4.2b, podemos dizer que a sobrecarga no tempo de execução da tarefa pode ser observado quando particionamos os dados em partições com tamanho de $88\ MB$ ou mais, gerando um tempo de execução pelo menos 534% mais alto que o uso de partições que não extrapolam o limite da EPC.

Discussão

Mais uma vez, podemos notar que o limite de memória disponível na EPC é um dos fatores-chave no desenvolvimento de aplicações SGX. Mais especificamente, mostramos com estes experimentos que é essencial que grandes volumes de dados sejam particionados antes de serem processados em aplicações SGX. Dados os resultados obtidos, recomendamos que grandes volumes de dados sejam particionados em pedaços tão pequenos quanto possível, e evitar a todo custo processar pedaços de dados que excedam o tamanho de $80\ MB$.

4.3 Particionamento de aplicações

Conforme exposto na Seção 3.4, aplicações SGX são divididas entre parte segura, *i.e.*, enclave que protege dados contra modificação ou acesso não autorizados, e parte insegura, *i.e.*, código em volta do enclave, que serve para processar dados não-sensíveis, e para estabelecer uma comunicação entre o enclave e outras aplicações. Desta forma, nesta seção, buscamos analisar qual a melhor forma de particionar aplicações SGX, de modo que sua performance e sua segurança não sejam comprometidas.

Experimentos

A característica que avaliamos nesta seção envolve variações na divisão de uma aplicação, com o objetivo de torná-la segura, através do uso de SGX. Deste modo, coletamos tanto o tempo de execução necessário para completar a tarefa de computar a soma de um número variado de elementos, como fazemos considerações sobre o TCB resultante.

Neste experimento, consideramos uma aplicação simples, que tem como funções (i) gerar e (ii) computar a soma de uma quantidade N de números inteiros. Para realizar esta tarefa, consideramos duas implementações. A primeira delas, exibida no Código Fonte 4.3, realiza ambas as funções dentro do enclave, gerando um maior TCB, mas mantendo todo o processamento dentro do enclave, e é descrita da seguinte forma:

1. O enclave inicializa a variável *sum* com o valor 0 (linha 10);
2. O enclave gera um novo número (linhas 3-7 e linha 13);
3. O enclave soma o novo número à variável *sum* (linha 14);
4. O enclave repete N vezes os passos 2 e 3, até obter o resultado final (linhas 12-15).

A segunda implementação, exibida no Código Fonte 4.4, por sua vez, realiza a função de somar os elementos dentro do enclave, enquanto a função de gerar o próximo elemento é realizada pela parte insegura da aplicação, gerando um TCB reduzido, mas executando um grande número interações entre as partes segura e insegura através de OCalls. Esta implementação é descrita da seguinte forma:

1. O enclave inicializa a variável *sum* com o valor 0 (linha 10);
2. O enclave solicita um novo número através da chamada da OCall *ocall_generate_number* (linha 14);
3. A parte insegura gera um novo número (linhas 3-6);
4. O enclave soma o novo número à variável *sum* (linha 15);
5. Repetimos N vezes os passos 2, 3 e 4, até obter o resultado final (linhas 13-16).

Código Fonte 4.3: Pseudocódigo do experimento de particionamento de aplicações, com todas as tarefas executadas no enclave.

```
1   // No enclave da aplicação
2
3   int generate_number(){
4      int res;
5      sgx_read_rand(&res, sizeof(int));
6      return res;
7   }
8
9   long enclave_calculate_sum(int N){
10     long sum = 0;
11     int i, next_number;
12     for (i=0; i<N; ++i){
13        next_number = generate_number();
14        sum += next_number;
15     }
16     return sum;
17  }
```

Código Fonte 4.4: Pseudocódigo do experimento de particionamento de aplicações, com as tarefas separadas entre o enclave e a parte insegura.

```
1   // No parte insegura da aplicação
2
3   int ocall_generate_number(){
4      int res = rand() % 100;
5      return res;
6   }
7
8   // No enclave da aplicação
9
10  long enclave_calculate_sum(int N){
11     long sum = 0;
12     int i, next_number;
13     for (i=0; i<N; ++i){
14        ocall_generate_number(&next_number);
15        sum += next_number;
```

```
16          }
17          return sum;
18      }
```

Neste experimento, o número de valores gerados, N, em ambas implementações foi variado entre 100 e 100000000, em uma escala logarítmica.

Resultados obtidos

A Figura 4.3 mostra a comparação do tempo de execução de ambas implementações descritas, considerando cada uma das confirgurações determinadas. Neste experimento, podemos notar que a implementação que gera os valores fora do enclave e o soma ao resultado parcial é, em todos os casos, cerca de 20 vezes mais lenta que a implementação que realiza ambas as funções dentro do enclave.

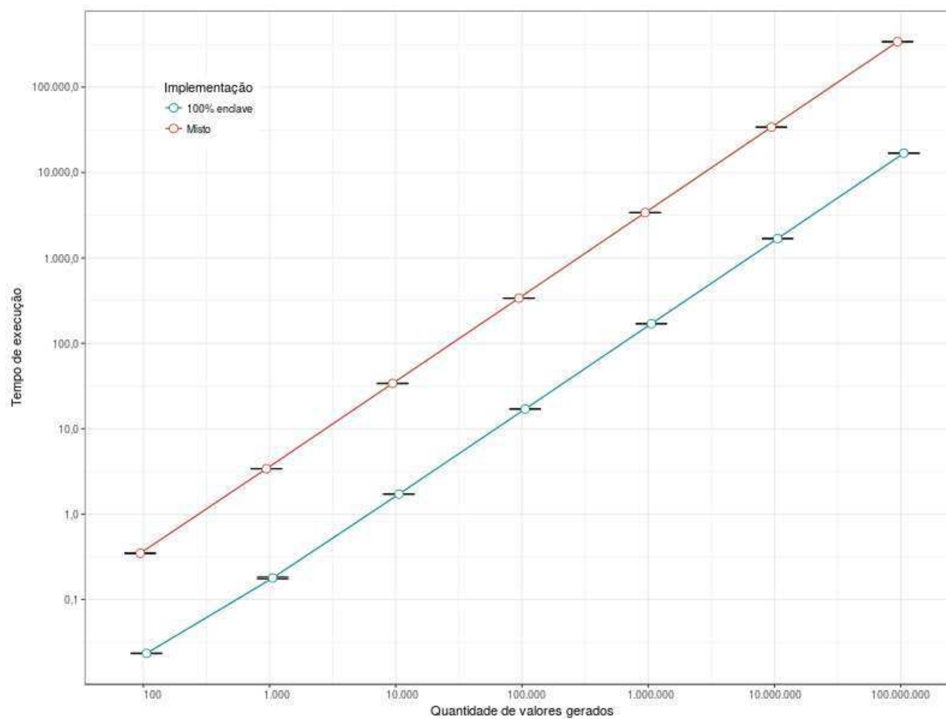


Figura 4.3: Tempos médios para gerar e calcular a soma de N elementos, variando N entre, 100 e 100000000, com um intervalo de 95% de confiança.

Discussão

A recomendação feita pela Intel, em relação ao particionamento de aplicações, é de que o enclave contenha apenas o código das funções que lidam diretamente com dados sensíveis, com o objetivo de manter o TCB o menor possível. Considerando os resultados obtidos neste experimento, porém, podemos notar que nem sempre esta divisão será viável, devido à grande sobrecarga gerada no tempo de execução da aplicação. Esta sobrecarga acontece pelo fato de que, a cada execução de uma OCall, o processador precisa realizar uma troca de contexto do modo enclave para o modo normal, e de volta para o modo enclave ao terminar a execução da OCall. Desta forma, recomendamos que funções da parte insegura que têm muita interação com funções do enclave da aplicação também sejam implementadas e executadas dentro do enclave.

Capítulo 5

DynSGX

5.1 Introdução

Considerando as limitações do SGX apresentadas na Seção 3.9 e as observações feitas na Seção 4.1, foi desenvolvida uma ferramenta chamada DynSGX [SBB17]. DynSGX tem como objetivos: (i) permitir o carregamento e remoção de código dinamicamente em enclaves SGX, (ii) garantir a segurança e privacidade de código, (iii) permitir uma melhor gerência de memória por parte do desenvolvedor e usuário de aplicações SGX, e (iv) facilitar o uso de Intel SGX por desenvolvedores sem conhecimento sobre programação de aplicações SGX.

DynSGX permite que aplicações seguras sejam inicializadas com enclaves pequenos, e carregar e remover funções no enclave em tempo de execução, à medida que elas são necessárias. Isso permite que desenvolvedores e usuários possam gerenciar de forma melhor o consumo de memória. Além disso, DynSGX faz uso do processo de atestação remota para estabelecer um canal de comunicação confiável para ser usado no carregamento e remoção de funções do seu enclave, possibilitando a privacidade do código das funções.

5.2 Componentes

O DynSGX é construído baseado em apenas dois componentes: (i) o `DynSGX enclave` e (ii) o `DynSGX Client`. O `DynSGX enclave` funciona como um TEE, responsável pela segurança e privacidade de código e dados de desenvolvedores e usuários. O `DynSGX Client`, por sua vez, funciona como uma interface entre o `DynSGX enclave` e seus de-

senvolvedore de aplicações e usuários. Outra função do `DynSGX Client` é automatizar o processo de conversão de uma aplicação qualquer para uma aplicação segura. O funcionamento de ambos os componentes é melhor definido nas seções que seguem.

5.3 TCB do DynSGX

No `DynSGX enclave` o TCB é limitado ao Intel SGX SDK e PSW, como qualquer aplicação SGX, acrescentando-se apenas um conjunto de seis funções compatíveis com SGX, resultando em um enclave com tamanho inicial de apenas $1.4\ MB$. Tais funções são usadas para (i) realizar o processo de atestação remota, (ii) carregar funções no enclave, (iii) executar funções carregadas dinamicamente, e (iv) remover funções do enclave.

Para o processo de atestação remota, apenas a função *enclave_ra_init* precisa ser inserida no enclave. Esta função internamente executa a função *sgx_ra_init* do SDK, que inicia o processo de atestação. Para carregar funções no enclave, duas funções são necessárias: *enclave_get_fas*, responsável por prover uma lista de funções que já estão disponíveis no enclave, e *enclave_register_function*, responsável por carregar novas funções no enclave. A função *enclave_execute_function* é utilizada para executar funções carregadas no enclave. Por fim, as funções *enclave_unregister_function* e *enclave_clear_functions* podem ser utilizadas para remover funções do enclave.

O `DynSGX Client` não é considerado como parte do TCB do DynSGX. Isso porque ele é um *script* escrito em Python, contendo apenas cerca de 300 linhas de código, que é executado no ambiente do próprio usuário, podendo ser facilmente auditado pelo mesmo.

5.4 Modelo de programação

Assim como várias outras ferramentas baseadas em programação na nuvem, DynSGX segue o modelo cliente-servidor, onde o enclave do DynSGX executa no lado do servidor, e os desenvolvedores e usuários interagem com ele a partir do lado do cliente.

DynSGX não requer que desenvolvedores tenham conhecimento sobre como desenvolver aplicações SGX. Ao invés disso, desenvolvedores podem escrever suas funções como se estivessem desenvolvendo um programa qualquer na linguagem de programação C. Após

escrever suas funções, uma das ferramentas providas pelo DynSGX, `bytes_extractor` compila o arquivo *.c* que contém o código das funções, e posteriormente recupera os *bytes* que compõem essas funções. Os *bytes* recuperados pela ferramenta são então enviados de forma segura para o enclave do DynSGX, onde são armazenados para uma posterior execução. É importante mencionar que os arquivos *.c* são compilados com a opção -fPIC, de modo a gerar um código binário independente de posição.

Para exemplificar esse processo, vamos supor que um usuário deseje computar de forma segura e privada a soma de dois números inteiros. Para isso, o desenvolvedor da aplicação escreve uma função chamada *sum_function*, como a descrita no Código Fonte 5.1.

Código Fonte 5.1: Exemplo de função C para somar dois números inteiros.

```
1 int sum_function(int a, int b) {
2     return a + b;
3 }
```

Após compilar o arquivo *.c* que contém essa função, a ferramenta `bytes_extractor` recupera os *bytes* do código *Assembler* em arquitetura *x86-64*, como o exibido no Código Fonte 5.2, resultando na seguinte *hexstring*: `\x55\x48\x89\xe5\x89\x7d\xfc\x89\x75\xf8\x8b\x55\xfc\x8b\x45 \xf8\x01\xd0\x5d\xc3`. Esta *hexstring* pode, então, ser carregada no enclave do DynSGX, onde ela será registrada e armazenada na *heap*, que é uma área de memória protegida pelo SGX. Quando um usuário precisa executar a sua função, a *hexstring* será internamente convertida pra um ponteiro de função antes de ser executada como uma função qualquer.

Código Fonte 5.2: Código *Assembler* correspondente à função *sum_function*.

```
1     push %rbp
2     mov %rsp,%rbp
3     mov %edi,-0x4(%rbp)
4     mov %esi,-0x8(%rbp)
5     mov -0x4(%rbp),%edx
6     mov -0x8(%rbp),%eax
7     add %edx,%eax
8     pop %rbp
9     retq
```

Quando uma função não é mais necessária para o usuário, ele pode excluí-la do enclave do DynSGX, de forma a liberar o precioso espaço de memória que está sendo ocupado com o código. Internamente, o enclave DynSGX executa uma chamada da função *free*, que efetivamente libera o espaço de memória para a *heap*. Atualmente este processo ainda é mecânico, demandando que o usuário requeira a remoção das funções, mas formas de automatizar este processo podem ser desenvolvidas, conforme discutidas mais à frente.

5.5 Ciclo de vida de aplicações

Os enclaves do DynSGX são instanciados com um número mínimo de funções, que são essenciais para o seu funcionamento. Depois que o enclave é instanciado, desenvolvedores ou usuários podem estabelecer uma conexão para prover suas funções dinamicamente ao `DynSGX enclave`. Após enviar suas funções, usuários podem executá-las dentro do enclave, e até mesmo removê-las posteriormente. Os passos necessários para completar esse processo estão ilustrados na Figura 5.1, e descritos da seguinte forma:

1. O `DynSGX Client` se comunica com o `DynSGX enclave` e realiza o processo de atestação remota para verificar a integridade e identidade do enclave, e estabelecer um canal de comunicação seguro entre ambos;

2. O `DynSGX Client` compila as funções providas pelo desenvolvedor em um arquivo *.c*, recupera os *bytes* do código *Assembler* gerado usando a ferramenta `bytes_extractor`, e envia a *hexstring* obtida para o enclave através do canal de comunicação seguro. Como resposta, o `DynSGX Client` recebe um identificador das funções carregadas no enclave;

3. Usando o canal de comunicação seguro, o `DynSGX Client` solicita que o `DynSGX enclave` execute alguma das funções que foram carregadas dinamicamente no enclave, enviando junto à requisição quaisquer parâmetros fornecidos pelo usuário. Como resposta, o resultado da execução é enviado de volta ao usuário através do `DynSGX Client`;

4. Por fim, o usuário usa o `DynSGX Client` para requisitar que o `DynSGX enclave` remova funções que não serão mais utilizadas, de forma a liberar espaço na memória.

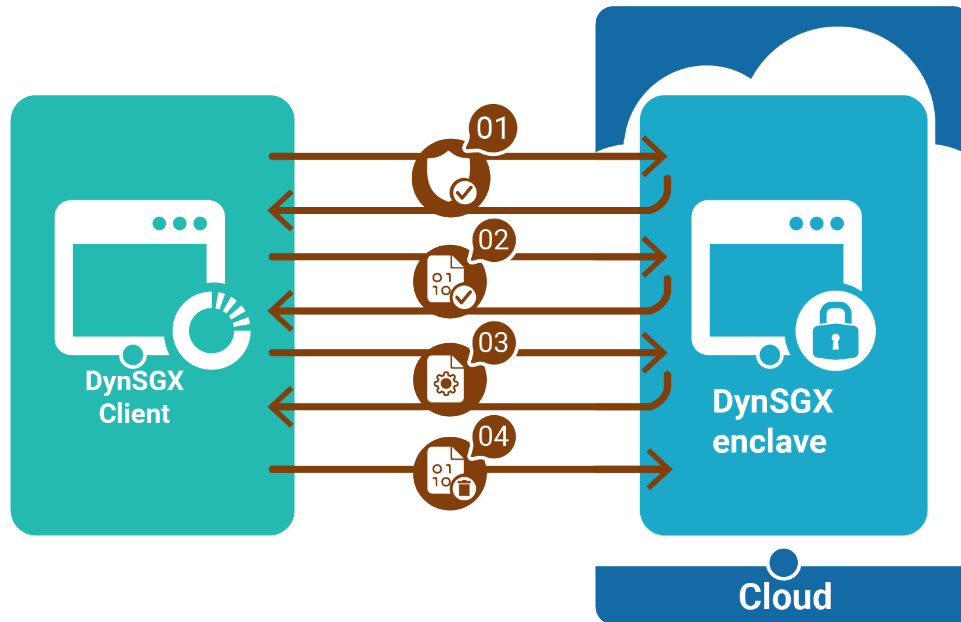


Figura 5.1: Ciclo de Vida de aplicações DynSGX.

5.6 Ligação distribuída de código

Para uma função autocontida, *i.e.*, não utiliza elementos externos, apenas compilar e enviar os *bytes* da função original é o suficiente. Entretanto, se a função usar elementos externos, um mecanismo distribuído é necessário para mapear estes elementos para os seus respectivos endereços no lado do servidor. Exemplos de elementos externos incluem:

- Funções de bibliotecas, como as da `libc`;
- Outras funções previamente carregadas no enclave pelo desenvolvedor;
- Variáveis globais.

DynSGX possui um mecanismo que recupera endereços e tipos de retorno de todas as funções disponíveis no enclave e os envia para o `DynSGX Client` na forma de *JSON*, como o apresentado no Código Fonte 5.3, depois de estabelecido um canal de comunicação seguro pelo processo de atestação remota.

Código Fonte 5.3: Exemplo de *JSON* contendo o mapeamento de elementos externos para seus respectivos endereços, que podem ser usados por outras funções.

```
{
  "snprintf": "(*(int(*)(0x7f1e438176f0)))",
  "vsnprintf": "(*(int(*)(0x7f1e4381d770)))",
  "strcmp": "(*(int(*)(0x7f1e438179a0)))",
  ...
}
```

Para uma função como a apresentada no Código Fonte 5.4, o texto *strcmp*, na linha 3, é substituído por `(*(int(*)(0x7f1e438179a0)))`. Esse mecanismo funciona por ser o mesmo que converter e executar um ponteiro de função. O compilador não sabe qual função estará carregada neste endereço, mas em tempo de execução, a função *strcmp* estará neste endereço dentro do enclave.

Código Fonte 5.4: Exemplo de função que usa um elemento externo (a função *strcmp*).

```
int check_password(char* input) {
  char password[] = "topsecret123";
  return !strcmp(input, password);
}
```

5.7 Requisitos

Para executar o `DynSGX enclave` e carregar funções nele em tempo de execução, três requisitos devem ser satisfeitos:

- ***Hardware* com suporte a SGX**: A tecnologia SGX precisa estar disponível e habilitada no *BIOS*. Tal tipo de *hardware* está disponível em processadores de prateleira desde o fim de 2015.
- ***Driver* SGX**: O *driver* SGX driver precisa estar instalado, de modo a permitir que o SO e outro conjunto de *software* seja capaz de acessar o dispositivo.
- **SGX PSW**: O SGX PSW[4] é usado para instanciar enclaves SGX, e também para gerar estruturas de dados necessárias durante o processo de atestação remota. DynSGX requer que uma pequena modificação seja feita ao PSW. Esta modificação diz respeito

à permissão de tornar a *heap* executável, e pode ser feita aplicando-se um *patch*[8] ao código do SGX PSW.

5.8 Vulnerabilidades

Além da vulnerabilidade a ataques de canal lateral inerente do próprio SGX, DynSGX adiciona uma superfície de ataque: as funções enviadas pelo desenvolvedor. Para prover suas funcionalidades, DynSGX desabilita duas proteções: (i) canários de pilha nas funções dos desenvolvedores, e (ii) *heap* não executável.

Canários de pilha são usados para detectar um *buffer overflow* antes que a execução de código malicioso aconteça. Este método funciona inserindo um número inteiro na memória, cujo valor é escolhido aleatoriamente na inicialização de um programa, logo antes do ponteiro de retorno da pilha. A maioria dos ataques de *buffer overflow* sobrescrevem alguns endereços de memória com o objetivo de sobrescrever o ponteiro de retorno, ganhando assim controle sobre o processo. Com esse ataque, porém, o valor do canário também é sobrescrito. Esse valor é verificado para garantir que não foi modificado antes de uma rotina utilizar o ponteiro de retorno da pilha. Se o valor for modificado, uma função de erro é executada. Esta função, e sua execução são adicionadas pelo compilador. Um programador não é capaz de acessá-la nem do lado do enclave para obter o seu endereço, nem do lado do `DynSGX Client` para substituir a sua execução. Portanto, o mecanismo descrito na Seção 5.6 não deve funcionar para canários de pilha.

Heap não executável é uma medida de segurança que ajuda a prevenir que certos ataques logrem sucesso, especialmente aqueles que injetam e executam código na *heap*. Com DynSGX, funções de desenvolvedores são carregadas dinamicamente para o espaço da *heap*. Portanto, foi necessário desabilitar esta proteção usando a opção `<HeapExecutable>1</HeapExecutable>` no arquivo `Enclave.config.xml`.

Considerando a limitação do mecanismo ASLR devido ao pequeno espaço de memória, e desabilitar as proteções de canários de pilha e *heap* não executável, é extremamente importante um alto empenho no desenvolvimento de código seguro. O Código Fonte 5.5 apresenta

[8] `https://patch-diff.githubusercontent.com/raw/01org/linux-sgx/pull/63.patch`

um exemplo de código vulnerável que poderia ser carregado dinamicamente no *DynSGX enclave*.

Código Fonte 5.5: Exemplo de função vulnerável a *buffer overflow*.

```
1   void check_password(char *input) {
2       char buffer[16];
3       char password[] = "topsecret123";
4       strncpy(buffer, input, strlen(input));
5       if (!strcmp(buffer, password))
6           access();
7   }
```

Esta função é vulnerável a *buffer overflow*. Se um atacante prover uma entrada maior que 16 *bytes*, valores que estivessem armazenados em memória poderiam ser sobrescritos, como o valor do ponteiro da base da pilha, e o endereço de retorno. Considerando uma função maliciosa armazenada no endereço *0x7ffff580160d*, um atacante poderia desviar o fluxo de execução para a função maliciosa enviando a seguinte carga útil como parâmetro da função: *AAAAAAAAAAAAAAAAAAAAAAAA\x0d \x16\x80\xf5\xff\x7f\x00\x00*.

Outro exemplo de vulnerabilidade é a formatação não verificada de *strings*. Uma função de formatação é um tipo especial de função em *C* que recebe um número variável de argumentos, onde um deles é chamado de formato de *string*. Sem as devidas precauções neste tipo de função, um atacante se torna capaz de ler e escrever em qualquer lugar da memória. O Código Fonte 5.6 exemplifica um código vulnerável a este tipo de ataque.

Código Fonte 5.6: Exemplo de função vulnerável a formatação não verificada de *strings*.

```
1  void check_password(char *input) {
2      char password[] = "topsecret123";
3      if (!strcmp(input, password)) {
4          access();
5      } else {
6          char error[30];
7          snprintf(error, 30, input);
8          strncat(error, " is incorrect!", 14);
9          log_msg(error);
10     }
11 }
```

Em uma situação normal, esta função teria o mesmo comportamento da função apresentada no Código Fonte 5.4, acrescentando apenas a funcionalidade de registrar uma mensagem de erro. A vulnerabilidade de formatação não verificada de *strings* se encontra na linha 7, e um atacante poderia enviar a seguinte carga útil como parâmetro da função: `%10$p %11$p`. Desta forma, a função irá registrar a mensagem: `0x6572636573706f74 0x33323174 is incorrect!`. Os numeros apresentados em hexadecimal são a representação no formato *little endian* da senha.

Dadas as limitações nas proteções de segurança, é muito importante que a escrita de código seja feita de forma cuidadosa, e regularmente fazer revisões de código. O uso de ferramentas que examinam o código fonte e alertam sobre possíveis vulnerabilidades também pode ser útil. Flawfinder[9], Cppcheck[10] e CheckConfigMX [BGM+16] são exemplos de ferramentas de análise estática de código que podem ser integradas ao DynSGX para rapidamente encontrar e eliminar potenciais falhas de segurança de funções antes de enviá-las para o enclave.

5.9 Avaliação

Para avaliar a performance de aplicações que usam DynSGX, uma série de experimentos foram realizados, comparando implementações DynSGX com implementações em C puro e usando SGX. Nesta seção são descritos os experimentos realizados, e é apresentada uma discussão baseada nos resultados obtidos.

5.9.1 Preparação dos experimentos

Os experimentos foram conduzido em uma máquina com sistema operacional Ubuntu Linux 16.04, um processador Intel i7-6700, e $8\ GB$ de memória principal disponível. O `DynSGX enclave` foi desenvolvido na linguagem de programação *C++*, e o `DynSGX Client` foi desenvolvido usando a linguagem de programação *Python* 2.7.

Nos experimentos, foi computada a latência, que representa o tempo total decorrido desde o início de uma requisição feita pelo cliente para executar uma função do lado servidor, até

[9] `https://www.dwheeler.com/flawfinder/`
[10] `http://cppcheck.sourceforge.net/`

o mesmo cliente obter o resultado do processamento feito. No cálculo da latência também é incluído o tempo para realizar o processo de atestação remota quando ela é necessária, e de cifrar e decifrar os dados em ambos os lados para aumentar os níveis de segurança e privacidade dos dados enviados entre cliente e servidor.

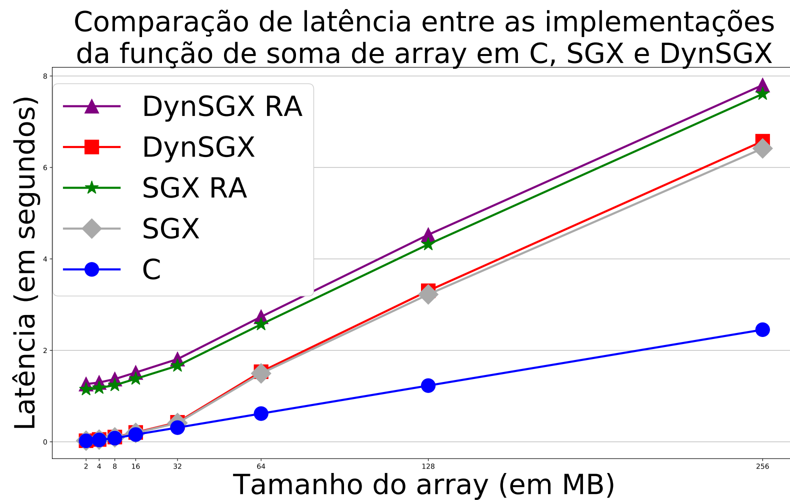
A latência foi calculada baseada em duas funções com comportamentos distintos. A primeira é a função *sum_array*, que itera sobre um *array* de números inteiros, computando a soma de todos eles. A segunda é a função *recursive_fibonacci*, que recursivamente calcula o n-ésimo termo da sequência de Fibonacci. Estas duas funções foram escolhidas para ilustrar uma aplicação com gargalo no consumo de memória, que é uma limitação conhecida do SGX, e uma aplicação com gargalo no consumo de CPU.

Para avaliar a performance da ferramenta proposta, ambas as funções foram implementadas de três maneiras distintas. A primeira implementação, feita em C puro, não provê nenhuma funcionalidade de segurança e privacidade. A segunda usa o modelo de programação SGX, provendo segurança e privacidade dos dados envolvidos. A terceira usa o modelo de programação DynSGX, que permite uma gerência da memória consumida pelas funções de um enclave, e adiciona a funcionalidade de privacidade do código sendo executado no enclave. Para as implementações do modelo SGX e DynSGX, foram consirados ambos os casos quando o processo de atestação remota precisaria ser realizado para estabelecer o canal de comunicação seguro, e quando este canal já havia sido estabelecido anteriormente.

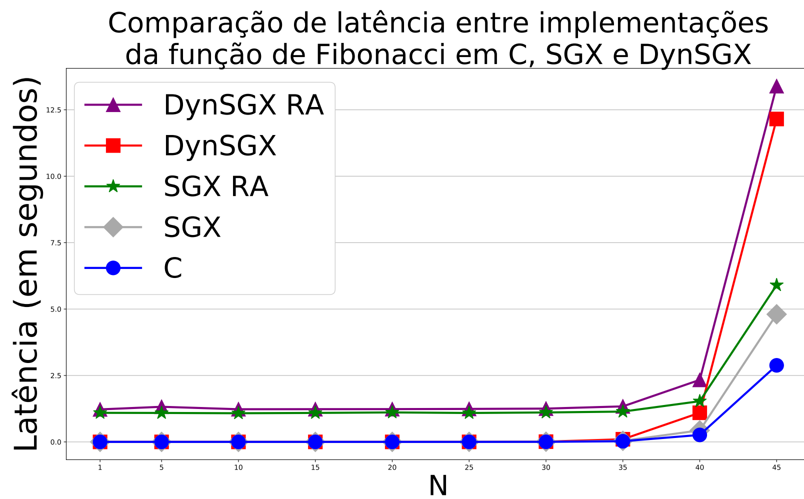
5.9.2 Experimentos

Os experimentos foram constituídos de duas partes. A primeira parte buscava comparar a latência para computar a função *sum_array* para *arrays* com tamanhos de 2, 4, 8, 16, 32, 64, 128, e 256 MB. A Figura 5.2a mostra a mediana dos valores de latência obtidos nesta parte do experimento. Este experimento serve para nos mostrar o comportamento do uso de DynSGX com funções iterativas, especialmente quando há um consumo de memória que extrapola a memória disponível na EPC.

A segunda parte do experimento visa comparar a latência para computar a função *recursive_fibonacci* assumindo os valores de 1, 5, 10, 15, 20, 25, 30, 35, 40, e 45 para a variável n. A Figura 5.2b mostra a mediana dos valores de latência obtidos nesta parte do experimento. Este experimento é importante para mostrar o comportamento do uso de



(a) Latência para a função *sum_array*



(b) Latência para a função *recursive_fibonacci*

Figura 5.2: Comparação de latência entre as implementações em C puro, Intel SGX, e DynSGX

DynSGX com funções recursivas.

Em ambas as partes do experimento, para cada tamanho de *array* e para cada valor da variável n, 30 repetições foram executadas. Os resultados obtidos nestes experimentos têm distribuição enviesada. Por este motivo, escolhemos a mediana como medida de tendência central utilizada na análise feita.

5.9.3 Discussão

Como pode ser visto na Figura 5.2, ambas as implementações que usam SGX e DynSGX sempre geram uma sobrecarga considerável se comparadas a implementação em C puro, que não se preocupa com segurança e privacidade de dados do usuário. Esta sobrecarga acontece por dois motivos principais: (i) o tempo necessário para realizar o processo de atestação remota, e (ii) a necessidade de cifrar e decifrar dados constantemente.

A partir dos resultados obtidos nos experimentos, pode-se observar também que para processar funções iterativas, DynSGX apresenta apenas uma pequena sobrecarga, de aproximadamente $2{,}5\%$, comparando-se com o uso de SGX. Este comportamento pode ser observado em ambos os casos quando é necessário realizar o processo de atestação remota, e quando o canal de comunicação seguro já foi estabelecido. Neste cenário, nós consideramos os benefícios de usar DynSGX significantes, por adicionar privacidade do código executado no enclave, e a possibilidade de gerenciar a memória ocupada por código.

Entretanto, é importante observar que o DynSGX pode não ter uma performance muito boa no caso de funções recursivas. A Figura 5.2b mostra este cenário, onde o tempo de execução da função *recursive_fibonacci* cresce exponencialmente à medida que o número de chamadas recursivas cresce. Neste cenário, DynSGX tem uma performance muito inferior à performance do SGX. Este comportamento acontece porque com o DynSGX o código das funções reside na *heap*, que é um segmento de dados, e compete com outras áreas de dados, como os quadros de dados das chamadas recursivas, por espaço na *cache* do processador. Processadores modernos têm *caches* distintas para código e dados. Deste modo, a implementação SGX tem melhor performance neste cenário por manter o código da função na *cache* de instruções. Este comportamento não é limitado ao DynSGX. Uma implementação C pura ou SGX de um algoritmo recursivo que armazene código na *heap* também obterá alguma sobrecarga.

Tabela 5.1: Comparação de performance entre as implementações em C puro, SGX e DynSGX

	Segurança		**Privacidade**		**Performance**	
Impl.	**dados**	**código**	**dados**	**código**	**iterativo**	**recursivo**
C puro					Alta	Alta
SGX	✓	✓	✓		Média	Média
DynSGX	✓	✓	✓	✓	Média	Baixa

Em relação aos aspectos de segurança mencionados na Seção 5.8, é importante evidenciar que desabilitar os canários de pilha e habilitar a execução da *heap* não necessariamente implicam riscos para o dono do *hardware* executando o `DynSGX enclave`, uma vez que em ambientes de computação na nuvem é possível configurar outras camadas de segurança. Por exemplo, o `DynSGX enclave` poderia ser executado em um ambiente isolado como uma máquina virtual.

Por fim, na Tabela 5.1 é apresentada uma comparação das vantagens e desvantagens de cada uma das três implementações consideradas nos experimentos.

5.10 Considerações

As limitações existentes na tecnologia Intel SGX levou à construção da ferramenta DynSGX, apresentada neste capítulo. A ferramenta proposta facilita o desenvolvimento de aplicações que fazem uso de SGX, uma vez que não requer que o desenvolvedor adquira conhecimento sobre as APIs da tecnologia. A ferramenta também possibilita a gerência de memória ocupada pelo código de uma aplicação, o que não era possível no modelo de programação convencional do SGX.

Considerando a avaliação conduzida sobre a ferramenta, foi mostrado que a ferramenta possibilita obter privacidade também do código da aplicação a ser executada dentro do enclave, gerando apenas uma pequena sobrecarga em relação a aplicações SGX comuns em alguns casos.

Por fim, é necessário ressaltar que DynSGX não deve ser aplicado em todos os cenários. Um dos cenários que se mostrou impróprio para o uso de DynSGX é o das funções recursivas que possuem uma quantidade exponencial de chamadas recursivas. Neste caso, o desenvolvedor deve analisar se o custo decorrente do uso do DynSGX é justificado pelos ganhos de privacidade do código e gerenciamento de memória consumida por código.

Capítulo 6

Conclusão

Neste capítulo, apresentamos, na Seção 6.1, um sumário do que foi proposto e realizado nesta pesquisa. Além disso, apresentamos algumas considerações sobre as análises feitas, bem como sobre a ferramenta proposta no Capítulo 5. Por fim, na Seção 6.2, apresentamos as limitações deste trabalho, e possíveis trabalhos futuros.

6.1 Sumário

Nesta pesquisa de mestrado discutimos os principais desafios no desenvolvimento de aplicações com propriedades de segurança e privacidade de dados, usando a tecnologia Intel SGX. Para tanto, no Capítulo 4, analisamos como várias decisões na implementação e no uso de aplicações SGX influenciam no seu desempenho geral, e nas garantias de segurança realacionadas ao tamanho do TCB da aplicação. As características analisadas incluem como devemos gerenciar a memória de enclaves SGX, como deve ser o particionamento entre parte segura e parte insegura das aplicações SGX, e como deve ser o particionamento dos dados a serem processados dentro de enclaves SGX.

As análises apresentadas servem para auxiliar desenvolvedores na tomada de decisões sobre boas práticas a serem aplicadas na arquitetura de aplicações seguras, diferenciando-se no particionamento de código e dados, e gerência de memória de tais aplicações em comparação a aplicações inseguras, onde esses aspectos não têm tanto impacto.

Este trabalho foi motivado pela crescente popularidade do uso da tecnologia Intel SGX para desenvolver aplicações que proveem segurança e privacidade de dados, observada na

literatura, e pela falta da definição de padrões no desenvolvimento de tais aplicações, a não ser as recomendações feitas pela própria Intel. Em experimentos realizados em nosso ambiente, constatamos que o mau planejamento de uma aplicação SGX pode acarretar uma grande perda de desempenho da mesma, chegando a um custo de execução até 534% mais alto, comparado a aplicações bem planejadas.

Considerando as análises feitas, propomos uma ferramenta, DynSGX, que serve como uma alternativa ao modelo de programação do Intel SGX, facilitando o uso desta tecnologia, *i.e.*, não requer que o desenvolvedor conheça o SGX SDK, e o gerenciamento de memória de suas aplicações SGX. Nós avaliamos esta ferramenta, por meio de experimentos, comparando o seu desempenho com o desempenho de uma aplicação desenvolvida usando o modelo de programação tradicional do Intel SGX. Os resultados obtidos nos experimentos indicam que o DynSGX pode ser utilizado de forma eficiente para o processamento de funções iterativas, gerando uma sobrecarga de apenas $2{,}5\%$ em relação ao uso de SGX puro, porém o seu uso para processamento de funções recursivas pode ter uma performance muito inferior à solução que usa SGX puro.

6.2 Limitações e trabalhos futuros

Apesar de mostrar algumas das características de aplicações SGX que afetam o seu desempenho, a lista aqui apresentada não é exaustiva, e não considera como duas ou mais características em conjunto podem impactar no desempenho das aplicações. Desta forma, mais análises precisam ser feitas para determinar outras boas práticas a serem aplicadas no desenvolvimento de aplicações SGX.

A segunda limitação deste trabalho está na forma como foi avaliada a segurança das aplicações, onde apenas foi considerado o tamanho do TCB.

Uma outra limitação deste trabalho diz respeito à ferramenta proposta, DynSGX. O DynSGX foi desenvolvido com o intuito de funcionar como uma plataforma de função como um serviço (FaaS, do inglês *Function as a Service*). A sua implementação atual, porém, requer que ambos o desenvolvedor e o usuário das funções carregadas no `DynSGX enclave` sejam a mesma pessoa, de modo a ganhar confiança em relação a segurança e privacidade dos dados processados. Além disso, como citado anteriormente, a tarefa de remover funções

do enclave do DynSGX é demasiadamente mecanizada. A automatização desta tarefa deve ser tratada em uma futura versão do DynSGX.

Como possíveis trabalhos futuros, além de solucionar as limitações destacadas, propomos a evolução do DynSGX, de forma a transformá-lo em uma plataforma completa de aplicações *serverless*, possibilitando o desenvolvimento e uso de aplicações por pessoas diferentes, com garantias de segurança e privacidade de dados e código carregados na plataforma. Essa abordagem permite maiores facilidade e segurança no desenvolvimento de aplicações SGX, bem como um melhor gerenciamento automático de memória utilizada pelos enclaves do DynSGX.

Bibliografia

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[AMD] AMD. Amd memory encryption. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf. Último acesso em 10/01/2018.

[ARM] ARM. Arm truszone. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. Último acesso em: 10/01/2018.

[ATG+16] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L Stillwell, et al. Scone: Secure linux containers with intel sgx. In *12th USENIX Symp. Operating Systems Design and Implementation*, 2016.

[BGM+16] L. Braz, R. Gheyi, M. Mongiovi, M. Ribeiro, F. Medeiros, and L. Teixeira. A change-centric approach to compile configurable systems with #ifdefs. In *ACM GPCE'16*, pages 109–119, 2016.

[BPH15] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.

[Fet16] Christof Fetzer. Building critical applications using microservices. *IEEE Security & Privacy*, 14(6):86–89, 2016.

[Fou] The Linux Foundation. The xen project. https://www.xenproject.org/. Último acesso em 10/01/2018.

[GESM17] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *EUROSEC*, pages 2–1, 2017.

[Glo] GlobalPlatform. Trusted execution environment (tee) guide. https://www.globalplatform.org/mediaguidetee.asp. Último acesso em 10/01/2018.

[GLS+17] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security Symposium*, 2017.

[GLX+17] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, pages 488–501, New York, NY, USA, 2017. ACM.

[HCP17] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association, Santa Clara, CA*, pages 299–312, 2017.

[HLP+13] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. *HASP@ ISCA*, 11, 2013.

[HPE] HPE. 2016 cost of cyber crime study & the risk of business innovation. https://www.ponemon.org/local/upload/file/2016Último acesso em 10/01/2018.

[HSH+09] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[ida] Ida disassembler and debugger. https://www.hex-rays.com/products/ida/index.shtml. Último acesso em 10/01/2018.

[Int] Intel R Intel. Software guard extensions sdk for linux* os, revision 1.9. https://download.01.org/intel-sgx/linux-2.0/docs/Intel_SGX_SDK_Developer_Reference_Linux_2.0_Open_Source.pdf. Último acesso em 10/01/2018.

[KOA+17] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 205–221. ACM, 2017.

[Kor09] Kostya Kortchinsky. Cloudburst: A vmware guest to host escape story. *Black Hat USA*, page 19, 2009.

[KPA16] Klaudia Krawiecka, Andrew Paverd, and N Asokan. Protecting password databases using trusted hardware. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*, page 9. ACM, 2016.

[kvm] Kernel virtual machine. https://www.linux-kvm.org/. Último acesso em 10/01/2018.

[LABW92] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems (TOCS)*, 10(4):265–310, 1992.

[Log] Trusted Logic. Trusted foundations by trusted logic mobility. Último acesso em 10/01/2018.

[LPM+17] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. Glamdring: Automatic application partitioning for intel sgx. In *2017 USENIX Annual Technical Conference (USENIX ATC 17), Santa Clara, CA*, 2017.

[LSG+16] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. *arXiv preprint arXiv:1611.06952*, 2016.

[MAB+13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ ISCA*, page 10, 2013.

[McC04] Steve McConnell. *Code complete*. Pearson Education, 2004.

[Mer78] Ralph C Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21(4):294–299, 1978.

[MIE17] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *Cryptographic Hardware and Embedded Systems*, CHES 2017, pages 69–90. Springer, 2017.

[MPP+08] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 315–328, 2008.

[MVOV96] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[OMT09] OMTP. Advanced trusted environment: omtp tr1 v1.1. http://www.omtp.org/OMTP_Advanced_Trusted_Environment_OMTP_TR1_v1_1.pdf, 2009. Último acesso em 10/01/2018.

[Pub01] NIST FIPS Pub. 197: Advanced encryption standard (aes). *Federal information processing standards publication*, 197(441):0311, 2001.

[qem] Qemu. https://www.qemu.org/. Último acesso em 10/01/2018.

[QZW+85] Lili Qiu, Yin Zhang, Feng Wang, Mi Kyung, and Han Ratul Mahajan. Trusted computer system evaluation criteria. In *National Computer Security Center*. Citeseer, 1985.

[Rus81] John M Rushby. *Design and verification of secure systems*, volume 15. ACM, 1981.

[SBB17] R. Silva, P. Barbosa, and A. Brito. Dynsgx: A privacy preserving toolset for dinamically loading functions into intel(r) sgx enclaves. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 314–321, Dec 2017.

[SCNS15] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing your faults from telling your secrets: Defenses against pigeonhole attacks. *arXiv preprint arXiv:1506.04832*, 2015.

[SLK+17] Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[SLKP17] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[SMVB17] Leandro Ventura Silva, Rodolfo Marinho, Jose Luis Vivas, and Andrey Brito. Security and privacy preserving data aggregation in cloud computing. In *Proceedings of the Symposium on Applied Computing*, SAC '17, pages 1732–1738, New York, NY, USA, 2017. ACM.

[Swa17] Yogesh Swami. Intel sgx remote attestation is not sufficient. 2017.

[SWG+17] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2017.

[TPV17] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical

library os for unmodified applications on sgx. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, page 8, 2017.

[tru] Trustonic secured platforms. https://www.trustonic.com/solutions/trustonic-secured-platforms-tsp. Último acesso em 10/01/2018.

[vmw] vmware. https://www.vmware.com/. Último acesso em 10/01/2018.

Apêndice A

Exemplo de aplicação SGX

Como discutido na Seção 3.4, aplicações SGX são divididas entre parte segura, que executa dentro de enclaves SGX, e parte insegura, usada para instanciar o enclave da aplicação, executar ações que não necessitem de garantias de segurança, e intermediar qualquer tipo decomunicação entre um enclave e o mundo externo a ele, inclusive a execução de *system calls*. Além disso, precisamos definir uma interface de comunicação entre as partes segura e insegura (arquivo EDL).

Para exemplificar o desenvolvimento de aplicações SGX, utilizamos a seguir a aplicação utilizada nos experimentos da Seção 4.2.

A.1 Desenvolvimento do arquivo EDL

A primeira coisa a se pensar no desenvolvimento de aplicações SGX é em como particionar a aplicação entre as partes segura e insegura, e, mais precisamente, é necessário pensar em como ambas as partes se comunicam entre si. Para isso, precisamos escrever um arquivo EDL, contendo a declaração das ECalls (declaradas na seção *trusted* do arquivo EDL), e das OCalls (declaradas na seção *untrusted* do arquivo EDL).

No caso da aplicação em questão, apenas a função *enclave_sum* cruza entre as partes segura e insegura. O conteúdo do arquivo EDL está descrito no Código Fonte A.1. No código apresentado, o parâmetro "*in*", entre colchetes, indica que o *array* é um ponteiro de entrada, ou seja, será copiado da parte insegura para o enclave antes da execução da função *enclave_sum*. O parâmetro "*size=array_size*"indica que o tamanho do *array*, em *bytes*, é

dado pela variável *array_size*.

Código Fonte A.1: Arquivo EDL contendo a declaração das funções que cruzam a fronteira entre as partes segura e insegura da aplicação SGX.

```
1  // Arquivo enclave.edl
2  enclave{
3     trusted{
4        public long enclave_sum(
5           [in, size=array_size] int *array,
6                             size_t array_size
7        );
8     };
9     untrusted{}; // Nenhuma função a declarar aqui.
10 };
```

Após a criação do arquivo EDL, utlizamos a ferramenta *sgx_edger8er* para compilar o arquivo EDL, e gerar as interfaces de comunicação entre as partes segura e insegura propriamente ditas. Executamos, portanto, os comandos `sgx_edger8r --trusted enclave.edl --search-path ${SGXSDK_PATH}/include` e `sgx_edger8r --untrusted enclave.edl --search-path ${SGXSDK_PATH}/include`, para gerar os arquivos de interface a serem utilizados, respectivamente, pelas partes segura e insegura da aplicação.

Mais informações sobre a sintaxe dos arquivos EDL pode ser encontrada no documento de referência para desenvolvedores [Int].

A.2 Desenvolvimento do enclave

O código do enclave contém a implementação de todas as ECalls declaradas no arquivo EDL, adicionando-se o código de quaisquer funções internas necessárias. No caso da aplicação em questão, precisamos escrever apenas a implementação da função *enclave_sum*. Tal implementação pode ser encontrada no Código Fonte A.2.

Código Fonte A.2: Arquivo contendo a implementação das funções a serem executadas totalmente dentro do enclave da aplicação SGX.

```
1  // Arquivo enclave.cpp
2  #include "stdlib.h"
3  #include "enclave_t.h" // Gerado pela ferramenta sgx_edger8er
4
5  long enclave_sum( int *array, size_t array_size) {
6     long sum = 0;
7     int i;
8     for (i=0; i<array_size/sizeof(int); ++i){
9        sum += array[i];
10    }
11    return sum;
12 }
```

O enclave deve, então, ser compilado com o compilador *g++*, de modo a gerar uma biblioteca dinâmica (*.so* no caso do Linux, ou *.dll* no caso do Windows). Por fim, o enclave deve ser assinado usando a ferramenta *sgx_signer*, usando o comando `sgx_signer sign -key priv_key -enclave enclave.so -out enclave.signed.so -config enclave.config.xml`, onde `priv_key` é a chave privada do desenvolvedor, `enclave.so` é a biblioteca dinâmica do enclave produzido pelo *sgx_edger8er*, `enclave.signed.so` é o enclave resultante, assinado, e `enclave.config.xml` é o arquivo de configuração das características do enclave SGX, conforme exibido no Código Fonte A.3.

Entre as configurações do arquivo *enclave.config.xml* que podemos, destacar, estão o tamanho máximo da *heap* e tamanho máximo da pilha do enclave. Ambas configurações têm impacto no direto no tempo necessário para a instanciação de um enclave, pois o SGX aloca todo esse espaço de memória durante a instanciação do enclave. No caso caso específico da configuração do enclave utilizada nos experimentos da Seção 4.2, a instanciação de um enclave levava cerca de 30 segundos para completar. Enclaves com tamanhos máximos de *heap* e pilha menores que os usados nesta configuração levam menos tempo para serem instanciados.

Código Fonte A.3: Arquivo de configuração do enclave SGX.

```
1    // Arquivo enclave.config.xml
2    <EnclaveConfiguration>
3    <ProdID>0</ProdID>
4    <ISVSVN>0</ISVSVN>
5    <StackMaxSize>0x8000</StackMaxSize>
6    <HeapMaxSize>0x100000000</HeapMaxSize>
7    <TCSNum>10</TCSNum>
8    <TCSPolicy>1</TCSPolicy>
9    <DisableDebug>0</DisableDebug>
10   <MiscSelect>0</MiscSelect>
11   <MiscMask>0xFFFFFFFF</MiscMask>
12 </EnclaveConfiguration>
```

Mais informações sobre o arquivo de configuração do enclave podem ser encontradas no documento de referência para desenvolvedores [Int].

A.3 Desenvolvimento da parte insegura da aplicação

A parte insegura de uma aplicação SGX compreende todo o código que lida com dados não sensíveis, adicionando-se o código necessário para instanciar e se comunicar com o enclave, e a implementação de quaisquer OCalls que tenham sido declaradas no arquivo EDL. No caso da aplicação em questão, nenhuma OCall precisa ser implementada. O código necessário para a parte insegura desta aplicação, é exibido no Código Fonte A.4.

O fluxo da aplicação pode ser descrito nos seguintes passos:

1. A aplicação instancia o enclave SGX (linhas 13-19);

2. A aplicação gera um *array* com 1GB, povoado por números inteiros aleatórios (linhas 20-22);

3. A aplicação invoca a função do enclave que calcula a soma dos valores contidos em uma partição do *array* (linha 29);

4. A aplicação soma o resultado obtido do enclave ao total armazenado na variável *sum* (linha 30);

5. A aplicação repete os passos 3 e 4 até que todas as partições do *array* tenham sido processadas (linhas 27-31);

6. A aplicação exibe o resultado da soma de todos os elementos do *array* (linha 32);

7. A aplicação destrói o enclave (linha 33).

Código Fonte A.4: Arquivo contendo a implementação da parte insegura da aplicação SGX.

```
1    // Arquivo app.cpp
2    #include "stdlib.h"
3    #include "enclave_u.h" // Gerado pela ferramenta sgx_edger8er
4    #include "sgx_urts.h" // Necessário para instanciar enclaves
5
6    #define ENCLAVE_NAME "enclave.signed.so"
7    #define MB_SIZE 1024*1024
8    #define ARRAY_SIZE 4096
9    #define CHUNK_SIZE 1
10
11   int main(){
12
13      sgx_enclave_id_t eid;
14      sgx_status_t ret = SGX_SUCCESS;
15      sgx_launch_token_t launch_token;
16      int updated = 0;
17      ret = sgx_create_enclave( // Instancia um enclave SGX
18            ENCLAVE_NAME, SGX_DEBUG_FLAG, &launch_token,
19            &updated, &eid, NULL );
20      int total_num_of_elements = ARRAY_SIZE * MB_SIZE / sizeof(int);
21      int *array = (int *) calloc(total_num_of_elements,sizeof(int));
22      generate_random_numbers(array, total_num_of_elements); // Preenche
            array com números inteiros aleatórios
23      int chunk_num_of_elements = CHUNK_SIZE * MB_SIZE / sizeof(int);
24      long sum = 0, partial_sum;
25      int i, idx;
26
27      for(i=0; i<total_num_of_elements/chunk_num_of_elements; ++i){
28         idx = i*chunk_num_of_elements;
```

```
29            enclave_sum(eid, &partial_sum, &array[idx], CHUNK_SIZE);
30            sum += partial_sum;
31        }
32        printf("A soma dos elementos do array é: %ld.\n", sum);
33        sgx_destroy_enclave(eid);
34        return 0;
35    }
```