Rodrigo Martins Cardoso

Licenciado

Blockchain-based Storage with SGX Clients for Mobile Games

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Doutor Nuno Manuel Ribeiro Preguiça, Professor Associado com Agregação, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Júri

Presidente: Doutor Pedro Manuel Corrêa Calvente Barahona, Professor Catedrático, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Vogais: Doutor Rui Miguel Soares Silva, Professor Adjunto, Instituto Politécnico de Beja

Doutor Nuno Manuel Ribeiro Preguiça, Professor Associado com Agregação, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Julho, 2020

Blockchain-based Storage with SGX Clients for Mobile Games

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Agradecimentos

Primeiramente, quero enaltecer a indispensável ajuda do meu orientador Nuno Preguiça que foi fundamental durante todo o processo de elaboração deste trabalho. Toda a sua paciência, rigor e conhecimento foram imprescindíveis para a conclusão desta dissertação. Quero também deixar o meu agradecimento ao Bernardo Ferreira por ter ajudado na construção de determinados detalhes do meu trabalho. Não posso também deixar de agradecer o empréstimo do *hardware* necessário para elaborar este trabalho e a oportunidade que me foi dada para trabalhar no projeto em que este está inserido. O meu fortíssimo obrigado à Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa como organização pela experiência inigualável que me proporcionou durante estes anos de persistência e trabalho árduo. O meu profundo obrigado ao meu colega e mais que tudo amigo Abel Silva por ter sido a verdadeira demonstração de que realmente existem amigos que na faculdade ficam para a vida. Estes anos não seriam o mesmo se não fossem os meus colegas João Alpoim, Paulo Dias, Tito Ferreira, Diogo Oliveira e André Pereira que numa base diária me apoiaram, me mostraram o que era o verdadeiro espírito de entre-ajuda e tornaram todo este tempo memorável. Todos eles ajudaram a escrever uma parte da história da minha vida que ficará para sempre na minha memória pelos bons e maus momentos, mas que num todo servem de aprendizagem para o resto da minha existência.

Do lado mais pessoal, estou grato pela confiança que o meu pai, José Cardoso, e avó, Venceslau Cardoso, depositaram em mim para seguir o meu gosto especial pelo mundo da informática que tenho desde criança. Estou também bastante grato pela preocupação da minha mãe Paula Cardoso e avó Fernanda Martins. O meu obrigado mais sentido vai para a mulher e namorada, Inês Silva, que me acompanhou durante todo este trajeto, que esteve sempre presente nos piores e melhores momentos, que festejou comigo as minhas conquistas ao longo do curso e que nunca me deixou desistir de o acabar. Todo o seu suporte incansável e o seu crer de que eu era capaz foram peça chave para o término deste ciclo. Finalmente quero agradecer-me por ter aguentado as noites mais difíceis, pela perseverança em todas as avaliações e pela dedicação e entrega de corpo e alma ao curso que sempre quis. O meu comprido obrigado a todos os que tiveram presentes nesta minha temporada e que este curso me faça valer para o resto da minha vida.

Resumo

Nos últimos anos, muitos jogos multijogador para telemóveis com um grande número de utilizadores tornaram-se populares e um excelente exemplo disso é o *Pokémon GO*. Estes jogos de escala planetária são verdadeiros desafios de engenharia que obrigam a uma arquitetura descentralizada para dar suporte ao tráfego de milhares de jogadores.

Os recentes e contínuos avanços tecnológicos dos dispositivos móveis permitem construir *smartphones* com um poder de computação, armazenamento e processamento gráfico cada vez maiores e melhores, o que atrai um exorbitante número de jogadores/utilizadores a nível global. Ao combinar todas as funcionalidades disponíveis neste tipo de aparelhos (ecrã táctil, sensor de movimento, sistema de localização preciso, entre outros) com a ligação ubíqua à rede são possíveis jogos *online* ligados ao movimento do dispositivo, jogos multijogador, jogos baseados em localização e lojas de aplicações. Com todas estas capacidades unidas, os jogos de telemóveis inteligentes fornecem uma experiência distinta aos utilizadores e abrem um horizonte de possibilidades não exploradas.

Com a expansão do número base de utilizadores móveis, proliferação de *smartphones* e *tablets* e o aumento do interesse em jogos móveis multijogador, uma crescente demanda por serviços, tecnologias e arquiteturas para dar suporte a jogos *wireless*, surge inevitavelmente para dar resposta à dificuldade de suportar de forma centralizada tais jogos de massiva escala.

Como uma solução inovadora para esta procura, foi desenhado um modelo que permite a clientes móveis comunicarem com fortes garantias, para suportar carteiras/moedas virtuais, através de transações diretas com outros com confiança e sem a necessidade de uma entidade central reguladora. Para este fim, o sistema pensado é composto por uma rede *peer-to-peer* onde os clientes utilizam novas extensões presentes nos novos processadores Intel a partir da sexta geração: as Intel *Software Guard Extensions*. Ao confiar no cliente, que executa de forma isolada código em *hardware* seguro, é esperado um sistema transacional mais escalável e mais rápido na produção de transações do que os tradicionais atuais.

Palavras-chave: jogos de telemóvel, moeda virtual, Intel *Software Guard Extensions*, rede *peer-to-peer*, dispositivos móveis

Abstract

In the last years, many mobile multiplayer games with a large number of users have become popular and a prime example of this is the Pokémon GO. These planetary-scale games are truly engineering challenges requiring a decentralized architecture to support the traffic of thousands of players.

The recent and continuous technological advances of mobile devices allow us to build smartphones with better computing, storage and graphics power which attracts an exorbitant number of players globally. By combining all the functionalities available in this type of device (touch screen, motion sensor, precise location system, etc.) with the ubiquitous connection to the network online games connected to device movement, multiplayer games, location based games and application stores are possible. With all these capabilities together, smartphone games deliver a distinct user experience and open up a horizon of unexplored possibilities.

With the expansion of the number of mobile users, the proliferation of smartphones and tablets, and the growing interest in mobile multiplayer games, a growing demand for services, technologies and architectures to support wireless gaming inevitably arises to address the difficulty of supporting such mass-scale games in a centralized fashion.

As an innovative solution to this demand, a model was designed that allows mobile clients to communicate with strong guarantees, to support virtual wallets/currencies, through direct transactions with others with confidence and without the need for a central regulatory authority. To this end, the system designed consists of a peer-to-peer network where clients use new extensions present in the new Intel processors from the sixth generation: the Intel Software Guard Extensions. By relying on the client, which executes code in isolation on secure hardware, a more scalable and faster in the production of transactions transactional system is expected than the current traditional ones.

Keywords: mobile games, virtual coin, Intel Software Guard Extensions, mobile devices, peer-to-peer network

Índice

Índice xiii

Lista de Figuras xv

Lista de Tabelas xvii

Listagens xix

Siglas xxi

Lista de Algoritmos xxiii

1 Introdução 1
- 1.1 Motivação . . . 1
- 1.2 Contexto . . . 2
- 1.3 Resumo . . . 3
- 1.4 Contribuições . . . 3
- 1.5 Organização . . . 3

2 Estado da Arte 5
- 2.1 O problema dos generais Bizantinos . . . 5
  - 2.1.1 Sistema síncrono . . . 6
  - 2.1.2 Sistema assíncrono . . . 8
- 2.2 Soluções *Blockchain* . . . 8
  - 2.2.1 *Bitcoin* . . . 9
  - 2.2.2 MultiChain . . . 10
  - 2.2.3 *Ethereum* . . . 12
  - 2.2.4 Contratos Inteligentes . . . 14
- 2.3 Sistemas de replicação de máquinas de estado tolerantes a falhas Bizantinas 17
  - 2.3.1 O Primeiro Sistema Prático . . . 17
  - 2.3.2 Hybrid Quorum . . . 19
  - 2.3.3 Swirlds Hashgraph . . . 21
- 2.4 Intel Software Guard Extensions . . . 22
  - 2.4.1 Natureza Interna das Intel SGX . . . 23

2.4.2 Parte Externa das Intel SGX 33

3 Desenho do Sistema **41**
3.1 Arquitetura 41
3.2 Protocolo de autenticação 44
3.3 Protocolo de transferência 49

4 Implementação do Sistema **51**
4.1 Ferramentas 51
4.2 Cliente 53
4.2.1 Habilitar as SGX e carregar o enclave 53
4.2.2 Primeira etapa da atestação 55
4.2.3 Segunda etapa da atestação 58
4.2.4 Terceira etapa da atestação 61
4.2.5 Enviar uma transação 61
4.3 Enclave 65
4.3.1 Modelo de Programação 65
4.3.2 Desenvolvimento 69
4.4 Provedor de Serviços 75

5 Avaliação **83**
5.1 Abordagem 83
5.2 Testes 84
5.3 Resultados 85

6 Conclusão **89**
6.1 Trabalho futuro 90

Bibliografia **91**

Anexos **95**

I Atestação e Selagem baseadas no CPU **95**
I.1 Modelo de Segurança SGX 95
I.2 Instruções Intel SGX 96
I.3 Medições 96
I.3.1 MRENCLAVE - Identidade do Enclave 96
I.3.2 MRSIGNER - Identidade de Selagem 97
I.4 Atestação 97
I.5 Atestação Intra-Plataforma 99
I.6 Atestação Inter-Plataforma 100
I.6.1 *Intel Enhanced Privacy ID* 101
I.6.2 *Quoting Enclave* 101

I.7 Selagem 101
I.8 Políticas de Selagem Intel SGX 102
I.8.1 Selagem para a Identidade de Enclave 102
I.8.2 Selagem para a Identidade de Selagem 102
I.9 Remover Segredos da Plataforma 103

II Provisionamento EPID e Serviços de Atestação **105**
II.1 Atualização da TCB 105
II.2 Requerimentos da Atestação Intel 105
II.3 Vinculação da Chave TCB 106
II.4 Derivação da TCB do *Hardware* 106
II.5 Chaves Raiz 107
II.6 Derivação da TCB do *Software* 108
II.7 Propriedades do EPID 108
II.8 Modos de Assinatura EPID 109
II.9 Modos de Revogação EPID 110
II.10 Serviços das Infraestruturas SGX 111
II.11 Chaves e Dados de Revogação 112
II.12 Cenários de Provisionamento 113
II.13 Fluxo de Provisionamento 114
II.14 Serviço de Atestação da Intel 115
II.15 Verificação do *Quote* 115

Lista de Figuras

2.1 Dois cenários em que apenas um de três processos é defeituoso (retirados de [14]) . . . 7
2.2 Ilustração detalhada da Máquina Virtual *Ethereum* (fonte indicada na figura) 13
2.3 Composição de uma aplicação SGX (retirada de [1]) . . . 23
2.4 Disposição da memória de uma aplicação SGX (retirada de [1]) . . . 24
2.5 Verificação de páginas estendida (retirada de [1]) . . . 26
2.6 Gestão de páginas (retirada de [1]) . . . 27
2.7 Conteúdo da EPC (retirada de [1]) . . . 28
2.8 Verificação da SIGSTRUCT de um enclave (retirada de [1]) . . . 29
2.9 Criação de um enclave (retirada de [1]) . . . 29
2.10 Ciclo de vida de um enclave (retirada de [1]) . . . 30
2.11 Tratamento de exceções no enclave (retirada de [1]) . . . 31
2.12 Atestação Local por SGX (retirada de [1]) . . . 33
2.13 Atestação Remota por SGX (retirada de [1]) . . . 34
2.14 Derivação de chaves nas Intel SGX (retirada de [2]) . . . 36
2.15 Recuperação de chaves nas Intel SGX (Key Recovery Transformation) (retirada de [2]) . . . 37
2.16 Vista geral das chaves das Intel SGX (retirada de [2]) . . . 38

3.1 As três componentes do sistema (clientes + enclaves, SP e IAS) . . . 42
3.2 Protocolo de Autenticação em alto nível retirada de [27]) . . . 43
3.3 Diagrama do fluxo de atestação . . . 46

4.1 Aplicação Intel SGX . . . 67
4.2 API do IAS para obtenção da lista de assinaturas revogadas (retirado de [26]) 78
4.3 API do IAS para obtenção do relatório de verificação de atestação (retirado de [26]) . . . 81

I.1 Criação de Asserção de Atestação (retirada de [3]) . . . 99

II.1 Derivação de TCB básica (retirada de [28])) . . . 106
II.2 Ciclo PRF TCB (retirada de [28])) . . . 107
II.3 Linha temporal das Chaves TCB (retirada de [28])) . . . 107

II.4 Hierarquia de Chaves SGX (retirada de [28]) 108
II.5 Serviços das Infraestruturas SGX (retirada de [28]) 112
II.6 Elementos da Infraestrutura EPID (retirada de [28]) 113
II.7 Fluxo de Provisionamento (retirada de [28]) 114

Lista de Tabelas

2.1 Regras de transação das diferentes plataformas (retirada de [21]) 15
2.2 Determinismo nas diferentes plataformas (retirada de [21]) 16
2.3 Prevenção de conflitos nas diferentes plataformas (retirada de [21]) 17
2.4 Instruções SGX do Supervisor 25
2.5 Instruções SGX do Utilizador 25

3.1 Dados das primeiras duas mensagens 47
3.2 Dados da terceira mensagem 47
3.3 Dados da quarta mensagem 48
3.4 Dados da última mensagem 49
3.5 Dados de uma transação 49

4.1 Estrutura da msg0 57
4.2 Estrutura da msg1 57
4.3 Estrutura da msg3 59
4.4 Estrutura do *quote* 60
4.5 Estrutura de uma transação 63
4.6 Detalhes da API de receção transações 64
4.7 Comportamento dos atributos *in*, *out* e *user_check* 72
4.8 Estrutura da msg2 77

I.1 SIGSTRUCT 98
I.2 Acesso às chaves de relatório e de *seal* 100

II.1 Algumas propriedades do *Software* usadas na Derivação de Chaves (retirada de [28]) 109
II.2 Chaves SGX (blocos verdes na Figura II.4) (retirada de [28]) 110

Listagens

4.1 *Template* de um ficheiro EDL 70
4.2 Ficheiro *remote_attest_flow.edl* 71
4.3 Ficheiro *transactions.edl* 75

Siglas

API	Application Programming Interface
BIOS	Basic Input/Output System
CPU	Central Processing unit
DAA	Direct Anonymous Attestation
DCAP	Data Center Attestation Primitives
DHKE	Diffie-Hellman Key Exchange
EPC	Enclave Page Cache
EPID	Enhanced Privacy ID
IAS	Intel Attestation Service
ISK	Intel Signing Key
JSON	JavaScript Object Notation
MAC	Message Authentication Code
PPID	Platform Provisioning ID
PRF	Pseudorandom Function Family
PrivRL	EPID Private Key Revocation List
QE	Quoting Enclave
REST	Representational State Transfer
SGX	Software Guard Extensions

SIGSTRUCT	Enclave Signature Structure
SP	Service Provider
SPID	Service Provider ID
SVN	Security Version Number
TCB	Trust Computing Base
TLB	Translation Lookaside Buffer
TPM	Trusted Platform Module

Lista de Algoritmos

1 Consulta e ativação do estado das Intel SGX 54
2 Carregamento e inicialização do enclave 56
3 Criação da primeira mensagem 57
4 Serialização e envio da primeira mensagem 58
5 Criação da terceira mensagem 61
6 Serialização da msg3 para envio e processamento da msg4 62
7 Receção de uma transação 64
8 Processamento da msg1 e construção para envio da msg2 79
9 Processamento da msg2 e construção para envio da msg4 82

Capítulo 1

Introdução

Atualmente os dispositivos móveis desempenham um papel preponderante no mundo tecnológico, pois permitem serviços inovadores sob a forma de aplicações móveis. Estes programas que executam nos telemóveis são uma tendência em constante evolução pelo conforto, acessibilidade e ubiquidade que proporcionam. Aplicações móveis mais recentes, tais como as de redes sociais, depositam extrema importância na interação entre os seus utilizadores. Um exemplo notável destas plataformas são jogos multijogador que exigem interação em tempo real de centenas de milhões de utilizadores.

Arquitetar aplicações desta natureza é um verdadeiro desafio e uma prova disso são, por exemplo, as dificuldades sentidas ao "tentar dar vida" ao jogo *Pokémon GO* [38] . Os principais problemas de desenhar estas aplicações surgem quando toda a interação do utilizador é mediada por servidores. Esta centralização provoca uma sobrecarga sobre os servidores, torna-os um ponto único de falha e causa um *bottleneck* no nível de escalabilidade das aplicações.

1.1 Motivação

A era dos *smartphones* está a provocar uma disrupção na abordagem para prestar serviços *online*. Estes dispositivos são equipados com *hardware* e *software* em constante evolução que lhes permite oferecer uma experiência inovadora e conveniente ao utilizador [16] quando comparada às interfaces *desktop*. Para além disso, a comodidade e facilidade são fatores chave que tornam os dispositivos móveis, nos dias de hoje, a plataforma de eleição para prestar serviços *online* sob a forma de aplicações móveis.

É notória a prevalência destes aparelhos nos dias que correm, basta observar o comportamento das pessoas no intervalo de um filme numa sala de cinema. A imposição das redes sociais e a forma como instantaneamente podemos conversar com alguém despoleta

um fator imprescindível: interação. Aplicações atuais de jogos multijogador *online* necessitam de gerir interações em tempo real de uma quantidade massiva de jogadores. Mas os jogos móveis não estão sozinhos, os próprios festivais de música já adotam aplicações móveis com utilizadores a interatuar em tempo real nas suas estratégias de negócio.

A tendência é que as interações sejam cada vez mais exigentes devido à elevada afluência do número de pessoas. Assim, a capacidade de controlar interações entre um número quase hiperbólico de utilizadores é um derradeiro desafio de engenharia, como já demonstrou o jogo *Pokémon GO* [6], devido às falhas no seu serviço com uma abordagem centralizada.

As arquiteturas dos sistemas atuais em que todas as interações entre os utilizadores são mediadas por servidores centrais exibem vários problemas. Primeiro, os servidores limitam severamente a escalabilidade do sistema. Como uma atualização de um utilizador precisa de ser propagada para um subconjunto de todos os utilizadores, a computação que precisa de ser executada cresce mais do que linearmente com o número de utilizadores. Em segundo lugar, quando os servidores se tornam inacessíveis devido a uma falha ou a uma falha de rede o cliente é desconectado. Por fim, a interação com utilizadores próximos exibe uma longa latência desnecessária, uma vez que todas as mensagens passam pelos servidores mesmo quando dependem de infraestruturas de nuvem replicadas geograficamente. A superação destes problemas pode ter custos, exigir soluções complexas de engenharia e em alguns casos pode ser até impossível abordar estas limitações com arquiteturas centralizadas.

1.2 Contexto

O tema desta dissertação está inserido no projeto SAMOA (*Secure and Scalable Platform for Massive-scale Mobile Applications*) e tem como objetivo fundar uma moeda virtual para jogos de telemóvel de massiva escala. Considerando o elevadíssimo número de utilizadores deste tipo de jogos, pretende-se endereçar os problemas dos sistemas baseados em *cloud* [6] com a utilização de uma arquitetura descentralizada baseada em *blockchain*, onde os dispositivos móveis comunicam diretamente entre si para suportar uma criptomoeda.

Com os jogadores a interagir diretamente sem aceder à *cloud*, a comunicação é mais eficiente e rápida, a latência é reduzida entre os utilizadores próximos e não existe uma dependência com a *cloud* quando acontecem falhas de comunicação [6]. Todavia, os algoritmos baseados em *blockchain* não permitem uma latência baixa [19], porque os protocolos atuais obrigam a uma coordenação entre os vários nós da rede para assegurar que os utilizadores não realizam operações inválidas o que implica alta latência. Em virtude deste problema, pretende-se negar qualquer operação não especificada por parte dos clientes para simplificar os protocolos que impedem que tal aconteça.

Posto isto, são necessários mecanismos descentralizados capazes de prevenir o acesso não autorizado aos procedimentos de geração de novas transações entre os jogadores. Se tais não fossem utilizados, os jogadores poderiam adulterar as transferências de dinheiro

e gastar a mesma moeda múltiplas vezes (ataque de *double-spending*). Para satisfazer esta necessidade, é explorada a utilização de novo *hardware* especializado introduzido em novos processadores da Intel, as Intel *Software Guard Extensions* [33], que consegue proteger dados e código de computador residentes em zonas de memória reservadas cujos acessos são inteiramente controlados pelo processador.

1.3 Resumo

De forma sumária, neste trabalho foi produzido um molde para sistemas de transações diretas entre clientes numa rede *peer-to-peer* cuja responsabilidade é de as fabricar através do seu *hardware* seguro. A solução obtida, para além de recorrer às Intel SGX no lado do cliente, necessita também de um servidor de autenticação da Intel (IAS) associado a estas novas extensões de *hardware*. É mediante este servidor que é possível indicar se um processador de um cliente é legítimo e não revogado, porém é apenas suficiente para verificar a entidade da máquina do cliente. No sentido de verificar a entidade do código que executa no processador é necessário outro servidor, o provedor de serviços (SP), que está em contacto com o IAS para unir os dois tipos de autenticação cruciais: do código e do *hardware*.

1.4 Contribuições

As contribuições principais são as seguintes:

- **Produção de uma moeda virtual para jogos de telemóvel de massiva escala:** Desenvolver uma criptomoeda especializada para jogos em dispositivos móveis onde o número de clientes é extremamente grande. Esta moeda é a representação eletrónica do dinheiro num jogo, a única forma de transacionar dentro dele e que é armazenada numa zona de memória na máquina do cliente completamente gerida pelo respetivo processador.
- **Desenvolvimento de um molde para sistemas transacionais com base em novas técnicas de segurança de novo *hardware*:** Formular um modelo para construir sistemas que produzem transações com base nos processadores Intel dos seus clientes usando as respetivas extensões de segurança para controlo de acessos. Especificar que entidades e mecanismos de autenticação são necessários para debitar a confiança da geração das transações no lado do cliente. Permitir transações diretas entre clientes após assegurar que executam numa máquina fidedigna.

1.5 Organização

Esta dissertação está decomposta em seis capítulos e dois anexos. No capítulo 2 é elaborado um apanhado das pesquisas científicas de temas pertinentes interligados com o

intuito principal do tema a investigar. São expostos neste capítulo o problema dos generais Bizantinos, marcante para muitos dos sistemas discutidos, a aplicabilidade da tecnologia *blockchain*, exemplificada com sistemas reais de destaque, sistemas de replicação de máquinas de estado tolerantes a falhas bizantinas e, por fim, uma tecnologia inovadora em novas arquiteturas dos processadores Intel, as Intel *Software Guard Extensions*.

No capítulo 3 deste trabalho, apresenta-se de forma minuciosa os pilares teóricos do desenho do sistema, ou seja, que arquitetura foi montada e que protocolos foram criados. Depois deste capítulo, em 4, indica-se como foi aplicado este projeto na prática, referindo que utensílios se usaram e explicando cada entidade do sistema. Logo a seguir, no capítulo 5, estudasse o desempenho do que foi produzido. No último capítulo, são efectuadas reflexões que concluem este trabalho para resumir as ideias principais e os benefícios desta nova abordagem.

Os dois anexos incorporados nesta dissertação servem para conhecer os pormenores dos mecanismos do tipo de processador explorado que suportam muitas das suas funcionalidades que foram aproveitadas. Assim, é naturalmente um conteúdo mais teórico e complexo mas que é fundamental para entender o cerne das novas extensões deste tipo de processador.

Capítulo 2

Estado da Arte

No desenrolar deste capítulo é apresentado o resultado do levantamento, e respetiva síntese, da produção científica nas áreas de interesse do tema abordado. Primeiramente, são expostos sistemas predominantes assentes na tecnologia *blockchain*, cada um com diferentes objetivos. Logo de seguida, é mostrado um confronto entre quatro populares plataformas que suportam contratos inteligentes. Após a comparação das estratégias usadas por cada uma, é exibida uma discussão acerca do problema dos generais Bizantinos. Depois de esclarecer este problema, é enunciado o primeiro algoritmo utilizável na prática para replicação de máquinas de estado e, dois outros com o mesmo propósito, também capazes de resistir a falhas Bizantinas. Para fechar o capítulo, é descrito um modelo de *software* para execução isolada, as Intel *Software Guard Extensions*.

2.1 O problema dos generais Bizantinos

Esta secção é uma síntese de conteúdos retirados de [14].

O problema dos generais Bizantinos é uma abstração utilizada para exprimir a capacidade de um sistema de computadores confiável lidar com falhas, de um ou mais dos seus componentes, que os levam a enviar informação conflituosa para diferentes partes do sistema [31]. Informalmente, neste problema existem três ou mais generais a cercar uma cidade, cada um no comando de um exército, que devem concordar em atacar ou recuar. Cada comandante emite ordens aos seus tenentes que devem, da mesma maneira, decidir se atacam ou se recuam. A questão é que um ou mais generais podem ser traidores, como se de uma falha de um componente se tratasse. Posto isto, se um comandante é traidor, propõe atacar a um general e recuar a outro. Caso um tenente seja traidor, diz a um dos seus camaradas que o comandante lhe disse para atacar e a outro que devem recuar.

O problema dos generais Bizantinos difere do problema do consenso, porque um processo distinto (general) fornece um valor que outros (tenentes) devem chegar a acordo, em vez de todos eles individualmente proporem um valor. Uma solução para este problema exige [14]:

- *Terminação*: Eventualmente cada processo correto define a sua variável de decisão.
- *Acordo*: O valor de decisão de todos os processos corretos é o mesmo: se p_i e p_j são correctos e entraram no estado *decidido*, então $d_i = d_j$ $(i, j = 1, 2, ..., N)$.
- *Integridade*: Se o comandante é correto, então todos os processos correctos decidem o valor que o comandante propôs

2.1.1 Sistema síncrono

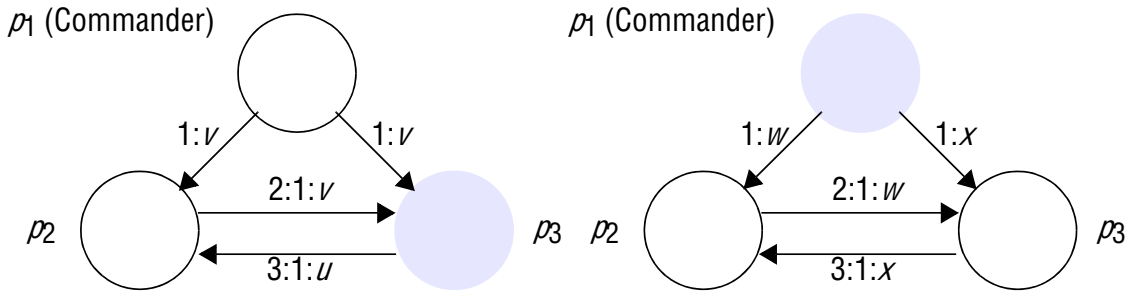
Ao contrário do algoritmo para consenso, assume-se que os processos podem manifestar falhas arbitrárias, isto é, um processo defeituoso pode enviar qualquer mensagem com qualquer valor em qualquer instante e pode omitir o envio de qualquer mensagem. Até f de N processos podem falhar, mas processos corretos conseguem detectar a ausência de uma mensagem através de um *timeout*. No entanto, não são capazes de concluir se o emissor falhou, dado que pode ter estado em silêncio por algum tempo e depois enviou mensagens novamente.

Assume-se que os canais de comunicação entre pares de processos são privados, visto que, se um processo pudesse examinar todas as mensagens que outros processos enviaram, poderia detectar as inconsistências no que um processo com falha envia a diferentes processos. Esta suposição de confiabilidade do canal define que nenhum processo defeituoso pode injetar mensagens no canal de comunicação entre os processos corretos.

Em 1982 Lamport et al. [31] refletiram sobre o caso em que três processos enviam mensagens não assinadas entre eles e demonstraram que não há solução que ateste as condições do problema dos generais Bizantinos se permitido a um processo falhar. Deste modo, generalizaram este resultado para provar que não existe solução se $N \leq 3f$ e prosseguiram com um algoritmo que resolvesse este problema num sistema síncrono se $N \geq 3f + 1$, para mensagens não assinadas.

Para provarem a impossibilidade de três generais suportarem um único traidor, utilizaram dois cenários, onde por simplicidade a decisão possível é apenas uma de dois valores.

Na figura 2.1 estão ilustrados dois cenários em que apenas um de três processos é defeituoso. Os processos com falha estão apresentados a cinzento. São mostradas duas rondas de mensagens com os valores que o comandante envia e os valores que os tenentes posteriormente enviam um ao outro. Os números sob as setas classificam as fontes das mensagens e permitem mostrar as diferentes rondas. O símbolo ":" lê-se "disse" (e.g. "3:1:u" significa "3 diz que 1 disse u"



(a) Cenário 1

(b) Cenário 2

Figura 2.1: Dois cenários em que apenas um de três processos é defeituoso (retirados de [14])

Na subfigura 2.1a, o comandante envia o valor v para cada um dos outros processos, e o processo p_2 reitera esse valor para p_3. Contudo, o processo p_3 envia um valor u para p_2 que é diferente do valor v que na realidade recebeu. Tudo o que p_2 sabe até ao momento é que recebeu dois valore diferentes e não é capaz de dizer qual deles foi enviado pelo comandante. Já na subfigura 2.1b, o comandante defeituoso envia diferentes valores para os tenentes. O processo p_3 ecoa o valor x obtido e p_2 está na mesma situação em que estava quando p_3 estava com falha em 2.1a, em consequência de adquirir dois diferentes valores.

Na hipótese de existir solução, o processo p_2 é forçado a decidir o valor v em 2.1a, pela condição de integridade. Se se aceitar que nenhum algoritmo consegue distinguir entre os dois cenários, p_2 também tem que seleccionar o valor enviado pelo comandante em 2.1b.

Assumindo que p_3 é correto, por simetria, também decide o valor enviado pelo comandante. Porém, isto contradiz a condição de acordo, pois p_3 decide o valor x e p_2 o valor w. Desta forma, conclui-se que não existe solução.

Este argumento, que de facto é correto, é um raciocínio informal que se baseia na intuição de que nada pode ser feito para melhorar o conhecimento de um general correto para além da primeira ronda, onde não consegue dizer que processo falha. Todavia, um acordo bizantino pode ser alcançado com três generais, com um deles com falha, se assinarem digitalmente as suas mensagens.

Este resultado foi generalizado por Pease et al. [31] para provar, por contradição, que continua a não existir solução para este problema se $N \leq 3f$. Um esboço do argumento é o seguinte: assumindo que existe uma solução com $N \leq 3f$, cada um de três processos p_1, p_2 e p_3 usa essa solução para simular o comportamento de n_1, n_2, e n_3 generais, respectivamente, onde $n_1 + n_2 + n_3 = N$ e $n_1, n_2, n_3 \leq \frac{N}{3}$. Assume-se também que um dos processos tem falha. Dos processos p_1, p_2 e p_3, os que são correctos simulam generais correctos, logo, simulam as interacções dos seus próprios generais internamente e enviam mensagem dos seus generais para aqueles simulados por outros processos. Os generais simulados pelo processo defeituoso são defeituosos, por conseguinte, as mensagens que

envia como parte da simulação para os outros dois processos podem ser falsas. Visto que $N \leq 3f$ e $n_1, n_2, n_3 \leq \frac{N}{3}$, no máximo f generais simulados apresentam falha.

Como o algoritmo que os processos executam é assumido como correto, a simulação termina. Os generais correctos simulados (nos dois processos correctos) chegam a acordo e satisfazem a propriedade de integridade. Mas desta maneira, há forma de dois processos correctos de três alcançarem consenso, pois cada um decide o valor escolhido por todos os seus generais simulados, o que contradiz o resultado de impossibilidade para três processos, um deles com defeito.

2.1.2 Sistema assíncrono

Lamport et al. [31] forneceram uma solução para o problema dos generais Bizantinos tendo por base um sistema síncrono com $N \geq 3f + 1$. O algoritmo divulgado adopta rondas para troca de mensagens e os processos podem assumir que um processo falhou caso este exceda o tempo limite estipulado para o envio de uma mensagem durante uma ronda.

Em 1985 Fischer et al. [18] comprovaram que não existe nenhum algoritmo capaz de garantir consenso num sistema assíncrono, mesmo com uma falha por *crash* de um processo. Num sistema assíncrono, os processos podem responder às mensagens em instantes arbitrários, logo, um processo *crashed* é indistinguível de um mais lento.

A partir deste resultado, não há solução garantida num sistema assíncrono para o problema dos generais Bizantinos. Isto não significa que processos nunca atinjam consenso distribuído num sistema assíncrono caso um deles falhe, mas sim que o consenso pode ser alcançado com uma probabilidade mais do que zero.

Para contornar esta impossibilidade pode-se considerar sistema parcialmente síncronos, mais fracos o suficiente que sistema síncronos para serem modelos úteis para sistemas práticos, e suficientemente mais fortes que sistemas assíncronos para que se possa neles chegar a consenso.

2.2 Soluções *Blockchain*

A tecnologia *blockchain* teve um rápido crescimento e o desenvolvimento de aplicações ou sistemas baseados em cadeia de blocos revolucionou o mundo financeiro. Para além de altamente popular em criptomoedas, a aplicabilidade desta tecnologia vai desde contratos inteligentes até ao processamento de transações financeiras em redes proprietárias [34].

Uma criptomoeda é uma moeda digital na qual técnicas de criptografia são usadas para regular a geração de unidades de moeda e verificar a transferência de fundos, operando de forma independente de um banco central. Já um contrato inteligente, usufruindo também do que é a peça chave (criptografia) da tecnologia dos sistemas de cadeia de blocos, são código de computador que reside na *blockchain* e que possibilita fluxos de processamento automáticos e com alto peso computacional [12].

Baseada na tecnologia de registo distribuído, em que cada nó replica e guarda uma cópia idêntica de um registo, atualizando-a de forma independente, emprega uma cadeia de blocos para fornecer um consenso distribuído seguro e válido, que é gerida por redes *peer-to-peer*. Sendo um registo distribuído, pode existir sem uma autoridade centralizada ou servidor a geri-la, e a sua qualidade de dados pode ser mantida por replicação de bases de dados e por confiança computacional. No entanto, a estrutura da *blockchain* torna-a distinta de outros tipos de registo distribuído, uma vez que os dados são agrupados e organizados em blocos, blocos estes que são ligados uns aos outros e protegidos usando criptografia.

Esta cadeia é essencialmente uma lista crescente de registos (blocos) que geralmente não podem sofrer remoções ou alterações. Embora os registos na *blockchain* não sejam inalteráveis, esta tecnologia pode ser considerada segura por design e exemplifica um sistema de computação distribuído com elevada tolerância a falhas bizantinas.

2.2.1 *Bitcoin*

O protocolo *Bitcoin* é um servidor *timestamp* distribuído *peer-to-peer* que gera uma prova computacional, criptográfica, da ordem cronológica de transações. Este surge como uma solução para o problema do *double-spending*, inerente das moedas eletrónicas, e permite que quaisquer duas partes transacionem entre si diretamente sem a necessidade de um terceiro confiável [36].

Neste sistema uma moeda digital é representada como uma cadeia de assinaturas digitais, portanto, utiliza criptografia assimétrica para garantir a autenticidade das transações. Dado que neste tipo de criptografia existe uma relação matemática de uma só direção, mesmo que saibamos a chave pública não podemos obter a chave privada associada, o proprietário de uma moeda assina digitalmente (usando a sua chave privada) um *hash* sobre a transação anterior e a chave pública do próximo proprietário da moeda. Desta forma, quem recebe "o dinheiro" apenas tem que utilizar a chave pública do anterior proprietário para verificar a autenticidade da transação. É gerado um *hash* porque quanto maior o *input* a assinar, pelo ECDSA, algoritmo usado pelo protocolo, mais lento é esse processo e como o *hash* devolve uma *string output* de tamanho fixo, acelera-se a assinatura.

A técnica de assinatura digital aplicada às transações, por si só, não permite verificar se um dos proprietários não efectuou *double-spending* da moeda. Como o objectivo deste sistema é permitir transações diretas entre partes, sem existência de terceiros confiáveis, a única maneira de garantir que não surgem fraudes por *double-spending* é anunciar publicamente todas as transações e fazer com que os participantes estejam de acordo num único histórico da ordem de recepção destas. Para além disto, acresce-se o uso mais uma vez de funções criptográficas de *hash*, mas com um diferente intuito.

Através de um servidor *timestamp* fornece-se prova da existência de uma transação gravando o tempo exato e data em que cada uma é processada. Nesse servidor isto é conseguido efetuando um *hash* sobre um bloco de itens e data que se quer carimbar.

Hashs sobre blocos posteriores utilizam o *hash* resultante do bloco anterior criando uma cadeia onde cada *timestamp* inclui o *timestamp* anterior, com cada *timestamp* adicional a fortalecer os anteriores. Desta maneira garante-se que os dados que serviram de *input* para o *hash* existiram naquele momento, porque é altamente improvável que para dois diferentes *inputs* de dados (bloco + *timestamp*) numa função criptográfica de *hash* se gere o mesmo *output*. Para descentralizar este servidor de *timestamp* há que o distribuir sobre uma rede *peer-to-peer*, usando um sistema de prova-de-trabalho.

O sistema de prova-de-trabalho é a essência deste protocolo onde múltiplos pares na rede competem para validar cada transação consecutiva, validação esta que constitui encontrar um bloco (que inclui o *hash* do bloco anterior, o *timestamp*, o *nonce* incremental, entre outros) que quando submetido a uma função de *hash*, o *hash* resultante começa com um dado nº de zeros. Este processo de validação leva tempo e poder de computação suficientes tal que nenhuma entidade pode ter a certeza de que validará as próximas transações. Para além disso, uma vez que uma pequena alteração no *input* de uma função de *hash* criptográfica resulta em alterações dramáticas no *hash output*, os pares da rede estarão a executar uma tarefa de força bruta até encontrar o bloco cujo *hash* inicia com os zeros estipulados, arbitrariamente testando diferentes *nonces*. Assim, à medida que mais pessoas se ligam à rede para proceder à verificação de transações, mais seguro o sistema se torna, pois para modificar um bloco anterior um atacante teria que refazer a prova-de-trabalho desse bloco e todos os blocos após esse, e, em seguida, alcançar e superar o trabalho dos nós honestos. Finalmente, a decisão da maioria é representada pela cadeia mais longa, que tem o maior esforço de prova-de-trabalho investido nela. Se a maioria da potência do processador for controlada por nós honestos, a cadeia honesta crescerá mais rápido e ultrapassará as cadeias concorrentes.

A rede *peer-to-peer* funciona da seguinte maneira: **1)** Novas transações são transmitidas para todos (não tem que ser 100%) os nós; **2)** Cada nó guarda novas transações num bloco; **3)** Cada nó trabalha em encontrar uma prova-de-trabalho difícil para o seu bloco; **4)** Quando um nó encontra uma prova-de-trabalho, transmite o bloco para todos os nós; **5)** Os nós aceitam o bloco somente se todas as transações nele forem válidas e ainda não tiverem sido gastas; **6)** Os nós expressam a sua aceitação do bloco trabalhando na criação do próximo bloco da cadeia, usando o *hash* do bloco aceite como o *hash* anterior.

Sempre que um par obtém o *hash* vencedor é recompensado com um incentivo, uma quantia de bitcoins para ele próprio. Este incentivo fornece uma razão para suportar a rede e permite que bitcoins sejam distribuídos.

2.2.2 MultiChain

A plataforma *MultiChain* permite criar e desenvolver *blockchains* privadas e tem como principal objectivo superar os problemas inerentes de uma *blockchain* pública, como o *Bitcoin*, tais como escalabilidade, privacidade e custo. A mitigação destes problemas é feita através de uma gestão integrada de permissões de utilizadores [20].

Esta plataforma gere identidades e segurança com criptografia assimétrica. Cada utilizador aleatoriamente gera as suas chaves privadas nunca as revelando aos outros participantes. Cada chave destas tem uma relação matemática com um endereço público que representa uma identidade para receber fundos. Assim que enviado para um endereço público, esses fundos apenas podem ser gastos usando a correspondente chave privada para "assinar" uma nova transação. Neste sentido, a chave privada representa a pertença de quaisquer fundos que proteja e permite a um utilizador assinar uma mensagem para provar que possui a chave privada correspondente a um endereço particular. É através desta propriedade que a *MultiChain* consegue restringir o acesso a uma *blockchain* a uma lista de utilizadores permitidos, expandindo o processo de "*handshaking*" que ocorre quando dois nós *blockchain* se ligam.

Na *MultiChain* a gestão de privilégios é integrada via transações na rede que contêm metadados especiais. O mineiro do bloco "*genesis*", o primeiro da cadeia, recebe todos os privilégios, incluindo o direito de gerir os de outros utilizadores. Para atribuir privilégios a outros utilizadores são criadas transações cujos *outputs* contêm os endereços desses utilizadores, juntamente com metadados que indicam os privilégios concedidos. Sempre que existem alterações nos privilégios de administração e mineração há que haver um consenso, que é alcançado por votos de administradores de uma proporção mínima. Uma vez que a rede é descentralizada, há que garantir uma ordenação nas alterações das permissões nos nós desta, portanto, todos seguem a regra de que as transações são "reproduzidas" pela ordem *blockchain*. Desta forma, cada transação num bloco deve ser válida de acordo com o estado das permissões dos utilizadores que a precedem imediatamente. Para maior conveniência administrativa, privilégios temporários podem ser concedidos restringindo-os a um intervalo fixo de números de bloco. As transações que dependem de tais privilégios são válidas apenas em blocos cujos números estão no intervalo estabelecido. Para que uma *blockchain* seja genuinamente "privada", para cada endereço concedida uma permissão nessa cadeia, pelo menos um administrador deve conhecer a identidade real da entidade que está a usar esse endereço.

Para resolver o problema da monopolização do processo de mineração por parte de um participante, a *MultiChain* impõe uma restrição no número de blocos que podem ser criados pelo mesmo mineiro, dentro de uma dada janela, através de um parâmetro denominado por diversidade de mineração. Este parâmetro define a proporção de mineiros permitidos que precisariam de conspirar para minar a rede. Com isto, um bloco é validado da seguinte maneira: **1)** Aplicar todas as alterações de permissões definidas pelas transações no bloco, por ordem; **2)** Contar o número de mineiros permitidos que são definidos após a aplicação dessas mudanças; **3)** Multiplicar o número de mineiros pela diversidade de mineração, arredondando para cima para obter o espaçamento; **4)** Se o mineiro deste bloco extraiu um dos espaçamento - 1 blocos anteriores, o bloco é inválido. Isto impõe um agendamento *round-robin*, no qual os mineiros autorizados devem criar blocos em rotação para gerar uma *blockchain* válida.

A diversidade de mineração para além de prevenir abusos, ajuda quando surgem

divisões de rede temporárias. Como nestes casos ocorrem bifurcações na visão sobre a *blockchain*, esta assegura que a maior cadeia de blocos pertencerá à divisão com a maioria de mineiros permitidos, já que a outras vão rapidamente congelar.

Com esta plataforma é possível configurar e trabalhar com várias *blockchains* simultaneamente. Esta configuração é feita através de um ficheiro de configuração que define todos os parâmetros das *blockchains* pretendidas, entre eles, o protocolo da cadeia (*blockchain* privada ou *Bitcoin* puro), tempo alvo para blocos, tipos de permissões activas, diversidade de mineração (apenas em *blockchains* privadas), entre outros. É permitido que múltiplas *blockchains* estejam activas num único servidor e também executar uma *blockchain* em múltiplos nós.

Nas *blockchains* privadas da *MultiChain*, esquemas de multi moeda são integrados suportando ativos de terceiros diretamente nas regras da cadeia. Isto é conseguido codificando os identificadores e quantidades de todos os ativos em cada *output* de uma transação, usando uma extensão da linguagem de *scripting* do *Bitcoin*. A regra para validar transações é estendida para verificar se as quantidades totais de todos os ativos nos *outputs* de uma transação são exatamente iguais ao total dos seus *inputs*. Finalmente, realça-se que a *MultiChain* está desenhada para permitir transições suaves entre *blockchains* privadas e a *blockchain* do *Bitcoin*.

2.2.3 *Ethereum*

A *Ethereum* é uma plataforma descentralizada que executa contratos inteligentes e fornece uma *blockchain* com uma linguagem de programação *Turing-complete* embutida, que pode ser usada para criar esses contratos e aplicações descentralizadas [8]. Estes correm exatamente como programados sem a necessidade de terceiros e os utilizadores podem criar sistemas, por exemplo de propriedades inteligentes, apenas escrevendo a sua lógica em linhas de código. O principal objectivo desta plataforma é fundir e aperfeiçoar os conceitos de *scripting*, moedas alternativas e meta-protocolos *on-chain* para garantir escalabilidade, estandardização, *feature-completeness*, facilidade de desenvolvimento e interoperabilidade oferecida por estes diferentes paradigmas [8].

Na *Ethereum* o estado é composto por objetos denominados contas e as transições de estado são transferências diretas entre contas. Ether é o principal cripto-combustível da *Ethereum* e é usado para pagar taxas de transação. Essencialmente existem dois tipos de contas: contas de propriedade externa, controladas por chaves privadas, e contas de contrato, controladas pelo seu código de contrato. O envio de mensagens a partir de uma conta de propriedade externa é feita criando e assinando uma transação, já nas contas de contrato, sempre que recebem uma mensagem o seu código é ativado, o que permite ler e escrever em armazenamento interno e enviar outras mensagens ou criar contratos. O código nos contratos da *Ethereum* é escrito numa linguagem de *bytecode* baseada em pilha e de baixo nível, conhecida como código de máquina virtual *Ethereum* ou código EVM. O código consiste numa série de *bytes*, onde cada byte representa uma operação [40].

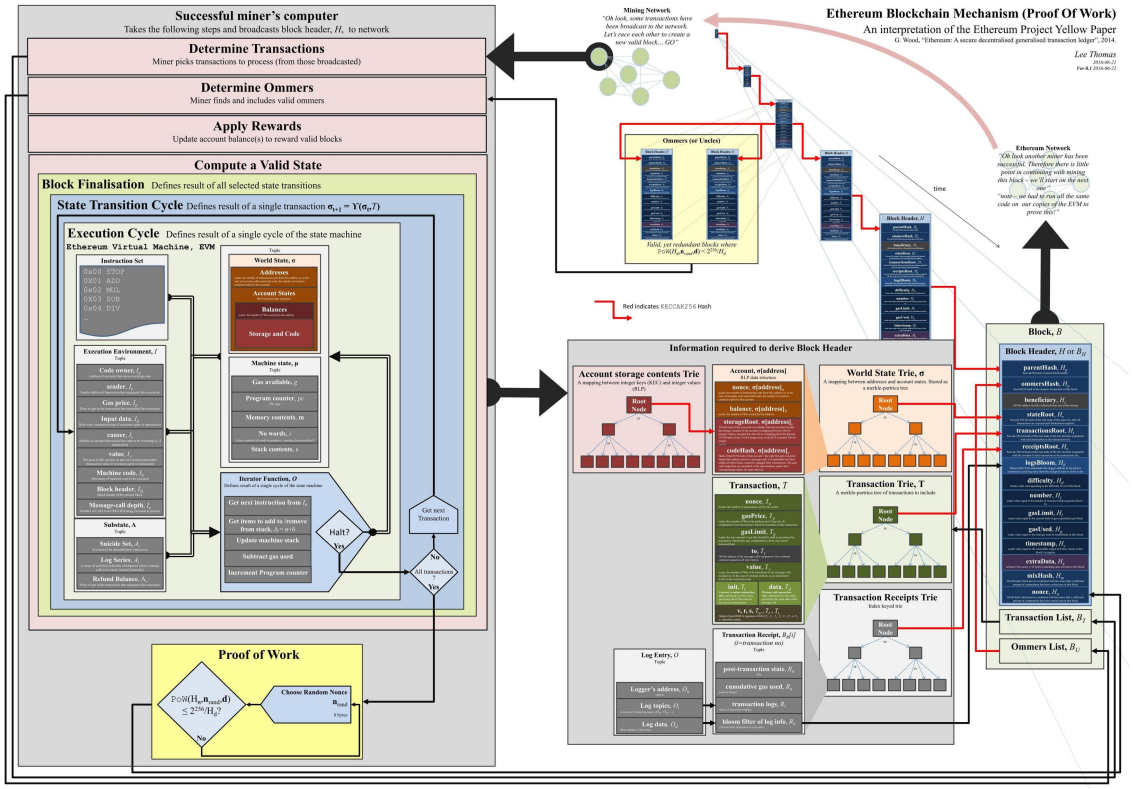


Figura 2.2: Ilustração detalhada da Máquina Virtual *Ethereum* (fonte indicada na figura)

Uma mensagem pode ser criada por uma entidade externa ou por um contrato e pode conter dados através de uma opção explícita. Caso o receptor da mensagem seja uma conta de contrato, há a possibilidade dessa conta devolver uma resposta, abrangendo o conceito de função. O termo "transação" é usado na *Ethereum* para representar um pacote de dados assinado que armazena uma mensagem a ser enviada de uma conta de propriedade externa. As transações contêm o destinatário da mensagem, uma assinatura identificando o remetente, a quantidade de ether e os dados a serem enviados, além de dois valores denominados *STARTGAS* (evita a expansão exponencial e ciclos infinitos no código) e *GASPRICE* (taxa a pagar ao mineiro por etapa computacional). Se a execução da transação "ficar sem gás", todas as alterações de estado serão revertidas - excepto para o pagamento das taxas, e se a execução da transação for interrompida com algum gás restante, a parte restante das taxas será reembolsada ao remetente. Há também um tipo de transação e um tipo de mensagem correspondente para criar um contrato. Uma vantagem desta plataforma é que as partes não precisam de se preocupar com o tipo de conta de cada uma num contrato.

A função de transição de estados, $APPLY(S, TX) -> S'$, permite através de um estado (S) e de uma transação (TX) devolver um novo estado (S') como resultado. Tal como as transações, os contratos podem assegurar os seus recursos computacionais limitados, estabelecendo limites estritos nas sub-computações que eles geram.

Cada operação da EVM tem acesso a 3 tipos de espaço para armazenar dados: a pilha,

um *container* LIFO na qual valores de 32 *bytes* podem ser *pushed* ou *popped*; memória, um vector de *bytes* infinitamente expansível; e o armazenamento de longo prazo do contrato, um armazenamento chave/valor onde chaves e valores são ambos de 32 *bytes*. Ao contrário da pilha e da memória, que são reiniciadas após o fim da computação, o armazenamento persiste por um longo período. O código pode aceder ao valor, ao remetente e aos dados da mensagem recebida, bem como aos dados do cabeçalho do bloco e também pode retornar um vector de *bytes* de dados como *output*.

Os blocos *Ethereum* contêm uma cópia da lista de transações e do estado mais recente. Além disso, dois outros valores, o número do bloco e a dificuldade, também são armazenados no bloco. O estado é armazenado numa estrutura em árvore e, após cada bloco, apenas uma pequena parte da árvore precisa de ser alterada. Assim, em geral, entre dois blocos adjacentes, a grande maioria da árvore deve ser a mesma e, portanto, os dados podem ser armazenados uma vez e referenciados duas vezes usando apontadores (ou seja, *hashs* de sub árvores). Um tipo especial de árvore conhecido como "árvore Patrícia" é usada para realizar isto, incluindo uma modificação no conceito da árvore *Merkle* que permite que os nós sejam inseridos e excluídos, e não apenas alterados, com eficiência. Além disso, como todas as informações de estado fazem parte do último bloco, não há necessidade de armazenar todo o histórico da *blockchain*.

2.2.4 Contratos Inteligentes

Nesta sub-seção é analisado um confronto entre as abordagens aos contratos inteligentes de quatro diferentes plataformas [21].

Atualmente existem diversas plataformas que oferecem diferentes formas de colocar código numa *blockchain*. Com isto, ao selecionar uma plataforma para implementar uma *blockchain*, há que raciocinar cautelosamente acerca dos vários tipos de contratos inteligentes que suporta. Os contratos inteligentes automatizam a execução de interações entre partes, sem requerer um intermediário confiável, exprimindo relações legais em código em vez de palavras e prometem permitir transações a ocorrer diretamente sem erros. Estes são equivalentes a código de aplicação, mas descentralizados: em vez de serem executados num local central são executados em vários nós na *blockchain*, criando ou validando transações que modificam o conteúdo de uma base de dados. Mais tecnicamente, um contrato deste tipo é código de computador que "vive" dentro de uma *blockchain* e define as regras das transações dessa cadeia. Tais plataformas suportam alguma forma de código *on-chain* customizável, com diferentes terminologias, mas que em última instância representam código específico de aplicação que define as regras de uma cadeia. A *Hyperledger Fabric* da IBM chama aos seus contratos "*chaincode*", a *MultiChain* na sua versão 2.0 introduz *smart filters*, a *Ethereum* popularizou o nome de "*smart contracts*" e finalmente o *Corda* referencia "*contracts*" nas suas transações.

Qualquer aplicação de base de dados partilhada pode ser desenhada usando uma

blockchain, embora tal abordagem crie certos desafios técnicos que não existem num cenário centralizado. Dentro destes desafios incluem-se como assegurar que os participantes seguem as regras de transação, como garantir determinismo no sistema e como prevenir conflitos entre transações. É na resolução destes desafios que os contratos inteligentes surgem, trabalhando com a infraestrutura subjacente da *blockchain*.

As regras de transação restringem as transformações que podem ser realizadas em bases de dados baseadas em *blockchain*. Estas são necessárias porque qualquer participante pode iniciar uma transação na *blockchain* e, porque, os participantes não confiam uns nos outros o suficiente para permitir modificações à base de dados sem restrições. As plataformas *MultiChain* e *Corda* expressam estas regras listando explicitamente linhas ou "estados" da base de dados que são removidas ou criadas, formando um conjunto de "*inputs*" e "*outputs*", respectivamente (*input-output model*). Com os conjuntos de *inputs* e *outputs* a validade de uma transação na *MultiChain* ou *Corda* é definida por código que realiza computações arbitrárias nesses conjuntos.

As plataformas *Hyperledger Fabric* e *Ethereum*, por sua vez, exprimem regras de transação com a criação de múltiplos contratos na *blockchain*, cada um com a sua própria base de dados e código associado. Assim, a base de dados de um contrato apenas pode ser modificada pelo seu código. Com este modelo, uma transação *blockchain* começa como uma mensagem enviada para um contrato com alguns parâmetros ou dados opcionais. O código do contrato é executado em reação à mensagem e aos parâmetros, e é livre de ler e escrever na sua própria base de dados como parte dessa reação. Os contratos também podem enviar mensagens para outros contratos, mas não podem aceder às bases de dados uns dos outros diretamente. Ao contrário de todas as outras, a *MultiChain* possui várias abstracções integradas que fornecem alguns blocos de construção básicos para aplicações controladas por *blockchain*, sem exigir que os desenvolvedores escrevam o seu próprio código.

Tabela 2.1: Regras de transação das diferentes plataformas (retirada de [21])

Regras de Transação	**Modelo**	**Inbutidas**
Hyperledger Fabric	*Contract-message*	Nenhuma
MultiChain	*Input-output*	Permissões + ativos + *streams*
Ethereum	*Contract-message*	Nenhuma
Corda	*Input-output*	Nenhuma

A execução repetida e redundante do código das regras de transação em diferentes nós, para validação de transações, exige um requisito raro nos sistemas centralizados: determinismo. Determinismo significa que um pedaço de código devolve sempre o mesmo resultado para os mesmos parâmetros, o que é completamente crucial para código na *blockchain*, pois, sem ele, o consenso entre os nós na cadeia pode ser quebrado.

O não-determinismo em código na *blockchain* é possível e pode acontecer devido,

por exemplo, à execução de várias partes de código em "*threads*" paralelos, o que leva a uma "*race condition*" em que a ordem na qual esses processos terminam não pode ser prevista, ou à execução de cálculos de vírgula flutuante que podem fornecer respostas minuciosamente distintas em diferentes arquiteturas de processadores.

Os contratos na plataforma *Ethereum* são expressos num formato denominado "*Ethereum bytecode*" que garante determinismo, uma vez que, esse formato não consegue codificar nenhuma operação não-determinística. Já os *filters* e *contracts* nas plataformas *MultiChain* e *Corda*, respetivamente, adaptam linguagens de programação existentes e ambientes de *runtime* para impedir o não-determinismo. De forma a garantir determinismo, a *Hyperledger Fabric* recorre a nós "*endorsers*", onde cada *chaincode* tem uma "política de aprovação" que define exactamente o nível de aprovação necessário para validar uma transação (e.g. aprovação de 50% dos *endorers* da *blockchain*).

Tabela 2.2: Determinismo nas diferentes plataformas (retirada de [21])

Determinismo	Modelo	Linguagens	Visibilidade do Código	Imposto
Hyperledger Fabric	*Endorsements*	Go + Java + JavaScript	*Counterparties + endorsers*	Não
MultiChain	*Runtime adaptado*	JavaScript	*Blockchain*	Não
Ethereum	Máquina Virtual	Solidity	Blockchain	Não
Corda	Runtime adaptado	Java + Kotlin	Counterparties + dependents	Não (por agora)

Para lidarem com a possibilidade de duas transações entrarem em conflito, os *filters* da *MultiChain* e os *contracts* do *Corda*, como implementam regras de transações através do modelo *input-output*, restringem que qualquer *output* seja gasto mais do que uma vez. A *Ethereum* utiliza *smart contracts* e mensagens em vez de *inputs* e *outputs*, portanto, os conflitos entre transações não são de imediato visíveis no motor da *blockchain* e apenas são detectados e bloqueados pelo contrato que processa as transações, depois das suas ordens serem confirmadas na cadeia. Na plataforma *Hyperledger Fabric*, os conflitos são resolvidos através de conjuntos de leitura-escrita. Cada *endorser* regista o conjunto de linhas que seriam lidas e escritas, anotando também a versão exacta dessas linhas nesse instante. Esse "conjunto de leitura e escrita" de linhas com versão é explicitamente referenciado e incluído na transação que o cliente transmite. No entanto, se uma transação lê ou escreve uma versão de uma linha da base de dados que já tenha sido modificada por uma transação anterior, essa segunda transação é ignorada.

Tabela 2.3: Prevenção de conflitos nas diferentes plataformas (retirada de [21])

Prevenção de Conflitos	Modelo	Verificação	Velocidade
Hyperledger Fabric	Conjuntos *read-write*	Independente	*~10s (confirmação)*
MultiChain	Gasto único	Independente	*~1s (propagação)*
Ethereum	Verificação de Contratos	Independente	~10s (confirmação)
Corda	Gasto único	Notários confiáveis	0~5s (notário)

2.3 Sistemas de replicação de máquinas de estado tolerantes a falhas Bizantinas

Na secção 2.2 sintetizaram-se sistemas capazes de resistir a falhas bizantinas, mas que dispunham os seus dados numa cadeia de blocos distribuída, segura criptograficamente, para gerar concordância entre vários nós numa rede. Enquanto que estes se baseiam num registo distribuído, os sistemas detalhados nesta secção utilizam um mecanismo diferente para replicar os dados: replicação de máquinas de estado.

O modelo de falhas bizantino compreende processos defeituosos que arbitrariamente podem provocar distúrbios num sistema, desviando-o da sua especificação. Estes processos têm a capacidade de personificar outras identidades, duplicar as mensagens que enviam, enviá-las com valores errados ou até mesmo não enviar qualquer mensagem [10].

Apesar de existirem variadas técnicas de replicação, duas diferentes abordagens são particularmente bem conhecidas, replicação ativa e replicação passiva. A replicação de máquinas de estado é um tipo de replicação ativa onde as operações dos clientes são ordenadas por um protocolo de ordenação e redirecionadas diretamente para um conjunto de réplicas [10]. Cada réplica executa posteriormente a operação e, para que estejam todas num estado consistente, é necessário que o processamento seja determinístico, isto é, dada a mesma operação de um cliente e o mesmo estado, a mesma atualização do estado é produzida por cada réplica. A replicação ativa, apesar de não conseguir lidar com processamento não determinístico, consegue mascarar falhas sem degradação de desempenho, e caso as operações possuam uma componente computacional forte pode desperdiçar recursos computacionais [10].

2.3.1 O Primeiro Sistema Prático

Nesta subsecção é apresentado o primeiro algoritmo de replicação de máquinas de estado tolerante a falhas bizantinas, prático, capaz de operar em ambientes assíncronos e que incorpora optimizações relevantes que o permitem funcionar eficientemente [9].

Este oferece propriedades de *liveness* e *safety* a no máximo $\frac{n-1}{3}$ réplicas de um total de n que estão simultaneamente com defeito. Com isto, os clientes eventualmente recebem respostas corretas aos seus pedidos, satisfazendo linearidade [9]. O algoritmo é invulnerável a determinados ataques de negação de serviço, uma vez que, não se baseia em sincronismo para fornecer *safety*. Para além de usar apenas um *round trip* para executar operações *read-only* e dois *round trip* para executar operações *read-write*, a autenticidade da comunicação usa um esquema de autenticação baseado em códigos de autenticação de mensagens, quando o sistema opera normalmente, e criptografia de chave-pública apenas em caso de falhas [9].

O modelo de sistema para este algoritmo é um sistema assíncrono distribuído com nós ligados por uma rede, rede esta que pode falhar na entrega de mensagens, atrasá-las, duplicá-las ou entregá-las fora de ordem. Supõe-se um modelo de falhas bizantino, ou seja, nós que falham comportam-se arbitrariamente, sendo as falhas independentes entre si. Para prevenir *spoofing*, *replays* e para se detectarem mensagem corrompidas são usadas técnicas criptográficas como assinaturas de chave-pública, códigos de autenticação de mensagens e *digests* produzidos por funções de *hash* resistentes a colisões. Com todas as réplicas a saberem as chaves públicas das outras, para verificarem assinaturas, assume-se que um adversário não pode nem atrasar nós correctos indefinidamente nem subverter as técnicas criptográficas usadas.

O algoritmo pode ser usado para implementar qualquer serviço replicado determinístico com um estado e algumas operações. Os clientes emitem solicitações para o serviço replicado para realizarem operações e bloqueiam à espera por uma resposta. O serviço replicado é implementado por réplicas e os clientes e as réplicas não têm falhas se seguirem o algoritmo e se nenhum invasor forjar a sua assinatura. É como se fosse uma implementação centralizada que executa operações atomicamente, uma de cada vez. *Safety* requer o limite do número de réplicas defeituosas, porque uma réplica defeituosa pode comportar-se arbitrariamente, e por exemplo, destruir o seu estado. A resiliência do algoritmo é óptima sendo $3f + 1$ o número mínimo de réplicas que permitem que um sistema assíncrono forneça as propriedades de *safety* e de *liveness* quando até f réplicas estiverem com defeito. São necessárias muitas réplicas porque deve ser possível prosseguir com a comunicação com $n - f$ réplicas, pois as n réplicas podem estar com defeito e não responder. O algoritmo não resolve o problema da privacidade tolerante a falhas, portanto uma réplica defeituosa pode libertar informações para um invasor. Não é viável oferecer privacidade tolerante a falhas no caso geral, porque as operações do serviço podem executar cálculos arbitrários usando os seus argumentos e o estado do serviço, ou seja, as réplicas precisam dessas informações em claro para executar essas operações com eficiência.

Este algoritmo é uma forma de replicação de máquinas de estado, dado que o serviço é modelado como uma máquina de estados que é replicada em diferentes nós num sistema distribuído. Cada réplica de máquina de estados mantém o estado do serviço

e implementa as operações do serviço. Por simplicidade, assume-se que $3f + 1$ é o número máximo de réplicas que podem estar com defeito, mas, embora possa haver mais de $3f + 1$ réplicas, réplicas adicionais degradam o desempenho (já que mais e maiores mensagens serão trocadas) sem fornecer resiliência aprimorada. As réplicas passam por uma sucessão de configurações chamadas *views*. Numa *view*, uma réplica é a primária e as outras são *backups*. As *views* são numeradas consecutivamente. As alterações às *views* são realizadas quando parece que a primária falhou. O algoritmo funciona aproximadamente da seguinte forma [9]: **1)** Um cliente envia uma solicitação para realizar uma operação do serviço para a primária; **2)** A primária faz *multicast* à solicitação para os backups; **3)** As réplicas executam a solicitação e enviam uma resposta para o cliente; **4)** O cliente aguarda por $f + 1$ respostas de réplicas diferentes com o mesmo resultado, sendo este o resultado da operação. Como todas as técnicas de replicação de máquina de estado, impõe-se dois requisitos sob as réplicas: devem ser determinísticas e começar no mesmo estado. Tendo em conta estes dois requisitos, o algoritmo garante a propriedade de *safety*, garantindo que todas as réplicas sem falhas concordam numa ordem total para a execução de pedidos, apesar de falhas.

Quando a primária recebe um pedido dum cliente, dá início a um protocolo de três fases para atomicamente fazer o *multicast* do pedido para as réplicas. As três fases são pré-preparar, preparar e confirmar. As fases de pré-preparação e preparação são usadas para ordenar totalmente as solicitações enviadas na mesma *view*, mesmo quando a primária, que propõe a ordenação de solicitações, estiver com defeito. As fases de preparação e consolidação são usadas para garantir que as solicitações que são confirmadas são ordenadas totalmente entre as *views*. O protocolo de mudança de *view* fornece *liveness* permitindo ao sistema progredir quando a primária falha. As alterações de *view* são accionadas por *timeouts* que impedem que os *backups* aguardem indefinidamente por solicitações a executar.

2.3.2 Hybrid Quorum

O Hybrid Quorum, um protocolo híbrido de replicação de máquinas de estados tolerante a falhas bizantinas, combina técnicas de *quorum* e de replicação de máquinas de estado baseadas em acordo para superar o problema do custo quadrático da comunicação entre réplicas, desnecessário em caso de não contenção, e do grande número de réplicas necessárias e pobre execução sob contenção do protocolo Q/U (*Query*/*Update*) [15]. Estas complicações são resolvidas na ausência de contenção, através de um novo e leve protocolo de *quorum* bizantino, no qual *reads* (operações que não alteram o estado do serviço) exigem um *round trip* (uma fase) entre o cliente e as réplicas, e *writes* (operações que alteram o estado do serviço) exigem dois *round trip* (duas fases) [15]. Quando ocorre contenção, este usa o algoritmo de replicação de máquinas de estados tolerante a falhas bizantinas, que escala à medida que o número de falhas aumenta, para ordenar eficientemente operações concorrentes. Para além disto, o Hybrid Quorum usa apenas $3f + 1$

réplicas para tolerar f falhas, o que oferece resiliência óptima a falhas nos nós [15].

O modelo deste protocolo consiste num conjunto de processos cliente e num conjunto de até $3f + 1$ processos servidor ou réplicas. Estes processos podem ser correctos e obedecer à sua especificação, ou defeituosos e poderem desviar-se arbitrariamente da sua especificação, ou seja, assume-se um modelo de falhas Bizantino. É assumido também um sistema distribuído assíncrono de nós ligados através de uma rede totalmente ligada, onde dado um identificador de um nó, qualquer outro pode (tentar) contactar o primeiro directamente enviando-lhe uma mensagem, que pode falhar entrega de mensagens, atrasá-las, duplicá-las, corrompê-las, entregá-las fora de ordem e não existem limites nos atrasos das mensagens ou no tempo de execução das operações. Para *liveness* é requerido apenas que se um cliente continuar a retransmitir um pedido para um servidor correto, a resposta a esse pedido irá eventualmente ser recebida. Os nós do sistema usam assinaturas digitais para autenticar a comunicação, logo, apenas é possível alterar o conteúdo de uma mensagem conhecendo a chave privada do remetente. Para estabelecer comunicações seguras entre pares de nós são usados códigos de autenticação de mensagens, assumindo a existência de um mecanismo confiável de distribuição de chaves. Por fim, é admitida a existência de funções de *hash* resistentes a colisões de forma a assegurar a integridade da mensagem.

Cada fase, quer de *writes* quer de *reads*, consiste no cliente emitir uma chamada de procedimento remoto para todas as réplicas e coleccionar um *quorum* de respostas. Neste protocolo, os *quorums* são de tamanho $2f + 1$, e são usados certificados que garantem uma ordenação correta para as operações de *write*. Estes certificados são um *quorum* de mensagens autenticadas de diferentes réplicas, todas garantindo um facto. O propósito da 1ª fase de *write* é obter um *timestamp* que determina a ordenação de um determinado *write* relativamente a outros. A conclusão com sucesso desta fase, fornece ao cliente um certificado que prova que pode executar a sua operação no *timestamp* obtido. O cliente usa então este certificado para convencer as réplicas a executar a sua operação no *timestamp* na 2ª fase de *write*. Uma operação de *write* termina quando $2f + 1$ réplicas processaram o pedido da 2ª fase de *write*, e o cliente coleccionou as respectivas respostas. Na ausência de contenção, um cliente obtém um certificado utilizável no fim da 1ª fase e conseguirá executar a 2ª. Na presença de clientes lentos ou que falharam, o progresso é garantido pelas operações *writeBackWrite* e *writeBackRead*, permitindo a outros clientes completarem a 2ª fase. No entanto quando existe contenção, um cliente pode não obter um certificado de *write* utilizável e, nesse caso, solicita que o sistema resolva a contenção. É possível que algumas réplicas já tenham agido num pedido de 2ª fase de um *write* para executar uma operação num *timestamp*, mas como resultado da resolução de contenção, podem precisar de desfazer essa *atividade*. Portanto, todas as réplicas mantêm um único estado *backup* para que possam desfazer o último *write* executado.

Na arquitectura do sistema o código é executado como *proxies* nas máquinas dos clientes e dos servidores: o código de aplicação no cliente chama uma operação no *proxy* do cliente, enquanto o código do servidor é chamado pelo *proxy* do servidor em resposta

a solicitações do cliente. O código do servidor mantém o estado da réplica, executa operações de aplicação e também deve poder desfazer a operação recebida mais recente, mas ainda não concluída, para manipular a reordenação na presença de contenção. As réplicas também executam o protocolo de replicação de máquinas de estado tolerante a falhas bizantinas para a resolução de contenção. De notar que o protocolo de replicação de máquinas de estado tolerante a falhas bizantinas não está envolvido na ausência de contenção. O sistema também consegue suportar transações com vários objectos. As réplicas *writers* têm permissão para modificar objectos diferentes em paralelo, mas estão restritas a ter apenas uma operação pendente num objecto específico de cada vez. Este tipo de réplicas numera as suas solicitações em objectos individuais sequencialmente, o que evita repetir a execução de operações de modificação e pode consultar o sistema para obter informações sobre o seu *write* mais recente a qualquer momento.

A transferência de estado é necessária para actualizar as réplicas lentas, para que possam executar *writes* mais recentes. Uma réplica detecta que perdeu algumas actualizações quando recebe um certificado válido para executar um *write* no *timestamp* t, mas possui um valor existente de *timestamp* no seu certificado *atual* menor que $t-1$.

Uma réplica solicita a transferência de estado a $f+1$ réplicas, fornecendo um intervalo de *timestamp* para as actualizações necessárias. Uma réplica é designada para retornar as actualizações, enquanto outras enviam um *hash* sobre esse *log* parcial. As respostas são solicitadas a outras réplicas se os *hashes* não concordarem com o *log* parcial ou após um tempo limite seja excedido. Como o *log* parcial é provável que seja consideravelmente maior que f *hashes*, o custo da transferência de estado é essencialmente constante em relação a f.

Para evitar a transferência de grandes *logs* parciais, propõe-se pontos de verificação regulares do sistema para estabelecer o estado completo em todas as réplicas. Isto reduz o custo de *write-back* subsequente e permite que os *logs* antes do ponto de verificação sejam descartados. Para minimizar ainda mais o custo da transferência de estado, os registos de *log* podem ser comprimidos, explorando *overwrites* e semânticas específicas de aplicação. Alternativamente, o estado pode ser transferido na forma de *differences* ou árvores *Merkle*.

2.3.3 Swirlds Hashgraph

O algoritmo de consenso *hashgraph* da plataforma Swirlds é um sistema de replicação de máquinas de estado tolerante a falhas bizantinas e justo (no sentido em que é difícil para um atacante manipular qual de duas transações será escolhida para ser a primeira na ordem de consenso) [5]. É assíncrono, com consenso eventual com probabilidade um e de alta velocidade na ausência de falhas [5]. Baseando-se no protocolo *gossip*, os seus participantes constroem, em conjunto, um *hashgraph* que reflecte todos os eventos *gossip*, o que torna possível um acordo bizantino através de votação virtual (sem enviar quaisquer votos pela rede) [5]. Com tudo isto, é possível chegar a uma concordância bizantina acerca da ordem total das transações, sem grande *overhead*.

A segurança deste sistema é composta por assinaturas digitais seguras, para que atacantes não modifiquem mensagens sem serem detectados e por funções de *hash* seguras resistentes a colisões. Sem utilizar um líder é resolvente a ataques de negação de serviço em pequenos conjuntos de participantes/membros.

O algoritmo de consenso via *hashgraph* é equivalente a uma cadeia de blocos na qual a "cadeia" está constantemente a ramificar, sem qualquer remoção, onde nenhum bloco é obsoleto, e onde cada mineiro pode minar muitos novos blocos por segundo, sem prova de trabalho e com 100% de eficiência.

Qualquer membro em qualquer instante pode criar e assinar uma transação e todos os membros obtêm uma copia dessa, através do protocolo *gossip*, o que permite à comunidade alcançar uma concordância bizantina sobre a ordem das transações. É usada uma estrutura de dados, um grafo dirigido (neste caso denominado *hashgraph*) que regista como os membros comunicaram, que é espalhada através do protocolo *gossip*, formando um histórico. Este grafo possui esta denominação, *hashgraph*, pois cada evento possui um *hash* de dois eventos pai, *self-parent* e *other-parent*, e, para além disso, pode conter também um *payload* de quaisquer novas transações, e respectiva assinatura digital do criador. Dado que qualquer membro tem uma cópia do *hashgraph*, todos podem calcular a ordem total dos eventos de acordo com uma função determinística desse grafo e todos obterão a mesma resposta (votação virtual). Assim, o consenso é alcançado sem qualquer envio de mensagens de votação, com gasto zero de *bandwidth*, para além do *gossiping* do *hashgraph*.

A abordagem usada para se chegar a um acordo bizantino é um protocolo que escolhe alguns eventos (vértices no *hashgraph*), denominados testemunhas (*witnesses*), e as define como sendo famosas se o grafo mostrar que a maioria dos membros as receberam logo após a sua criação. Com isto, corre-se um protocolo de acordo bizantino apenas sob testemunhas, decidindo para cada uma se é famosa. Após se estabelecer um acordo bizantino no conjunto exacto de testemunhas famosas, é fácil derivar a partir do *hashgraph* uma ordem total justa para todos os eventos. Para finalizar, o conceito de *strongly seeing*, que diz que o evento x pode fortemente ver o evento y no *hashgraph*, se estiverem ligados por múltiplos caminhos dirigidos que passam por membros suficientes, é peça chave para permitir votações consistentes e garantir que diferentes membros nunca vão calcular resultados inconsistentes, mesmo com votações puramente virtuais, o que faz com que se chegue a um acordo bizantino com probabilidade 1.

2.4 Intel Software Guard Extensions

As Intel SGX são uma tecnologia desenvolvida para atender às necessidades da indústria da computação confiável, de maneira semelhante ao ARM TrustZone [25], mas direcionada para plataformas *desktop* e servidores. Esta permite que o código local de um utilizador crie regiões de memória privadas, chamadas enclaves, que são isoladas de outros processos em execução no mesmo ou superior nível de privilégio. O código executado

dentro de um enclave é efetivamente isolado de outras aplicações, sistema operativo, *hypervisor*, entre outros.

Foi introduzida em 2015 com os processadores Intel Core de sexta geração, baseados na microarquitetura *Skylake*. O suporte às SGX pode ser verificado executando a instrução CPUID com o sinalizador *Structured Extended Feature Leaf* definido e verificando se o segundo bit do registo EBX está definido. Para poder usar as SGX, devem ser ativadas pela/na BIOS, e apenas algumas suportam esta tecnologia. Esta é uma das razões pela qual não é amplamente utilizada.

2.4.1 Natureza Interna das Intel SGX

A implementação das Intel SGX pode ser resumida em alguns pontos:

- uma aplicação é dividida em duas partes: uma segura e uma não segura;
- a aplicação inicia o enclave e este é colocado na memória protegida;
- quando uma função do enclave é chamada, somente o código dentro do enclave pode ver os seus dados. Os acessos externos são sempre negados;
- quando a função retorna os dados do enclave permanecem na memória protegida.

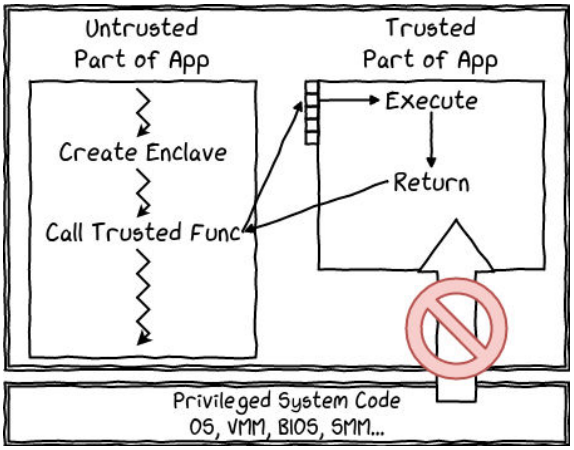


Figura 2.3: Composição de uma aplicação SGX (retirada de [1])

O ambiente de execução segura faz parte do processo do *host*, o que significa que:

- uma aplicação contém o seu próprio código, os seus próprios dados e o enclave;
- o enclave contém o seu próprio código e os seus próprios dados;
- as SGX protegem a confidencialidade e a integridade do código e dos dados do enclave;
- os pontos de entrada do enclave são predefinidos durante a compilação;
- *multi-threading* é suportado, mas a sua implementação de forma correta não é trivial;
- um enclave pode aceder à memória da sua aplicação, mas não o contrário.

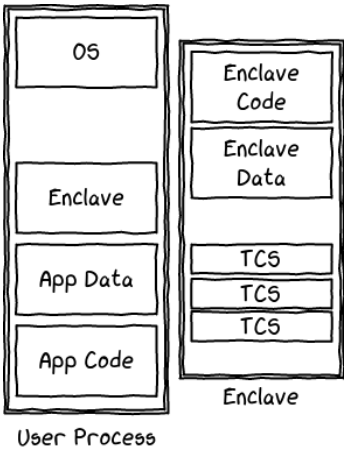


Figura 2.4: Disposição da memória de uma aplicação SGX (retirada de [1])

As Intel SGX definem dezoito novas instruções: treze a serem usadas pelo supervisor e cinco pelo utilizador. Todas elas são implementadas em microcódigo (para que o seu comportamento possa ser modificado). Na tabela 2.4 e na tabela 2.5 estão a lista completa de instruções.

As Intel SGX também definem treze novas estruturas de dados: oito são usadas para a gestão de enclaves, três para gestão de páginas de memória e duas para gestão de recursos. Abaixo está a lista completa das estruturas:

- SGX Enclave Control Structure (SECS)
- Thread Control Structure (TCS)
- State State Area (SSA)
- Page Information (PAGEINFO)
- Security Information (SECINFO)

Tabela 2.4: Instruções SGX do Supervisor

Nome	Descrição
EADD	Adicionar uma página
EBLOCK	Bloquear uma página EPC
ECREATE	Criar um enclave
EDBGRD	Ler dados pelo *debugger*
EBDGWR	Escrever dados pelo *debugger*
EINIT	Inicializar um enclave
ELDB	Carregar uma página EPC como bloqueada
ELDU	Carregar uma página EPC como desbloqueada
EPA	Adicionar um vetor de versões
EREMOVE	Remover página da EPC
ETRACK	Ativar verificações EBLOCK
EWB	Escrever/invalidar uma página EPC

Tabela 2.5: Instruções SGX do Utilizador

Nome	Descrição
EENTER	Entrar num enclave
EEXIT	Sair de um enclave
EGETKEY	Criar uma chave criptográfica
EREPORT	Criar um relatório criptográfico
ERESUME	Reentrar num enclave

- Paging Crypto MetaData (PCMD)
- Version Array (VA)
- Enclave Page Cache Map (EPCM)
- Enclave Signature Structure (SIGSTRUCT)
- EINIT Token Structure (EINITTOKEN)
- Report (REPORT)
- Report Target Info (TARGETINFO)
- Key Request (KEYREQUEST)

O código e os dados do enclave são colocados numa área de memória especial denominada EPC (*Enclave Page Cache*). Esta área de memória é encriptada usando o *Memory*

Encryption Engine (MEE), um chip novo e dedicado. As leituras externas no *bus* de memória apenas podem observar dados encriptados. As páginas são desencriptadas apenas quando dentro do núcleo físico do processador. As chaves são geradas no momento da inicialização e são armazenadas no processador.

A verificação de páginas tradicional é estendida para impedir acessos externos a páginas EPC.

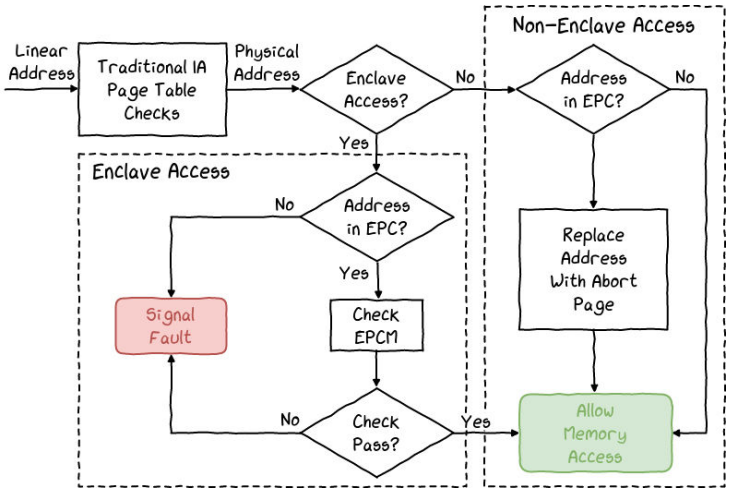


Figura 2.5: Verificação de páginas estendida (retirada de [1])

A estrutura *Enclave Page Cache Map* (EPCM) é usada para armazenar o estado das páginas. Esta está localizada dentro da memória protegida e o seu tamanho limita o tamanho da EPC (definido pelo BIOS, no máximo 128 MB). Este mapa contém a configuração, permissões e tipo de cada página.

A estrutura PAGEINFO é usada como parâmetro nas instruções de gestão da EPC para referenciar uma página. Esta contém os seus endereços lineares e virtuais e apontadores para as estruturas SECINFO e SECS.

A estrutura SECINFO é usada para armazenar os metadados das páginas: direitos de acesso (leitura/escrita/execução) e tipo (SECS, TCS, REG ou VA).

A estrutura PCMD é usada para procurar os metadados associados a uma página despejada. Esta contém a identidade do enclave ao qual a página pertence, um apontador para uma estrutura SECINFO e um MAC.

A estrutura VA é usada para armazenar os números de versão das páginas removidas do EPC. É um tipo de página especial que contém quinhentos e doze espaços de oito *bytes* para armazenar os números de versão.

A instrução EPA aloca uma página de memória com 4 *kilobytes* que contem o VA das páginas (número de versão da página) para proteger contra *replay*. Cada elemento tem 64 bits.

A instrução EBLOCK bloqueia todos os acessos à página que está a ser preparada para despejo. Todos os acessos futuros a esta página resultarão numa falha de página ("página bloqueada").

A instrução ETRACK remove uma página do EPC. A página deve ter sido preparada adequadamente: deve estar bloqueada e não deve ser referenciada pelo TLB. Antes de ser escrita na memória externa, a página é encriptada, um número de versão e metadados gerados e um MAC final é executado.

A instrução ELDB/ELDU carrega na memória uma página despejada anteriormente, num estado bloqueado ou não. Esta verifica o MAC dos metadados, número de versão (da entrada VA correspondente) e o conteúdo encriptado da página. Se a verificação for bem-sucedida, o conteúdo da página é desencriptado e colocado dentro da página EPC escolhida e a entrada VA correspondente apagada.

A memória EPC é definida pelo BIOS e de tamanho limitado. O SGX tem uma maneira de remover uma página do EPC, colocá-la na memória desprotegida e restaurá-la mais tarde. As páginas mantêm as mesmas propriedades de segurança, graças às instruções de gestão de páginas EPC, que permitem encriptar uma página e produzir metadados adicionais. Uma página não pode ser removida até que todas as entradas de *cache* referentes à mesma tenham sido removidas de todos os núcleos lógicos do processador. O conteúdo é exportado ou importado com uma granularidade de uma página (que é de *kilobytes*).

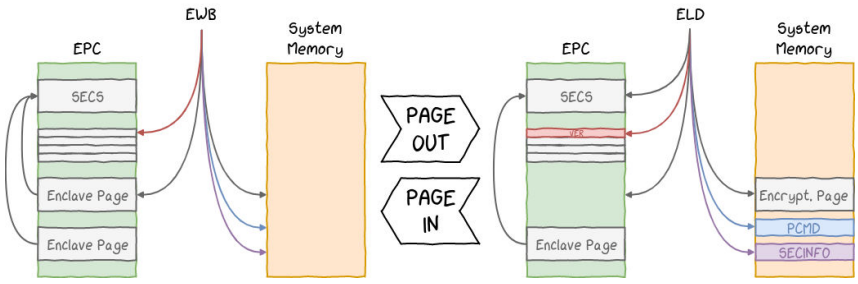


Figura 2.6: Gestão de páginas (retirada de [1])

Cada enclave está associado a uma estrutura SECS, que contem os seus metadados (por exemplo, o seu *hash* e tamanho). Esta não é acessível nem por código seguro nem por código não seguro, apenas pelo próprio processador. Também é imutável uma vez instanciado.

Cada enclave está associado a pelo menos uma estrutura TCS, que indica um ponto de execução no enclave. Como as SGX suportam *multi-threading*, um enclave pode ter tantos *threads* ativos quanto o TCS. Tal como a estrutura SECS, só é acessível pelo processador e também imutável.

Cada TCS está associado a pelo menos uma estrutura SSA, usada para salvar o estado do processador durante o tratamento de exceções e interrupções. É escrita ao sair e lida ao retomar.

Cada enclave pode usar a sua pilha e *heap*. Os registos RBP e RSP são salvos ao entrar e sair, mas o seu valor não é alterado. O *heap* não é manipulado internamente, os enclaves precisam do seu próprio alocador.

Cada enclave é representado por um *hash* dos seus atributos e da posição, conteúdo e

Figura 2.7: Conteúdo da EPC (retirada de [1])

proteção das suas páginas. Dois enclaves com o mesmo *hash* são idênticos. Esta medida é chamada MRENCLAVE e é usada para verificar a integridade do enclave. Cada enclave é também assinado pelo seu autor. Desta assinatura, resulta outra medida, o MRSIGNER, que contém o *hash* da chave pública do autor. Quer o MRENCLAVE quer o MRSIGNER são produzidos usando a função de *hash* SHA-256.

A estrutura EINITTOKEN é usada pela instrução EINIT para verificar se um enclave está autorizado a executar. Esta contém os atributos, *hash* e identidade do assinante do enclave. É autenticada usando um HMAC executado com a Chave de Inicialização.

Cada enclave é associado a uma estrutura SIGSTRUCT, assinada pelo seu autor e contém a medida do enclave, chave pública do assinante, número da versão (ISV, refletindo o nível de segurança) e identificador do produto (ISVPRODID, para distinção entre enclaves do mesmo autor). Esta permite garantir que o enclave não foi modificado e assinado novamente com uma chave diferente.

A instrução ECREATE instância um novo enclave, definindo o seu espaço de endereços e raiz da confiança. Estas informações são armazenadas num SECS recém-alocado.

A instrução EADD permite adicionar uma nova página ao enclave. O sistema operativo é o único responsável por escolher a página e o seu conteúdo. A entrada inicial para o EPCM indica o tipo de página e a sua proteção.

A instrução EEXTEND permite adicionar o conteúdo de uma página à medida do enclave, por blocos de 256 bytes. Deve ser chamada dezasseis vezes para adicionar uma página completa à medida.

A instrução EINIT verifica se o enclave corresponde ao seu EINITTOKEN (mesma medida e atributos) antes de o inicializar. Também verifica se o *token* está assinado com a Chave de Inicialização.

A instrução EREMOVE remove permanentemente uma página do enclave.

O processo de criação de um enclave divide-se nas seguintes etapas:

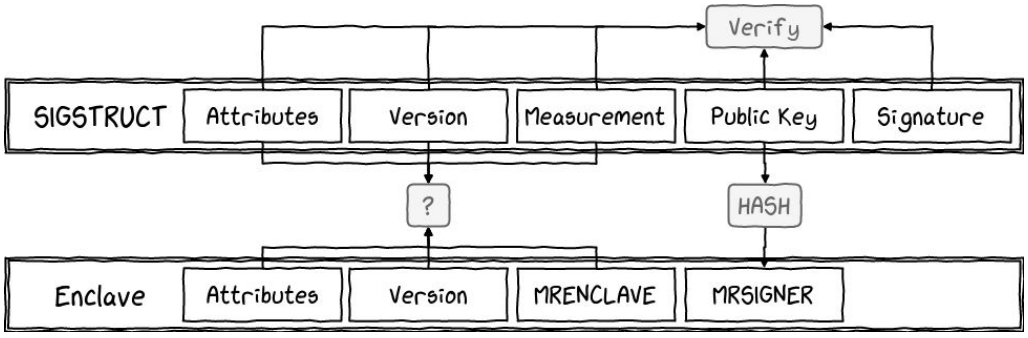


Figura 2.8: Verificação da SIGSTRUCT de um enclave (retirada de [1])

1. A aplicação solicita o carregamento do seu enclave na memória;
2. A instrução ECREATE cria e preenche a estrutura SECS;
3. Cada página é carregada na memória protegida usando a instrução EADD;
4. Cada página é adicionada à medida do enclave usando a instrução EEXTEND;
5. A instrução EINIT finaliza a criação do enclave.

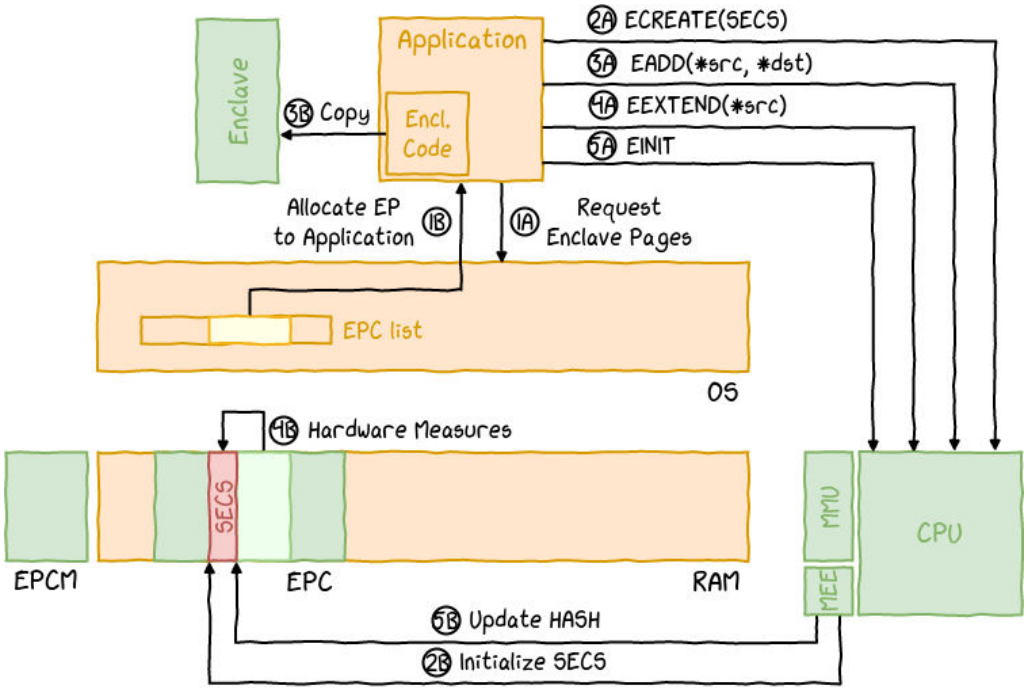


Figura 2.9: Criação de um enclave (retirada de [1])

A instrução EENTER transfere o controlo da aplicação para um local predeterminado dentro do enclave. Esta verifica se o TCS está livre e limpa as entradas TLB. Em seguida, coloca o processador no modo enclave e guarda os registos RSP/RBP e XCR0. Por fim, desativa o PEBS (*Precise Event Based Sampling*) para fazer com que a execução do enclave pareça como uma instrução gigante.

A instrução EEXIT coloca o processo novamente no seu modo original e limpa as entradas TLB dos endereços localizados no enclave. O controlo é transferido para o endereço localizado na aplicação e especificado no registo RBX, e a estrutura do TCS é liberada. O enclave precisa de limpar os seus registos antes de sair para evitar o vazamento de dados.

Para dar entrada no enclave:

1. A instrução EENTRY é executada;
2. O contexto da aplicação é gravado;
3. O processador é colocado no modo enclave.

Para sair do enclave:

1. A instrução EEXIT é executada;
2. O processador é colocado no modo normal;

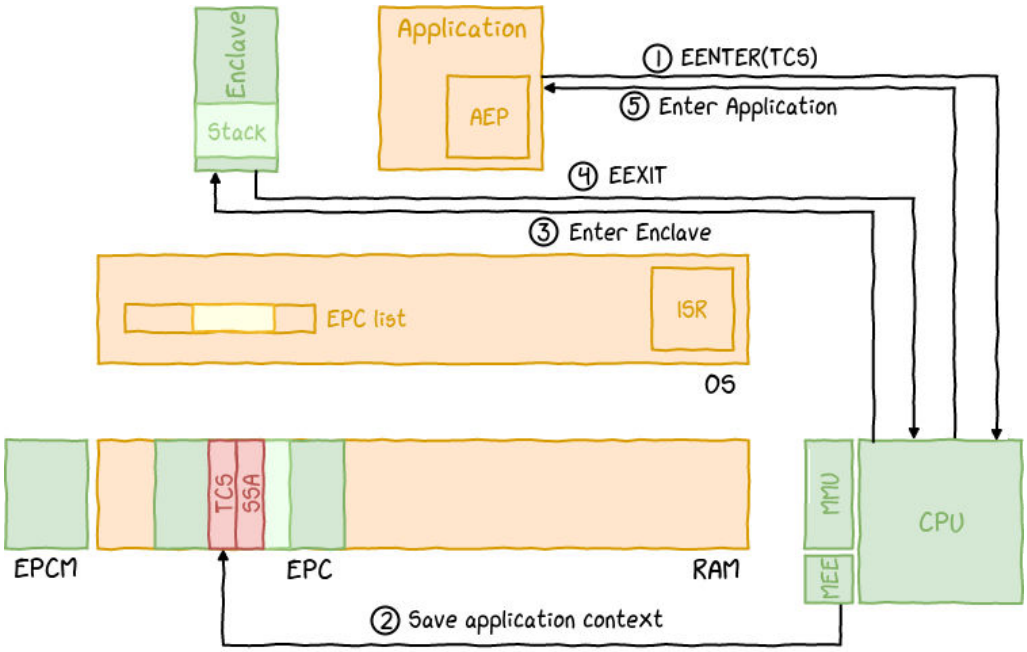


Figura 2.10: Ciclo de vida de um enclave (retirada de [1])

A instrução ERESUME restaura o contexto do SSA atual e retoma a execução.

Interrupções e exceções resultam em saídas assíncronas de enclave (AEX). O apontador de saída assíncrona (AEP) aponta para um manipulador localizado dentro da aplicação que retoma a execução após a exceção ter sido tratada pelo ISR (*Interrupt Service Routine*). O manipulador pode decidir retomar ou não a execução do enclave, executando a instrução ERESUME.

Quando uma AEX acontece, o contexto do enclave é guardado no SSA atual e o contexto da aplicação é restaurado. O contexto do enclave é restaurado quando a instrução

ERESUME é executada. O TCS contém um contador que indica o SSA atual, formando uma pilha de contextos.

O tratamento de exceções/interrupções ocorre da seguinte forma:

1. A interrupção ou exceção chega ao processador;
2. O contexto do enclave é guardado e o contexto da aplicação é restaurado;
3. A execução continua no manipulador do sistema operativo;
4. O manipulador retorna (IRET) ao AEP, uma função de trampolim;
5. O AEP executa ERESUME se decidir retomar a execução do enclave;
6. O contexto do enclave guardado anteriormente é restaurado;
7. A execução é retomada onde parou no enclave.

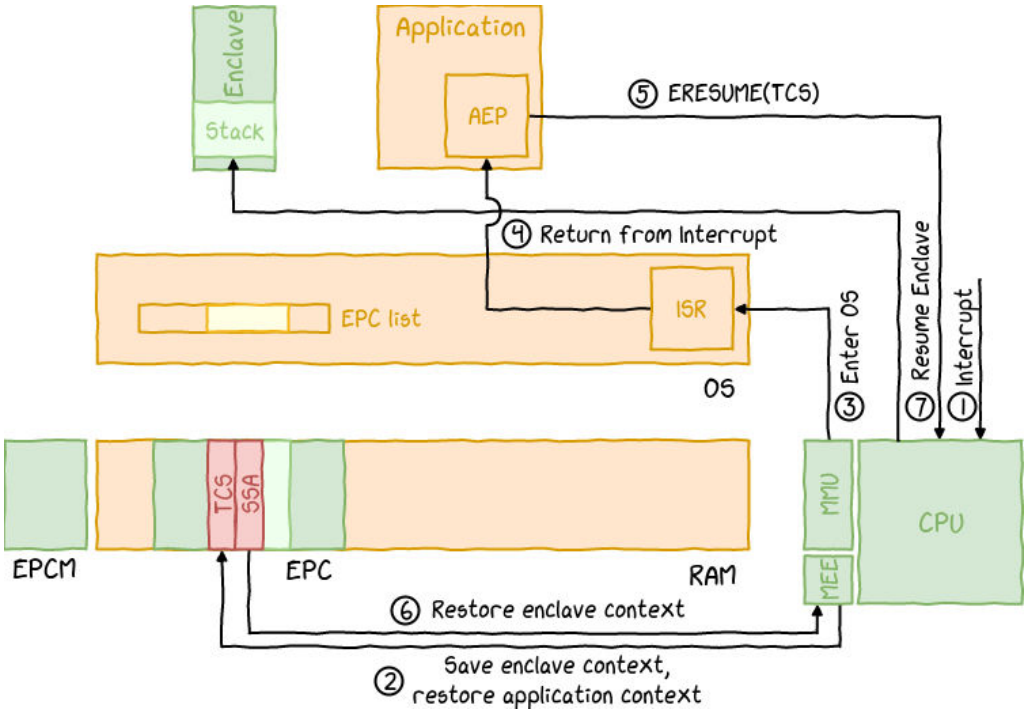


Figura 2.11: Tratamento de exceções no enclave (retirada de [1])

A instrução EGETKEY é usada por um enclave para aceder às diferentes chaves fornecidas pela plataforma. Cada chave permite uma operação diferente (selagem, atestação). Quando um enclave é instanciado, o seu código e dados são protegidos contra acessos externos. Mas quando para, todos os seus dados são perdidos. A selagem é uma maneira de salvar com segurança os dados fora de um enclave num disco rígido, por exemplo. O enclave deve recuperar a sua chave de selagem usando a instrução EGETKEY. Este usa esta chave para encriptar e garantir a integridade dos dados. O algoritmo usado é escolhido pelo autor do enclave.

A selagem pode ser feita usando a identidade do enclave. A derivação de chave é então baseada no valor de MRENCLAVE. Dois enclaves distintos têm chaves diferentes, mas também duas versões do mesmo enclave, o que impede a migração local de dados.

A selagem também pode ser feita usando a identidade do assinante. A derivação de chave é então baseada no valor de MRSIGNER. Dois enclaves distintos ainda têm chaves diferentes, mas duas versões de um enclave partilham a mesma chave e podem ler os dados selados. Se múltiplos enclaves forem assinados usando as mesmas chaves, todos poderão ler os dados uns dos outros.

As versões mais antigas de um enclave não devem ter permissão para ler dados selados por uma versão mais recente de um enclave. Para evitá-lo, o SVN é usado. Este é um contador incrementado após cada atualização que afeta a segurança do enclave. As chaves são derivadas usando o SVN de forma a que um enclave possa recuperar as chaves correspondentes ao nível de segurança atual ou mais antigo, mas não mais recente.

A estrutura KEYREQUEST é usada como uma entrada para a instrução EGETKEY. Esta permite escolher que chave obter e também parâmetros adicionais que podem ser necessários para a derivação. A estrutura TARGETINFO é usada como uma entrada para a instrução EREPORT. É usada para identificar qual enclave (*hash* e atributos) pode verificar o REPORT gerado pela CPU. Report (REPORT) A estrutura REPORT é a saída da instrução EREPORT. Esta contém os atributos do enclave, a medida, a identidade do assinante e alguns dados do utilizador para partilhar entre os enclaves de origem e de destino. O processador executa um MAC nesta estrutura usando a Chave de Relatório.

A instrução EREPORT é usada pelo enclave para gerar uma estrutura REPORT contendo várias informações sobre ele e autenticada usando a Chave de Relatório do enclave de destino. O código e os dados do enclave estão *plaintext* antes da sua inicialização. Embora as seções possam ser tecnicamente encriptadas, a chave para desencriptar não pode ser pré-instalada (ou não forneceria nenhuma segurança adicional). Os segredos têm que vir de fora, podem ser chaves e dados confidenciais. O enclave deve ser capaz de provar a terceiros que pode ser confiável (não foi adulterado) e está a executar numa plataforma legítima.

Existem dois tipos de atestação:

- atestação local: um processo de atestação entre dois enclaves da mesma plataforma;
- atestação remota: um processo de atestação entre um enclave e um terceiro que não está na plataforma.

Quanto à atestação local:

1. Um canal já deve ter sido estabelecido entre o enclave A e o enclave B. É usado pelo enclave A para recuperar o MRENCLAVE de B.
2. O enclave A chama EREPORT com o MRENCLAVE de B para gerar um relatório assinado para o último.

3. O Enclave B chama EGETKEY para recuperara sua Chave de Relatório e verificar o MAC da estrutura EREPORT. Se válido, o enclave é o esperado e está a executar numa plataforma legítima.

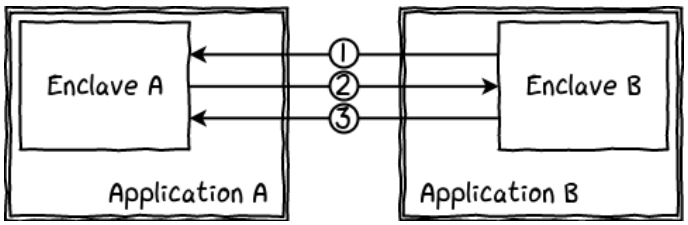


Figura 2.12: Atestação Local por SGX (retirada de [1])

No que diz respeito à atestação remota, esta requer um enclave arquitetural chamado *Quoting Enclave*. Este enclave verifica e transforma o REPORT (verificável localmente) num QUOTE (verificável remotamente) assinando-o com outra chave especial, a Chave de Provisionamento. Eis os passos deste processo:

1. Inicialmente, o enclave informa a aplicação que precisa de um segredo localizado fora da plataforma. A aplicação estabelece uma comunicação segura com um servidor. O servidor responde com um desafio para provar que a execução do enclave não foi violada e que a plataforma na qual executa é legítima;

2. A aplicação fornece a identidade do *Quoting Enclave* e o desafio ao seu enclave;

3. O enclave gera um manifesto, incluindo a resposta ao desafio e uma chave pública efémera que será usada posteriormente para proteger as comunicações entre o servidor e o enclave. Este gera um *hash* do manifesto que inclui na seção de dados do utilizador da instrução EREPORT. A instrução gera um RELATÓRIO para o *Quoting Enclave* que vincula o manifesto ao enclave. O enclave passa o RELATÓRIO para a aplicação.

4. A aplicação envia o QUOTE e o manifesto associado ao servidor para verificação.

5. O servidor usa o serviço de atestação fornecido pela Intel para validar a assinatura do QUOTE. Em seguida, verifica a integridade do manifesto usando o *hash* dos dados do utilizador no QUOTE. Por fim, garante que o manifesto contem a resposta esperada para o desafio.

2.4.2 Parte Externa das Intel SGX

Conceptualmente, um enclave pode ser visto como uma caixa negra capaz de executar qualquer algoritmo arbitrário. Esta caixa pode comunicar com o mundo exterior usando três maneiras diferentes.

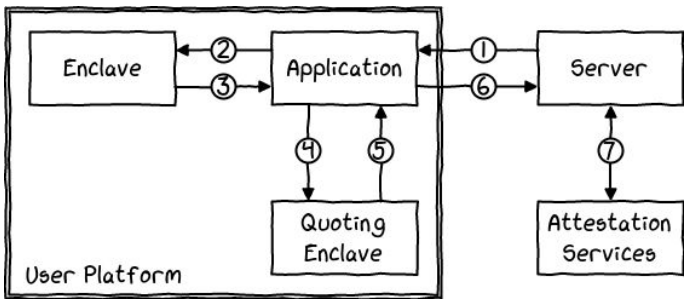


Figura 2.13: Atestação Remota por SGX (retirada de [1])

- **Chamadas do Enclave (ECALLs):** A aplicação pode chamar uma função predefinida dentro do enclave, passando parâmetros de entrada e apontadores para a memória partilhada dentro da aplicação. Estas invocações da aplicação para o enclave são chamadas ECALL.
- **Chamadas externas (OCALLs):** Quando um enclave é executado, pode executar uma OCALL numa função predefinida na aplicação. Ao contrário de uma ECALL, uma OCALL não pode partilhar a memória do enclave com a aplicação, portanto, deve copiar os parâmetros na memória da aplicação antes da OCALL.
- **Saída assíncrona (AEX):** A execução também pode sair de um enclave devido a uma interrupção ou exceção. Estes eventos são chamados de eventos de saída assíncrona (AEX) e podem transferir o controlo do enclave para a aplicação a partir de pontos arbitrários dentro do enclave.

O desenvolvimento de uma aplicação que usa um enclave requer a identificação dos recursos a serem protegidos, a estrutura de dados que contém esses recursos e o código que os gere. Tudo o que for identificado deve ser colocado dentro do enclave. Um arquivo de enclave é uma biblioteca compatível com os *loaders* de sistemas operacionais tradicionais. Contém o código e os dados do enclave, em *plaintext* no disco.

A interface entre a aplicação e o seu enclave deve ser desenhada com cuidado. Um enclave declara que funções podem ser chamadas pela aplicação e que funções da aplicação pode chamar. Os parâmetros de entrada do enclave podem ser observados e modificados pelo código não seguro, portanto, devem ser verificados extensivamente. Como um enclave não pode aceder diretamente aos serviços do sistema operativo, deve chamar a sua aplicação. Estas chamadas não devem expor nenhuma informação confidencial e não é garantido que executem conforme o esperado pelo enclave.

O Intel SGX SDK (*Software Development Kit*) fornece aos desenvolvedores tudo o que precisam para desenvolver um aplicação habilitada para SGX. É composto por uma ferramenta para gerar as funções de interface entre a aplicação e o enclave, uma ferramenta para assinar o enclave antes de o usar, uma ferramenta para o depurar e uma última para medir o desempenho. Também contém modelos e projetos amostra para desenvolver um enclave usando o *Visual Studio* no *Windows* ou *Makefiles* no *Linux*.

O *Software* de Plataforma (PSW) constitui a pilha de *software* que permite que aplicações habilitadas para SGX sejam executadas na plataforma alvo. Está disponível nos sistemas operativos *Windows* e *Linux* e é composto por 4 partes principais:

- o *driver* que fornece acesso aos recursos de *hardware*;
- várias bibliotecas de suporte para execução e atestação;
- os enclaves arquitetónicos necessários para a execução do ambiente;
- um serviço para carregar e comunicar com os enclaves.

Para permitir a execução do ambiente seguro, são necessários vários Enclaves arquitetónicos (AE) fornecidos e assinados pela Intel. Estes enclaves aplicam políticas de inicialização, executam os processos de provisionamento ,atestação e muito mais.

O Enclave de Inicialização (LE) é o enclave responsável pela distribuição de estruturas EINITTOKEN a outros enclaves que desejam executar na plataforma. Este verifica a validade da assinatura e da identidade do enclave. Para gerar os *tokens*, utiliza a *Launch Key* que apenas ele é enclave capaz de obter.

O Enclave de provisionamento (PvE) é o enclave responsável pela recuperação da Chave de Atestação, comunicando com os servidores do Serviço de Provisionamento da Intel. Para isso, comprova a autenticidade da plataforma usando um certificado fornecido pelo PcE.

O Enclave de provisionamento do certificado (PcE) é o enclave responsável por assinar o certificado do processador destinado ao PvE. Para o fazer, usa a chave de provisionamento que apenas ele é enclave capaz de obter. O PvE e o PcE são atualmente implementados como um único enclave.

O *Quoting Enclave* (QE) é o enclave responsável por fornecer confiança na identidade de um enclave e no ambiente em que é executado durante o processo de atestação remota. Desencripta a Chave de Atestação que recebe do PvE e usa-a para transformar uma estrutura REPORT (verificável localmente) numa estrutura QUOTE (verificável remotamente).

Os Enclaves de Serviço de Plataforma (PSE) são enclaves de arquitetónicos que oferecem a outros enclaves vários serviços, como contadores monótonos, tempo confiável, etc. Estes enclaves fazem uso do Motor de Gestão (ME), um mecanismo isolado e suposto coprocessador seguro que gere a plataforma.

A primeira chave criada pela Intel durante o processo de fabricação é a chave de provisionamento raiz (RPK). Esta chave é gerada aleatoriamente num Módulo de Segurança de Hardware (HSM) localizado dentro de uma instalação chamada Intel *Key Generation Facility* (iKGF). A Intel é responsável por manter uma base de dados que contem todas as chaves produzidas pelo HSM. As RPKs são enviadas para várias instalações de produção para serem incorporadas nos fusíveis electrónicos dos processadores.

A segunda chave localizada dentro dos fusíveis electrónicos é denominada de chave de selagem raiz (RSK). Tal como a primeira chave, é garantido que difere estatisticamente

entre cada unidade produzida. Ao contrário da RPK, a Intel declara que apaga todos os vestígios dessas chaves da sua cadeia de produção, para que cada plataforma tenha uma chave única conhecida exclusivamente pela própria.

Por design, um enclave não tem acesso às chaves raiz. No entanto, pode aceder a chaves derivadas das chaves raiz. A função de derivação permite que um autor de um enclave selecione uma política de derivação de chaves. Essas políticas permitem o uso de valores confiáveis, como o MRENCLAVE, o MRSIGNER e/ou os atributos do enclave. Os enclaves não podem derivar chaves pertencentes a um MRENCLAVE ou MRSIGNER de outro enclave. Além disso, quando a política de derivação da chave não faz uso de um campo é automaticamente definida como zero. Como resultado, mesmo quando estão disponíveis chaves não especializadas, destas não podem ser derivadas chaves especializadas.

Para adicionar entropia proveniente do utilizador, um valor chamado *Owner Epoch* é usado como parâmetro durante a derivação. Este valor é configurado em *boot-time* pela derivação de uma senha e salvo durante cada ciclo de energia numa memória não volátil. Este valor deve permanecer o mesmo para que um enclave possa recuperar as mesmas chaves. Pelo contrário, este valor deve ser alterado quando o proprietário da plataforma muda, pois impede que o novo proprietário aceda às informações pessoais do antigo proprietário até que a senha original seja restaurada.

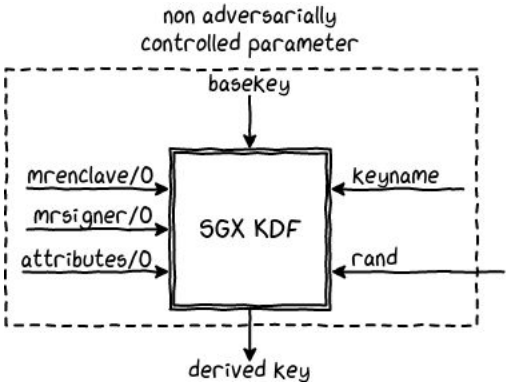


Figura 2.14: Derivação de chaves nas Intel SGX (retirada de [2])

A infraestrutura das SGX suporta atualizações da TCB dos seus componentes de *hardware* e *software*. Cada componente possui um SVN que é incrementado após cada atualização de segurança. Um novo SVN leva a uma nova Chave de Selagem. Existe um processo que permite que um TCB mais recente aceda às Chaves de Selagem do TCB mais antigo, para permitir a migração de dados. Um TCB antigo não pode aceder as chaves de um novo TCB.

A Chave de Provisionamento é derivada da RPK e usada como raiz de confiança (vinculada à versão da TCB) entre o Serviço de Provisionamento da Intel e o processador. Como a admissão de um processador não SGX num grupo de processadores SGX legítimos compromete a atestação remota para todos os processadores, devem ser tomadas precauções extremas para impedir o acesso à Chave de Provisionamento. Atualmente, o

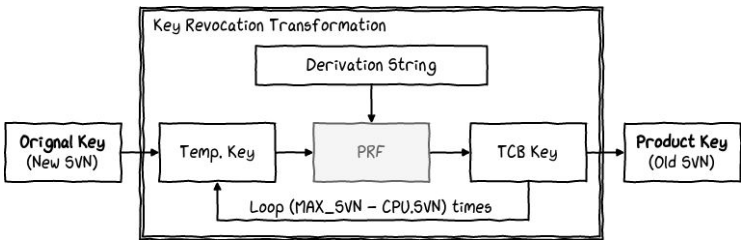


Figura 2.15: Recuperação de chaves nas Intel SGX (Key Recovery Transformation) (retirada de [2])

Enclave de Inicialização dá acesso a essa chave somente se o enclave for assinado pela Intel (o MRSIGNER da Intel é *hard coded* no código do Launch Enclave).

A Chave de Selagem de Provisionamento é derivada da RPK e da RSK. Durante o registo de um processador no grupo, a chave privada de cada plataforma é encriptada com esta chave e enviada ao Serviço de Atestação Intel. Deve observar-se que a chave privada não pode ser encriptada apenas usando o RPK, pois isso destruiria o protocolo de registo anónimo usado. Da mesma forma, a chave privada não pode ser encriptada apenas usando o RSK, pois permitiria que enclaves não privilegiados acedessem à chave privada da plataforma. Infelizmente, conhecida a incerteza existente no processo de geração do RSK, pode-se supor que a Intel conhece a chave privada de cada plataforma.

A Chave de Inicialização é derivada da RSK e é usada pelo Enclave de Inicialização para gerar um EINITTOKEN. Cada enclave que não é assinado pela Intel deve obter este *token*, caso contrário, o processador não poderá instanciá-lo. Somente um MRSIGNER específico, cujas chaves privadas correspondentes são conhecidas apenas pela Intel, pode aceder à Chave de Inicialização. Na versão dois das SGX, o MRSIGNER do Enclave de Lançamento pode ser alterado programaticamente, mas ainda não se sabe como a Intel pretende aplicar o controlo de acesso à Chave de Provisionamento.

A Chave de Selagem é derivada da RSK e usada para encriptar dados relacionados à plataforma atual. É importante não usar uma Chave de Selagem não especializada para criptografia ou autenticação, porque isso comprometeria a segurança do enclave.

A Chave de Relatório chave é derivada da RSK e é usada para o processo de atestação local.

A primeira preocupação expressa pelos utilizadores das SGX é que devem confiar na Intel. Notavelmente, a Intel diz que não retém a RSK incorporado em cada processador construído nas suas instalações de fabrico. Porém, se a retivesse, de qualquer maneira possível, isso invalidaria a segurança de todas as plataformas. Além disso, os enclaves assinados pela Intel recebem privilégios especiais, como o Enclave de Inicialização, que é usado para colocar numa lista que enclaves têm permissão para executar. Os desenvolvedores precisam de se registar nos programas da Intel para poder assinar a versão de lançamento dos enclaves.

A segunda preocupação é que seria possível um *malware* executar o seu código malicioso dentro de um enclave, protegendo-se de tudo e de todos. No entanto, é importante

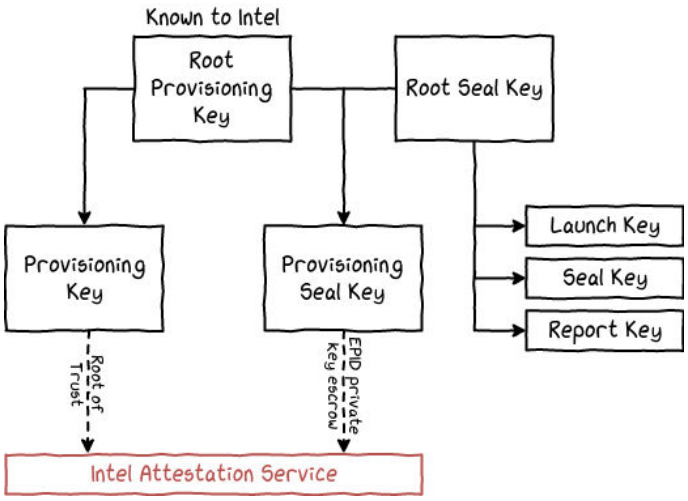


Figura 2.16: Vista geral das chaves das Intel SGX (retirada de [2])

ter em mente que o código dentro de um enclave não possui *input/output*, baseando-se unicamente na aplicação que o acompanha para aceder à rede, ao sistema de ficheiros, etc. Portanto, tecnicamente, a análise de uma aplicação pode dizer muito sobre o que um enclave pode fazer com um sistema, atenuando o medo de um "código malicioso protegido a executar dentro de um enclave". Por outro lado, a falta de *input/output* confiável é um problema para proteger as informações do utilizador e já foi realizado algum trabalho sobre este assunto, com soluções proprietárias como o *Protected Audio Video Path* (PAVP) [24] e académico como o SGXIO [39].

A terceira preocupação é com o impacto do *Meltdown* [32] e do *Spectre* [29] nos enclaves das SGX. Embora não sejam vulneráveis ao primeiro, os autores do artigo *SgxPectre*[11] demonstraram que variantes do *Spectre* permitiam ler a memória do enclave e registar valores. Isto permitiu a recuperação da Chave de Selagem da plataforma e, consequentemente, da Chave do Atestação, trespassando efetivamente toda a segurança oferecida pelas SGX. A Intel lançou uma atualização de microcódigo para evitar estes ataques e, graças ao Número da Versão de Segurança (SVN), também é capaz de garantir que estes *patches* tenham sido aplicados para passar no processo de atestação remota. Um outro ataque chamado *SpectreRSB* [30] surgiu em 2018 e era capaz de trespassar os *patches* tendo como alvo o RSB (*Return Stack Buffer*) em vez do *Branch Target Buffer*. A Intel lançou outra atualização de microcódigo para corrigir este novo problema. Mais recentemente, no final do ano passado, um novo ataque denominado *Plundervolt* [35] aproveitava-se da verificação inadequada das condições nas configurações de tensão de alguns processadores Intel, o que podia permitir a um utilizador privilegiado ativar potencialmente o escalar de privilégios e/ou a divulgação de informações por meio de acesso local.

As Intel SGX são uma tecnologia promissora que ainda está na sua infância, embora atenda às necessidades da indústria da Computação Confiável, permitindo que um enclave seja protegido contra os outros *softwares* em execução na plataforma e, de maneiras

limitadas, contra a adulteração intensa. Isto é feito ao fornecer ao proprietário da plataforma algum grau de controlo sobre a execução para permitir a gestão de recursos. O processo de atestação permite que os segredos sejam transmitidos com segurança ao enclave. O desenvolvimento de uma aplicação habilitada para SGX é facilitado graças ao SDK fornecido.

No entanto, ainda existem alguns problemas com a iteração atual do Intel SGX. A propensão a ataques de *side-channel*, infelizmente, limita a segurança oferecida pela plataforma, forçando os desenvolvedores a garantir proativamente que os seus programas não possam ser atacados. Num mundo perfeito, os desenvolvedores não precisariam de se preocupar com estas considerações e as SGX deviam garantir que estes ataques eram impossíveis. Todos estes ataques podem ser evitados encriptando o código do enclave e provando a chave por atestação remota, mas isto é um inconveniente. Por fim, os utilizadores precisam de confiar muito na Intel e, embora tenham demonstrado uma resposta perfeita a incidentes até o momento, ainda existem incertezas.

Capítulo 3

Desenho do Sistema

No decorrer deste capítulo, são descritos detalhadamente os alicerces que suportam o sistema proposto desde as várias entidades que o compõe até à forma como interagem entre si. Além disto, são também justificados os dados embutidos em cada uma das interações e o papel desempenhado pelas entidades estabelecidas. O objetivo é demonstrar que aspetos foram considerados para alavancar o sistema pensado, ou seja, que propriedades de segurança, autenticação e criptografia são as mais adequadas para resolver o problema em questão. Este capítulo serve, portanto, como base fundamentalmente teórica para o que foi realmente realizado (capítulo 4).

3.1 Arquitetura

O propósito principal da arquitetura é assegurar que qualquer cliente que está integrado no sistema executa uma versão de um código que é aceite pela entidade central que fornece o serviço de transações. Além disto, é crucial garantir que a máquina de qualquer cliente é legítima. Para este efeito, é vital que a entidade central verifique o código de cada cliente que adere ao serviço e que suporte um procedimento de autenticação que garanta a integridade da sua máquina. Assim, existem três operações que formam a base desta arquitetura: autenticação da máquina de um cliente, atestação do código que se executa na máquina de um cliente e transferência de transações entre clientes.

O sistema desenhado é composto por uma rede *peer-to-peer* constituída por nós portadores de um processador Intel com a tecnologia Intel SGX, um Provedor de Serviços e o Serviço de Atestação Intel. De grosso modo, os clientes fornecem provas de autenticação ao servidor SP e este por sua vez encaminha-as ao IAS que verifica essas evidências usando os dados dos processadores mantidos pela empresa. Embora seja o IAS que determine

que enclaves são criados numa plataforma SGX genuína, aplicando o esquema de assinatura em grupo Intel EPID, cabe em última instância ao provedor de serviços aplicar uma política de aceitação para decidir que enclaves de que clientes são permitidos pertencer ao sistema. O intuito central deste mecanismo de autenticação é afirmar que qualquer cliente aprovado pelo SP corre o código do enclave que foi desenvolvido para os seus clientes. No fim deste processo de atestação, os clientes obtêm um "passaporte" que usam para autenticar as suas transações, enviadas diretamente para os outros intervenientes na rede, de modo a provarem que foram aceites como fidedignos pelo SP.

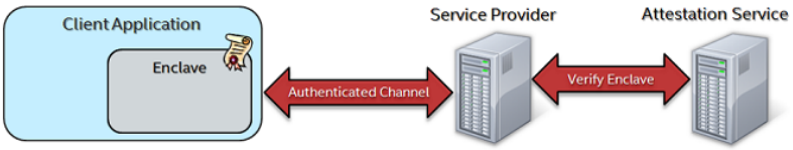


Figura 3.1: As três componentes do sistema (clientes + enclaves, SP e IAS)

A aplicação dos clientes SGX está dividida em duas partes, seguindo o modelo de programação das Intel SGX, portanto é composta por uma parte segura (enclave) e uma parte não segura. Dado que os enclaves não possuem *input/output*, é da responsabilidade da parte não segura comunicar com as outras entidades do sistema (o SP e os outros nós). Com isto, o dever do enclave é manter secreto tudo o que envolve as transações e os procedimentos da atestação remota. Seguindo também a metodologia de seleção de que segredos o enclave protege, os seguintes dados são os que o enclave de cada cliente guarda:

- a chave pública de curvas elípticas do SP;
- o saldo da carteira de moedas;
- um par de chaves (pública e privada) de curvas elípticas;
- uma estrutura de dados de *nonces* para anti-repetição de transações;
- duas chaves de sessão resultantes do processo de atestação perante o SP.

A chave pública do SP deve ser *hard coded* no enclave, pois combinado com a assinatura do enclave, garante que a chave não pode ser alterada pelos utilizadores finais para que o enclave apenas comunique com o SP pretendido. O saldo do cliente é mantido dentro do enclave para evitar o *double-spending* de uma moeda, no entanto, isto apenas garante que a moeda não é gasta mais que uma vez localmente, porque quando uma transação é enviada pela rede pode ser replicada por um atacante capaz de a copiar do canal de comunicação. Para resolver esta situação, o enclave armazena como complemento ao saldo da carteira uma estrutura de dados de *nonces* que armazena os últimos *nonces* das transações enviadas pelos outros nós. Desta forma, conseguimos assegurar de forma total que uma moeda não é efetivamente gasta duas vezes. Por fim, cada enclave necessita de

um par de chaves de curvas elípticas para poder assinar transações e provar aos outros nós a autenticidade e integridade das mesmas.

O SP é o servidor que hospeda um jogo, cujos jogadores são aqueles na rede *peer-to-peer*, e que pretende criar uma moeda virtual como base para todas as transações que acontecem no jogo. Este servidor serve de ponte de comunicação entre os seus jogadores e o IAS para os poder autenticar como genuinamente habilitados para SGX e como utilizadores que executam o código genuíno responsável pelas transações do jogo (o código do enclave). É esta a entidade que usa o valor MRENCLAVE dos enclaves dos clientes que se tentam registar para averiguar se têm o código do enclave autêntico criado e assinado pelo desenvolvedor.

O IAS é a entidade que assegura que cada membro da rede possui um ambiente de execução confiável, ou seja, o membro detém uma assinatura EPID que foi verificada corretamente e o nível TCB da sua máquina SGX está atualizada.

Quando o cliente se autentica no sistema e transaciona na rede o estado do seu enclave vai progredindo. Inicialmente, este estado possuiu apenas o saldo, a chave pública do SP, a estrutura de anti-repetição vazia e um par de chaves para autenticação de transações. Após o primeiro passo do protocolo de autenticação, onde o cliente computa o seu segredo da troca de chaves e o envia ao SP, o estado do seu enclave é atualizado. Quando recebe uma resposta do SP com o seu parâmetro da troca de chaves o mesmo acontece pois, são geradas as chaves de sessão e alocado o *quote* que representa a sua evidência de atestação. A partir daqui, sempre que o cliente recebe uma transação o seu estado altera-se, pois o seu saldo varia e a estrutura para anti-repetição é atualizada conforme os seus dados.

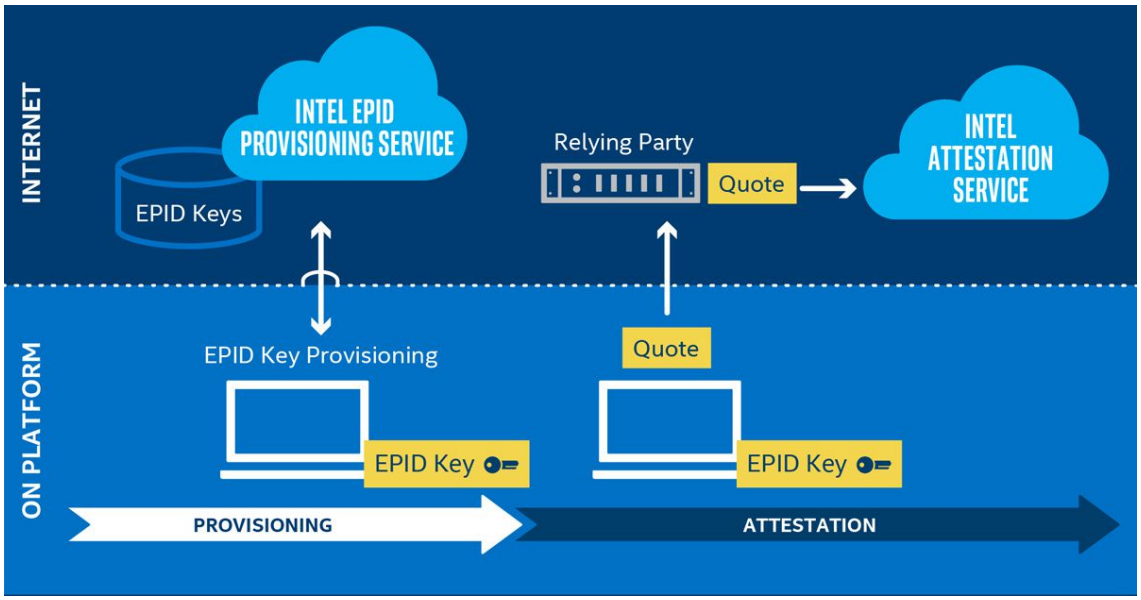


Figura 3.2: Protocolo de Autenticação em alto nível retirada de [27])

3.2 Protocolo de autenticação

O objetivo do protocolo de autenticação é permitir que o SP controle as entidades dos enclaves dos clientes da rede *peer-to-peer*. Para este efeito, o SP estipula uma política para estabelecer que assinatura e código do enclave os clientes têm de possuir para poderem integrar o sistema. O protocolo também permite assegurar, através do IAS, que as máquinas dos clientes são válidas e que os seus processadores não estão comprometidos.

Quanto ao mecanismo de autenticação existem duas alternativas: Atestação Remota com base no Intel EPID ou Atestação Baseada no Intel SGX DCAP [37]. A primeira tecnologia permite que um terceiro confiável ateste um enclave sem conhecer o processador Intel específico no qual o enclave está a executar. O uso desta tecnologia requer que uma plataforma e o terceiro confiável tenham acesso à Internet. Este serviço de atestação *online* é criado e gerido pela Intel para minimizar a complexidade de lidar com várias versões de segurança de uma plataforma com uma base de computação confiável Intel SGX e para fornecer propriedades de privacidade. A segunda forma de atestação é baseada em ECDSA e permite que provedores de serviços criem e forneçam o seu próprio serviço de atestação, em vez de usar o serviço de atestação remoto fornecido pela Intel. Isto é útil para fornecedores de serviços corporativos, de centros de dados e de nuvem que precisam atender a qualquer um dos seguintes requisitos:

- Executar grandes partes das suas redes em ambientes onde os serviços baseados na Internet não podem ser alcançados;
- Manter internamente as decisões de atestação;
- Entregar aplicações que funcionam de maneira muito distribuída (por exemplo, redes *peer-to-peer*) que beneficiam ao não depender de um único ponto de verificação;
- Impedir o anonimato da plataforma onde não é permitido;

Tendo em conta que o SP e os clientes têm acesso à Internet, existe um serviço de autenticação completamente controlado pela Intel e se se quer unicamente averiguar se os enclaves são ou não fidedignos, a primeira opção é realmente a mais acertada e cómoda. Para além disso, como a gestão das diferentes versões de segurança é gerida também pela Intel é possível apurar com maior precisão as entidades dos enclaves que se atestam, principalmente quanto às suas vulnerabilidades de segurança.

Para o SP poder autenticar os seus clientes, comunicando com o Serviço de Atestação Intel SGX EPID, são obtidas previamente chaves de API do serviço que são colocadas *hard coded* no código do SP. A política de atestação escolhida é do tipo vinculável o que permite determinar se várias solicitações de atestação foram originadas da mesma plataforma. Esta política não identifica a plataforma individual, mas possibilita afirmar se várias solicitações sucederam da mesma plataforma. Para além das chaves API, é obtido também um SPID usado pelo SP ao comunicar com o IAS.

O processo pelo qual os jogadores do sistema se registam no servidor do jogo constitui um protocolo cliente-servidor de atestação remota que utiliza um protocolo Sigma modificado para facilitar uma DHKE entre o cliente e o SP. A chave partilhada obtida desta troca é usada pelo SP para encriptar um certificado a ser fornecido ao cliente. O enclave do cliente pode derivar a mesma chave e usá-la para desencriptar o certificado. Este certificado assinado pelo SP é devolvido apenas se o processo de autenticação for bem sucedido (o IAS reportou um enclave autêntico) e contem a chave pública de curvas elípticas do enclave do cliente que se regista. A criação deste conteúdo digital é motivada pela necessidade de um emissor de uma transação ter que provar não só que executa num processador SGX credível, mas também que corre exatamente o código do enclave elaborado e assinado pelo desenvolvedor/autor pretendido ao respetivo recetor da transação. Como o SP apenas fornece certificados a clientes cujo o IAS afirmou serem legítimos, sempre que um jogador pretender transacionar basta anexá-lo à transação para demonstrar que é um jogador autenticado pelo servidor do jogo.

O esquema de autenticação descrito acima está particionado em quatro mensagens principais, porém supõe-se que os membros da rede efetuam previamente um primeiro pedido no formato *challenge-response* cujo comportamento é deixado ao critério do próprio servidor do jogo. Somente após o membro responder com sucesso ao desafio é que o fluxo de atestação pode começar. O fluxo completo de atestação que é de seguida explicado está apresentado na figura 3.3.

O primeiro passo que o cliente realiza é a inicialização do seu enclave para se construir a primeira mensagem (msg0). Para a criar é efetuada uma chamada ao enclave (ECALL), onde é inicializado e depois devolvido à parte não segura da aplicação, o contexto de atestação remota e de troca de chaves utilizado em várias operações no enclave para suportar o mecanismo de atestação. A fim de obter este contexto, é necessário fornecer a chave pública de curvas elípticas do SP que está *hard coded* no código do enclave. O contexto retornado é opaco ao cliente para proteção contra *replay*. Após isto, é requerido ao enclave o identificador estendido do grupo EPID (ExGID) que será o conteúdo da primeira mensagem. Quando o SP receber esta mensagem, deve verificar se este valor é suportado e abortar o processo de atestação caso contrário, porque o IAS atualmente suporta apenas o valor zero para o ExGID. A primeira mensagem é enviada concatenada com a segunda por razões de eficiência, eliminando uma ida e volta adicional entre o cliente e o servidor. A suposição aqui é que a maioria dos clientes possui o ExGID correto, portanto é um desperdício enviar a primeira mensagem separadamente quando a probabilidade de uma rejeição é astronomicamente pequena. Se for rejeitado, o cliente gasta apenas uma pequena quantidade de tempo para gerar chaves que não são usadas.

Posto que a segunda mensagem (msg1) é enviada juntamente com a primeira, o cliente cria de seguida a segunda mensagem. A segunda mensagem é gerada usando o contexto da atestação opaco obtido anteriormente e a funcionalidade inerente do enclave para calcular o segredo do lado do cliente. Esta mensagem contém a chave pública de curvas

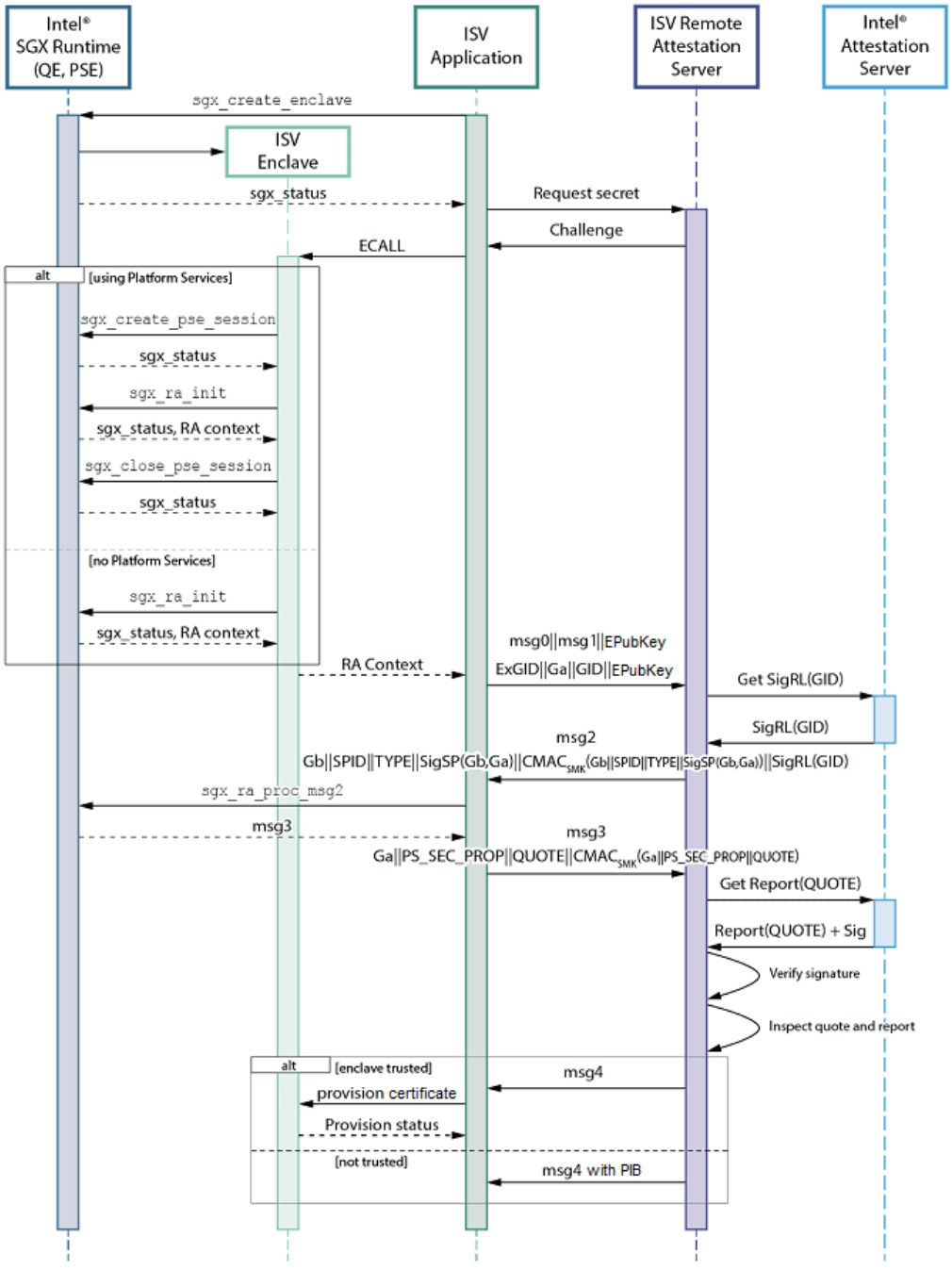


Figura 3.3: Diagrama do fluxo de atestação

elípticas do enclave do cliente (Ga) para a DHKE, com base na curva elíptica NIST P-256 [17], e o ID do grupo Intel EPID (GID) a que a plataforma pertence. Para além de tudo isto, é também anexada a chave pública de curvas elípticas do enclave (EPubKey).

Na tabela 3.1, onde || significa concatenação, são mostrados todos os dados que primeiramente seguem para o SP.

Tabela 3.1: Dados das primeiras duas mensagens

Mensagem	**Conteúdo**
msg0	ExGID
msg1	Ga || GID
msg0 || msg1 || EPubKey	ExGID || Ga || GID || EPubKey

Ao receber as duas primeiras mensagem de um cliente, o SP verifica o valor do ExGID, e como já foi dito anteriormente, aborta a atestação se o respetivo valor não for zero. Após esta verificação, o servidor gera o seu próprio parâmetro da DHKE e envia uma consulta ao IAS para adquirir a lista de revogação de assinaturas (SigRL) do GID recebido. O servidor cria a terceira mensagem do fluxo de atestação (msg2), como resposta à do cliente, gerando em primeiro lugar uma chave de curvas elípticas aleatória (Gb) usando a curva NIST P-256. Com esta chave e com a chave Ga do cliente, o SP deriva a chave de derivação de chaves (KDK). O servidor ao calcular o segredo partilhado usando a chave de sessão pública do cliente, Ga, e a chave Gb, obtém a coordenada x de Gab (Gab_x), que ao passar por um AES-128 CMAC cuja chave são 16 *bytes* com o valor zero, origina a KDK que é protegida em memória para impedir a sua inspeção trivial. Por fim, é efetuada uma última derivação que origina a chave do protocolo Sigma (SMK), fazendo novamente um AES-128 CMAC mas com a chave KDK, na seguinte sequência de *bytes*: 0x01 || "SMK" || 0x00 || 0x80 || 0x00, com "SMK" uma *string* sem as aspas.

Após efetuar os cálculos descritos acima, o SP constrói a terceira mensagem com o tipo de *quote* (Quote_Type) requerido ao cliente, que é vinculável como definido pela política do servidor, o identificador da função de derivação de chaves (KDF_ID), que neste caso possui o valor 0x1, a assinatura ECDSA (SigSP) da sequência r || s (r = Gb_x || Gb_y e s = Ga_x || Ga_y) com a chave privada de curvas elípticas do SP, o AES-128 CMAC de (Gb || SPID || Quote_Type || KDF_ID || SigSP) usando a chave SMK e por último a lista de revogação de assinaturas (SigRL) obtida através do IAS. Na tabela 3.2, considerando A = Gb || SPID || Quote_Type || KDF_ID || SigSP, são mostrados todos os dados que incluem a msg2 enviada para o cliente.

Tabela 3.2: Dados da terceira mensagem

A || $CMAC_{SMK}$(A) || SigRL

Aquando da receção da msg2, o cliente verifica a assinatura do SP, a lista de revogação

de assinaturas e constrói a quarta mensagem (msg3) que contém o *quote* usado para atestar o seu enclave. O *quote* inclui um *hash* criptográfico, ou medição, do enclave em execução assinado com a chave EPID da plataforma e somente o IAS consegue verificar esta assinatura. Na tabela 3.3, considerando M = Ga || *Ps_Security_Prop* || Quote, são mostrados todos os dados que incluem a msg3 enviada para o SP. A segunda componente em M é uma propriedade de segurança do serviço da plataforma Intel SGX, mas como as informações de propriedade de segurança do Serviço de plataforma Intel SGX não são necessárias no processo de atestação remota e troca de chaves, este campo é todo zeros.

Tabela 3.3: Dados da quarta mensagem

CMAC_{SMK}(M) || M

Ao receber a msg3 do cliente, o SP averigua se o Ga recebido é o mesmo que o recebido na msg1. Depois, verifica o CMAC_{SMK}(M) e se os primeiros 32 *bytes* dos dados do corpo do relatório incluído no *quote* correspondem ao SHA-256 de Ga || Gb || VK. O VK é derivado executando um AES-128 CMAC na sequência de *bytes* 0x01 || "VK" || 0x00 || 0x80 || 0x00 usando o KDK como chave.

Seguidamente, verifica a evidência de atestação fornecida pelo cliente extraindo o *quote* da msg3 e enviando-o para o IAS recorrendo à API respetiva para tal efeito. Ao receber a resposta do IAS, tem que validar o certificado de assinatura recebido. Com esse certificado dado como válido, resta validar a assinatura da resposta usando o mesmo certificado de assinatura.

Se o *quote* foi aprovado com êxito pelo IAS, o servidor extrai da resposta o estado de atestação (Attestation_Status) para o enclave do cliente e examina a sua identidade (MRSIGNER), versão de segurança, identificador do produto e atributo *debug*. Na política conceptualizada, o SP apenas confia em enclaves cujo atributo *debug* não está definido e cuja resposta do IAS foi "OK", ou seja, a assinatura EPID do *quote* do enclave foi verificada corretamente e o nível da TCB da plataforma SGX está atualizado.

Se o enclave está aprovado, deriva-se as chaves de sessão, SK e MK, que devem ser usadas para transmitir mensagens futuras entre o cliente e o servidor durante a sessão. Para as obter, executa-se um AES-128 CMAC nas seguintes sequências de bytes, usando o KDK como a chave: MK: 0x01 || "MK" || 0x00 || 0x80 || 0x00 e SK: 0x01 || "SK" || 0x00 || 0x80 || 0x00. Finalmente, gera-se a última mensagem (msg4) para o cliente cujo conteúdo é se o enclave é ou não confiável, um PIB (*Platform Info Blob*), explicado abaixo, e o certificado da chave pública de curvas elítpicas do enclave assinado pelo SP com a sua chave privada de curvas elípticas e encriptado usando AES no modo CBC com a chave MK (CertEPubKey).

A verificação das evidências de atestação exige que o SP envie o *quote* ao IAS e obtenha um relatório de atestação. Este relatório é assinado pela chave privada de assinatura de relatórios do IAS e o SP deve validar esta assinatura usando a chave pública de assinatura de relatórios do IAS.

O próprio relatório contém cabeçalhos de resposta e um *payload* que contém o estado de atestação para o enclave. Um valor de "OK" significa que o enclave é de confiança, mas qualquer outro valor significa que o mesmo não é confiável ou é confiável se determinadas ações forem tomadas (por exemplo, com uma atualização de *software* ou do BIOS).

Se o estado da atestação não for "OK", o IAS envia um PIB que o SP encaminha ao cliente para este obter informações mais detalhadas sobre o que falhou.

Tabela 3.4: Dados da última mensagem

Attestation_Status || PIB || AES-CBC$_{MK}$(CertEPubKey)

No momento em que o cliente tem a prova de aceitação do servidor explicitada na msg4, também deriva no enclave as mesmas chaves de sessão SK e MK, desencripta o certificado com a chave MK e verifica a assinatura do certificado usando a chave pública de curvas elípticas do SP.

Assim que o cliente processa a msg4, o fluxo de atestação termina e o cliente possui o seu CertEPubKey que funciona como "passaporte" para efetuar envios de transações na rede *peer-to-peer*. Daqui em diante, basta anexar este certificado a uma transação assinada pela sua chave privada de curvas elípticas para que o seu destinatário possa comprovar que o emissor da mesma é autenticado pelo SP. O diagrama de todo o fluxo de atestação está apresentado na figura 3.3.

3.3 Protocolo de transferência

Para os intervenientes da rede transferirem quantias entre si, enviam diretamente as transações sem passar por qualquer entidade central. Cada uma é identificada por 4 campos principais: o seu emissor, recetor, quantia e assinatura ECDSA (SigT). Sempre que um cliente pretende enviar uma moeda para outrem, recorre a uma chamada ao enclave para gerar a estrutura da transação totalmente preenchida, portanto, toda ela é criada em ambiente protegido. Assim que o enclave devolve a transação assinada, o cliente anexa o seu certificado resultante do processo de atestação à mesma e procede ao envio para o destinatário pretendido. Com estas ações é possível garantir a integridade total das transações, porque nenhum cliente consegue manipular o processo do enclave que as gera. Na tabela 3.5, considerando T = Emissor || Recetor || Quantia, são mostrados todos os dados que incluem uma transação:

Tabela 3.5: Dados de uma transação

T || SigT || CertEPubKey

Quando um cliente recebe uma transação, é efetuada novamente uma chamada ao enclave mas agora para se verificar a validade da transação. O primeiro passo neste processo,

é verificar se o certificado anexado é realmente emitido pelo SP, através da sua chave pública, para garantir a validade da sua assinatura. De seguida, é necessário assegurar que a assinatura SigT na transação é válida usando a chave pública do enclave certificada (dentro do certificado). Somente se todos os procedimentos anteriores ocorrerem sem problemas, é que o saldo da carteira de quem recebe a quantia transacionada pode ser incrementado.

Como já foi referido anteriormente, os dados do enclave são perdidos quando a aplicação é encerrada, portanto, tudo o que o enclave desenhado armazena é selado previamente em memória não volátil. Através dos mecanismos de selagem, neste caso pelo MRSIGNER, os dados que o enclave mantém secretos (saldo, par de chaves, etc) são salvaguardados em disco num ficheiro cujo conteúdo está completamente encriptado e ilegível. Apenas o enclave que encriptou/selou os dados no ficheiro é capaz de recuperar o conteúdo em *plaintext*.

Embora estas metodologias permitam criar transações íntegras, ainda existe um problema quando as transações atravessam a rede para chegar ao seu destinatário. Se as transações forem enviadas apenas com os dados indicados na tabela 3.5 podem surgir ataques de *replaying*, o que invalidaria por completo um sistema de troca de dinheiro, porque qualquer transação podia ser replicada e o dinheiro reutilizado mais que uma vez. Para proteção contra estes ataques, é adicionado um *nonce* como um campo extra da transação. Tendo em conta que os *nonces* devem aparecer apenas uma única vez para cada nova transação, se por ventura um cliente receber duas transações com o mesmo valor do *nonce* sabe automaticamente que se trata de uma transação repetida que deve ser invalidada.

Capítulo 4

Implementação do Sistema

Durante este capítulo são indicadas as ferramentas (bibliotecas, *kits* de desenvolvimento de *software*, arquiteturas e métodos criptográficos) que foram usadas na implementação do sistema cujo desenho foi apresentado no capítulo anterior. A seleção de uma alternativa em prol de outra também possível é sempre explicada com o intuito de mostrar a reflexão efetuada acerca das suas vantagens e desvantagens. Aqui, é igualmente apresentada a algoritmia, em pseudocódigo, dos procedimentos/funcionalidades mais importantes efetivamente concretizados(as).

4.1 Ferramentas

Para suportar o processamento isolado via Intel SGX e a atestação dos integrantes da rede, foram analisadas três possíveis ferramentas que encapsulavam à priori estas utilidades: um pacote *JavaScript* denominado secureworker[1], uma plataforma de nome SCONE [4] e o próprio *kit* de desenvolvimento de *software* nativo das Intel SGX, o Intel SGX SDK[2].

Numa primeira instância a primeira opção seria a mais atrativa, porque facilitaria não só os testes de avaliação e desempenho do sistema, dado que o objetivo desta dissertação seria integrar/testar o produto resultante nas infraestruturas do projeto onde está inserida, como também a respetiva implementação através das funcionalidades expostas pela sua API. Esta alternativa permitia focar inteiramente na vertente *blockchain* do sistema, que acabou por não ser realizada, porque toda a implementação quer do processo de atestação quer do processamento isolado (enclave) estava já feita pela biblioteca fornecida. Aquando da tentativa de utilizar este código-fonte, verificou-se que estava inutilizável

[1] https://github.com/evervault/node-secureworker
[2] https://software.intel.com/en-us/sgx/sdk

e inativo o que obrigou a uma mudança de abordagem. Com isto, restava utilizar ou a plataforma SCONE ou o *kit* da Intel.

Tal como a biblioteca *secureworker*, também a plataforma SCONE abstrai totalmente as configurações de atestação e a programação dos enclaves, no entanto, ao optar pelo SDK das Intel SGX ao invés desta, consegue-se um controlo mas rigoroso sobre os detalhes da atestação, flexibilidade na programação do enclave e suporte direto (APIs, bibliotecas, documentação, códigos-exemplo e ferramentas) da empresa proprietária da tecnologia. Embora com o SDK tenha que se programar de raiz os complexos procedimentos necessários para preparar os clientes da rede para enviarem transações, o que não seria preciso realizar usando uma das outras alternativas, as vantagens que advêm desta abordagem permitem construir um sistema mais refinado que atende às especificidades estipuladas.

O estilo arquitetural de *software* fixado para sustentar as comunicações entre as entidades definidas foi a REST. Esta foi a opção tomada para fornecer interoperabilidade pela Internet entre as diferentes máquinas do sistema, porque a separação de preocupações entre cliente e servidor na REST simplifica a implementação e a obtenção de uma interface uniforme. Além disto, é uma vantagem as interações não possuírem estado entre as solicitações e serem usados métodos e tipos de dados padrão, JSON neste caso, para indicar as semânticas e trocar informações. Assim, os membros da rede *peer-to-peer* desempenham o papel tanto de clientes como de servidores REST, dado o tipo de rede de que se trata, e o SP funciona como um servidor REST. Já o IAS, é um serviço *web* hospedado e operado pela Intel num ambiente em nuvem, cuja API é criada usando também o estilo arquitetural REST padrão da indústria e também a JSON como o formato de serialização de dados.

A solução criada assenta sobre o sistema operativo *Windows* e foi dividida em três projetos distintos: o projeto com o código da parte não segura (aplicação) do cliente, o projeto com o código da parte segura (enclave) do cliente e o projeto com o código do SP. A aplicação e o enclave estão programados usando a linguagem de programação C/C++ mas apenas o enclave é obrigatoriamente escrito nestas linguagens, pois está confinado às regras impostas pelo Intel SGX SDK. Embora esta tenha sido a decisão tomada, nada impede um desenvolvedor de usar outra linguagem de programação para a aplicação e interagir com a interface do enclave.

Programaticamente, as variadas interações e conexões remotas usam o código do projeto C++ REST SDK. Esta biblioteca foi escolhida pela sua forte abstração quanto aos conceitos de redes de mais baixo nível, pelo seu design moderno da API C++ assíncrona, em código nativo, e porque vai de encontro ao estilo arquitetural definido. De notar que esta biblioteca é multiplataforma (*Windows desktop*, *Windows Store* (UWP), *Linux*, OS X, *Unix*, iOS, e *Android*).

Quanto às rotinas criptográficas, o lado do cliente apenas precisa de tirar partido das que já estão incluídas na biblioteca confiável do próprio Intel SGX SDK. As funções criptográficas nesta biblioteca são exclusivas ao enclave e são suficientes para preencher os seus requisitos. Para além disto, o Intel SGX SDK também encapsula convenientemente todo o fluxo de atestação, assim como todas as funcionalidades criptográficas necessárias

para o suportar. O mesmo não acontece no lado do SP (servidor), o que obriga à escolha de outro conjunto de ferramentas para implementar de raiz as interações de atestação e a criptografia inerente. Neste caso, foi escolhido o OpenSSL por ser um *kit* robusto, composto, bem documentado e que cede os instrumentos para programar as ações em falta no servidor.

4.2 Cliente

Nesta seção, são detalhados os processos que o cliente utiliza para produzir as mensagens que envia no fluxo apresentado no capítulo 3.

4.2.1 Habilitar as SGX e carregar o enclave

O primeiro procedimento efetuado no programa do cliente é habilitar na sua máquina as SGX do seu processador. O conjunto de passos que constituem esta operação seriam completamente abstraídos caso se tivesse optado pelo *secureworker* ou pela SCONE. Como não se optou por nenhum dos dois, a ativação das SGX e a inicialização e destruição do enclave resultante têm de ser escrutinadas em código. Como forma de abstrair todos estes detalhes, foi criada uma classe com código inteiramente dedicada à gestão de todas as componentes das SGX.

Para ativar dinamicamente esta tecnologia, existe uma função específica da API do SDK para o fazer: *sgx_enable_device*. Esta função devolve o estado das SGX (ligadas ou desligadas) e, conforme este retorno, a aplicação do cliente toma a próxima ação:

1. Se ocorreu uma atualização da máquina, desativando as Intel SGX, a execução desta API reativa-as mas somente após reiniciar a máquina;

2. Se as Intel SGX não estiverem de todo ativadas, o utilizador é informado que deve encerrar a aplicação e que é necessária uma reinicialização do computador para que a aplicação possa correr;

3. Se o retorno possuir o valor *SGX_DISABLED*, é dado a conhecer ao utilizador que pode ser necessária uma configuração do BIOS para ativar manualmente as Intel SGX.

4. Em qualquer outro caso excecional, a aplicação impede o cliente de continuar e este é avisado acerca do que correu mal e como deve proceder em ordem a resolver o problema.

Em todo o caso, apenas clientes cujas máquinas possuem um processador com esta tecnologia podem utilizar esta aplicação, logo, é rejeitada qualquer execução em modo não SGX. No algoritmo 1 é apresentado em pseudocódigo como é verificado e ativado o estado das Intel SGX. Embora não estejam apresentados todos os casos de erro possíveis, e

como estes são abordados, o código responsável por esta parte está preparado para reagir a qualquer erro documentado na API da função *sgx_enable_device*.

```
Resultado: Estado das SGX
sgx_device_status = sgx_enable_device();
selecione sgx_device_status faça
    caso SGX_ENABLED faça // plataforma ativada para as Intel SGX
        print("This platform is enabled for Intel SGX.\n");
        break;
    fim
    senão faça
        imprimir erro que ocorreu;
        break;
    fim
fim
retorna sgx_device_status
```

Algoritmo 1: Consulta e ativação do estado das Intel SGX

Após a máquina do cliente estar estabelecida como pronta para utilizar as SGX o seu único enclave é carregado. Existem aspetos relevantes a ter em conta no que toca ao carregamento do enclave: o código-fonte do enclave é criado como uma biblioteca de *link* dinâmico no ato da compilação da solução com os três projetos já referidos. Para usar o enclave que foi escrito, o respetivo enclave.dll é carregado na memória protegida chamando a API *sgx_create_enclave*. O enclave.dll é assinado para o ficheiro Enclave.signed.dll pela ferramenta de assinatura de enclaves (*sgx_sign*.exe), fornecida pelo SDK, como um processo de pós compilação. Ao carregar o enclave pela primeira vez, o carregador obtém um *token* de inicialização que é guardado no parâmetro *token* da interface *sgx_create_enclave*. Este *token* é salvo num ficheiro (Enclave.token) na diretoria do sistema de ficheiros que serve como repositório de dados para aplicações locais (CSIDL_LOCAL_APPDATA), para que ao carregar o enclave pela segunda vez, a aplicação possa obter o *token* a partir desse ficheiro. Ao fornecer o *token* desta forma promove-se o desempenho do carregamento do enclave. Para descarregar o enclave é chamada posteriormente a interface *sgx_destroy_enclave* com o identificador do enclave.

Em termos programáticos, o arranque do enclave divide-se em três passos que podem ser observados com mais detalhe no algoritmo 2:

1. Obter o *token* salvo pela última execução na diretoria: C:\Users*username*\AppData\Local\Enclave.token (CSIDL_LOCAL_APPDATA). Se não existir, criar um novo;

2. Chamar a função *sgx_create_enclave* para gerar uma instância do enclave. Este método carrega o enclave usando o nome do seu arquivo assinado (Enclave.signed.dll) e o *token* de inicialização. Daqui resulta um identificador para o enclave e informação que indica se o *token* foi ou não atualizado;

3. Salvar o *token* se foi atualizado.

4.2.2 Primeira etapa da atestação

Já com as SGX ligadas e o enclave ativo, falta ainda o cliente efetuar o mecanismo de atestação remota com o SP para assegurar que é genuíno e que não está comprometido. Durante este processo, o cliente recorre ao enclave através das funções da sua interface para obter determinadas informações fundamentais que somente este tem acesso. Só depois de ter concluído a atestação é que o cliente pode efetivamente enviar transações para os outros participantes.

A primeira etapa da atestação remota envolve o cliente solicitar ao SP o fornecimento de segredos. Geralmente, isto é uma API específica que o SP implementa para fazer tal solicitação. O SP responde a este pedido emitindo um desafio solicitando ao cliente que se ateste. Pressupõe-se que os detalhes deste *handshake* são deixados para quem implementa o seu serviço. Em resposta à solicitação de desafio, o cliente executa várias etapas para construir a mensagem inicial do fluxo de atestação remota. Como o enclave do cliente já está inicializado, o código não confiável executa as seguintes etapas:

1. Executar a ECALL EcallGenerateRaDhkeContext que gera o contexto da atestação remota e de DHKE;

2. Já dentro do enclave:

    1. Chamar a função *sgx_ra_init*;

    2. Devolver o resultado e o contexto da DHKE para o código não confiável.

3. Chamar a função *sgx_get_extended_epid_group_id*.

A função *sgx_ra_init* aceita a chave pública do SP, *hardcoded* no código do enclave, como argumento e devolve um contexto opaco para a DHKE que ocorre durante a atestação remota. O formato da chave pública de 256 *bits* é definida pelo tipo de dados *sgx_ec256_public_t* que representa uma chave de curva elítica NIST P-256 representada pela sua coordenada x seguida da sua coordenada y. As coordenadas desta chave estão na ordem de *bytes little-endian*.

Após um resultado bem-sucedido da função *sgx_ra_init*, o cliente faz uma chamada à função *sgx_get_extended_epid_group_id* para adquirir o identificador estendido do grupo Intel EPID, o esquema de assinatura anónima usado. O cliente envia este identificador ao SP como o corpo da msg0, mensagem esta que é enviada juntamente com a msg1 por razões de eficiência. Afim de obter a msg1, que contém a sua chave pública para a DHKE e o identificador do grupo Intel EPID a que a sua máquina pertence, o cliente chama a função *sgx_ra_get_msg1*. Parâmetros adicionais deste método incluem o contexto da DHKE obtido anteriormente, um apontador para a função *sgx_ra_get_ga* para calcular o segredo

```
Resultado: Enclave inicializado
// Passo 1.
// Obter diretoria do ficheiro do token
token_path = CSIDL_LOCAL_APPDATA + "Enclave.token";
token = 0;
sgx_return = SGX_ERROR_UNEXPECTED;
enclave_id;
updated = 0;
// Abrir ficheiro do token
token_file = CreateFile(token_path);
se token_file == INVALID então
    print("Failed to create/open the launch token file");
senão // Ler token do ficheiro
    bytes_read = 0;
    ReadFile(token_file, token, bytes_read);
fim
se bytes_read != 0 && bytes_read != token_size então
    // Token inválido, limpar variável token
    token = 0;
    print("Invalid launch token read");
fim
// Passo 2.
sgx_return = sgx_create_enclave("Enclave.signed.dll", token, updated, enclave_id,
 NULL);
se sgx_return != SGX_SUCCESS então
    imprimir erro que ocorreu;
    se token_file != INVALID então
        CloseFile(token_file);
    fim
    retorna sgx_return;
fim
// Passo 3.
se updated == FALSE || token_file == INVALID então
    // Token não atualizado ou ficheiro inválido, não guardar token
    se token_file != INVALID então
        CloseFile(token_file);
    fim
    retorna sgx_return;
fim
bytes_read = 0;
WriteFile(token_file, token, bytes_read);
se bytes_read != token_size então
    print("Failed to save launch token");
fim
CloseFile(token_file);
retorna sgx_return;
```

Algoritmo 2: Carregamento e inicialização do enclave

do lado do cliente e o identificador do enclave. A função *sgx_ra_get_ga* é gerada automaticamente pelo SDK quando o enclave é vinculado com a biblioteca *sgx_tkey_exchange* e importa o arquivo *sgx_tkey_exchange*.edl de definição da linguagem do enclave. A função *sgx_ra_get_msg1* retorna a msg1 com todos os componentes na ordem de *bytes* correta, isto é, na ordem *little-endian*.

Tabela 4.1: Estrutura da msg0

Elemento	**Tamanho**
Extended Intel EPID GID	4 *bytes*

Tabela 4.2: Estrutura da msg1

Elemento		**Tamanho**
Ga	Ga_x	32 *bytes*
	Ga_y	32 *bytes*
Intel EPID GID		4 *bytes*

Junto da msg0 e da msg1 segue a chave pública do enclave, constituída por 64 *bytes*, pertencente ao par de chaves gerado imediatamente antes de proceder ao envio para efeitos de assinatura e verificação de transações. O resumo de todo este conjunto de operações que servem de construção à primeira mensagem do fluxo de atestação para o SP está apresentado no algoritmo 3.

```
Resultado: Primeira mensagem para o SP (msg0 + msg1 + EPubKey)
// Obter contexto de atestação remota e de DHKE entrando no enclave
ra_dhke_context = GenerateRaDhkeContext(enclave_id);
// Obter msg0 (GID estendido Intel EPID)
extended_epid_gid = sgx_get_extended_epid_group_id();
// Obter msg1 (Intel EPID GID e chaves para DHKE) entrando no
   enclave
msg1 = sgx_ra_get_msg1(ra_dhke_context, enclave_id, sgx_ra_get_ga);
// Obter chave pública para transações
GenerateEcc256KeyPair(enclave_id, enclave_pub_key);
```

Algoritmo 3: Criação da primeira mensagem

A partir deste ponto, o cliente efetua um pedido HTTP POST ao SP com a primeira mensagem no seu corpo para este a processar e dar seguimento ao fluxo de atestação. De notar que para todos os pedidos HTTP que ocorrem durante o fluxo pressupõe-se que existe um *path* especializado para cada um no SP. O pseudocódigo da conversão da primeira mensagem no formato indicado e o respetivo envio, utilizando a *framework* C++ REST SDK, está exposto no Algoritmo 4. Para além da primeira mensagem, também

as restantes mensagens do fluxo são serializadas e enviadas para o SP usando a mesma *framework*.

Resultado: Envio da primeira mensagem para o SP
Saída:

```
{
   "msg0": extended_epid_gid,
   "msg1": hex(ga || gid),
   "epubkey": hex(enclave_pub_key)
}
// Criar objeto JSON vazio
msg0_msg1_epubk_json;
// Criar par chave-valor "msg0": extended_epid_gid
msg0_msg1_epubk_json["msg0"] = extended_epid_gid;
// Criar par chave-valor "msg1": hex(ga || gid)
msg0_msg1_epubk_json["msg1"] = hex(msg1);
// Criar par chave-valor <"epubkey": hex(enclave_pub_key)>
msg0_msg1_epubk_json["epubkey"] = hex(enclave_pub_key);
// Pedido HTTP POST assíncrono da primeira mensagem ao SP
sp_client.request(POST, msg0_msg1_epubk_json).then(
    (http_response msg2){...};
```

Algoritmo 4: Serialização e envio da primeira mensagem

A função *request* atua sobre o objeto que funciona como o cliente do SP (sp_client) enviando-lhe assincronamente um pedido HTTP POST com a primeira mensagem no seu *payload*. Esta função devolve uma tarefa assíncrona que é concluída assim que é recebida a resposta à solicitação (msg2). Para manipular a resposta obtida, msg2, é anexada uma função manipuladora à tarefa, através da função *then*, que é chamada assim que a tarefa concluir.

4.2.3 Segunda etapa da atestação

Quando é recebida a msg2 vinda do SP, o cliente utiliza a função específica do SDK *sgx_ra_proc_msg2* para a processar e gerar a msg3 da atestação remota e do protocolo de troca de chaves que é enviada para o SP. Se o SP aceitar esta mensagem, as chaves de sessão negociadas entre o enclave da aplicação e o SP estão prontas para uso. O enclave da aplicação usa a função *sgx_ra_get_keys* para obter as chaves negociadas e a função *sgx_ra_close* para libertar o contexto de atestação remota e do processo de troca de chaves quando o fluxo terminar. Se porventura o processamento da msg2 resultar num erro, a aplicação notifica o SP sobre o erro ou o SP, através de um mecanismo de tempo limite, finaliza a atestação remota quando não receber a msg3.

Quando a aplicação executa a função *sgx_ra_proc_msg2* são realizadas as seguintes etapas:

1. Verificar a assinatura do SP;

2. Verificar a lista de assinaturas revogadas;

3. Devolver a msg3 que contem o *quote* usado para atestar o enclave.

Dois dos parâmetros passados para a função *sgx_ra_proc_msg2* são dois apontadores para funções, *sgx_ra_proc_msg2_trusted* e *sgx_ra_get_msg3_trusted*, geradas automaticamente pela ferramenta *Edger8r* que é responsável por produzir as interfaces entre os componentes não confiáveis e o enclave. Parte do processamento realizado por *sgx_ra_get_msg2_trusted* é computar o *quote* do enclave que inclui um *hash* criptográfico, ou medição, do enclave em execução assinado com a chave EPID da máquina do cliente. Esta assinatura só pode ser verificada pelo servidor IAS.

Tabela 4.3: Estrutura da msg3

Elemento			**Tamanho**
CMAC_{SMK}(M)			16 *bytes*
M	Ga	Ga_x	32 *bytes*
		Ga_y	32 *bytes*
	Ps_Security_Prop		256 *bytes*
	Quote		(436 + *quote.signature_len*) *bytes*

Antes de detalhar os componentes da msg3, é relevante mencionar a nomenclatura e significado das chaves usadas ao longo do fluxo de atestação:

- SK - *Signing Key*/*Symmetric Key*
- MK - *Master Key*/*Masking Key*
- SMK - SIGMA *Protocol Key*
- VK - *Verification Key*
- KDK - *Key Derivation Key*

O MAC da msg3 resulta da aplicação do AES-CMAC sobre Ga, *Ps_Security_Prop*, GID e *quote* usando a chave SMK que é derivada da seguinte forma:

- KDK = AES-CMAC(*key0*, *LittleEndian*(coordenada *x* de gab))
- SMK = AES-CMAC(KDK, 0x01 || 'SMK' || 0x00 || 0x80 || 0x00)

A chave *key0* usada na operação de extração da chave KDK são 16 *bytes* 0x00 e o *plaintext* gab é o elemento do campo de curvas elípticas secreto partilhado *Diffie-Hellman* entre o SP e o enclave no formato *little-endian*. Os parâmetros usados no cálculo da SMK são:

- um contador (0x01)
- um rótulo (a representação ASCII da *string* 'SMK' em *little-endian*)
- um número de *bits* (0x80)

Os primeiros 32 *bytes* do campo *report_body.report_data* do *quote* são definidos como o *hash* SHA-256 de Ga, Gb e VK e os segundos 32 *bytes* definidos todos como zero. VK é derivado do elemento do campo de curvas elípticas secreto partilhado *Diffie-Hellman* entre o SP e o enclave da aplicação: VK = AES-CMAC(KDK, 0x01 || 'VK' || 0x00 || 0x80 || 0x00). No cálculo da VK estão incluídos exatamente os mesmos valores do cálculo da SMK com a exceção do rótulo que ao invés de tomar o valor 'SMK' toma o valor 'VK'.

A estrutura do *quote* do enclave é definida no cabeçalho *sgx_quote.h* incorporado no SDK como apresentado na tabela 4.4.

Tabela 4.4: Estrutura do *quote*

Elemento	**Significado**	**Tamanho**
version	A versão da estrutura do *quote*	2 *bytes*
sign_type	Indicador do tipo de assinatura Intel EPID	2 *bytes*
epid_group_id	Identificador do grupo Intel EPID a que pertence a plataforma	4 *bytes*
qe_svn	SVN do *Quoting Enclave*	2 *bytes*
pce_svn	SVN do *Provisioning Certification Enclave*	2 *bytes*
xeid	Identificador estendido do grupo Intel EPID	4 *bytes*
basename	Nome base usado no sgx_quote	32 *bytes*
report_body	Corpo do relatório do enclave da aplicação	384 *bytes*
signature_len	Tamanho em *bytes* da assinatura	4 *bytes*
signature	Marcador de posição da assinatura de comprimento variável	signature_len *bytes*

No algoritmo 5 é possível observar todas as etapas que ocorrem desde que o cliente recebe a msg2 vinda do SP, como resposta à sua msg1, até ao seu processamento e criação da msg3 do fluxo de atestação.

Tal como no envio da primeira mensagem para o SP, é utilizada a função *request* sobre o objeto cliente do SP para lhe solicitar um HTTP POST com a msg3 no corpo do pedido. Mais uma vez, a função *request* devolve uma tarefa assíncrona que é concluída assim que é recebida uma resposta do SP (msg4). Para manipular a resposta obtida, msg4, é anexada novamente uma função manipuladora à tarefa, através da função *then*, que é chamada assim que a tarefa concluir.

```
Resultado: Terceira mensagem para o SP
// Pedido HTTP POST assíncrono da primeira mensagem ao SP
sp_client.request(POST,msg0_msg1_epubk_json).then(
    (http_response msg2)
{
    // Extrair resposta do SP (msg2)
    msg2_json = msg2.get();
    // Processar msg2 e criar msg3
    msg3 = sgx_ra_proc_msg2(ra_dhke_context, enclave_id,
     sgx_ra_proc_msg2_trusted, sgx_ra_get_msg3_trusted, msg2, msg2_size);
});
```

Algoritmo 5: Criação da terceira mensagem

4.2.4 Terceira etapa da atestação

Embora os detalhes da mensagem quatro que o SP envia como resposta à mensagem três do cliente sejam exibidos no capítulo seguinte, importa referir que no instante em que o cliente a recebe este efetua verificações essenciais sobre os dados da mensagem:

1. Verificar se o certificado do SP da chave pública de curvas elíticas do enclave para verificação de transações é realmente emitido pelo próprio.

2. Verificar se o SP aprovou a autenticidade do enclave da aplicação. Em caso negativo, o cliente tem disponível a informação da causa para tal.

Se a análise de ambos os dados é positiva, o fluxo de atestação encerra e o cliente pode obter as chaves simétricas secretas que resultaram deste processo. Para isto, como já foi dito anteriormente, o cliente recorre ao enclave para efetuar uma chamada à função *sgx_ra_get_keys* para obter estas chaves. Para mantê-las secretas no enclave, o que realmente é devolvido ao cliente não são as chaves propriamente ditas mas sim o seu *hash* SHA-256. Dado que o fluxo de atestação terminou, o cliente regressa de novo ao enclave para libertar o contexto de atestação e de DHKE através do método *sgx_ra_close*.

Tudo o que foi indicado nos dois últimos parágrafos e na enumeração anterior pode ser visto com mais detalhe no algoritmo 6. Daqui em diante, como o enclave do cliente já está autenticado pelo SP, ou seja, pronto para transacionar na rede *peer-to-peer*, fica em escuta permanente por novos pedidos de transação de outros participantes também já autenticados pelo mesmo mecanismo.

4.2.5 Enviar uma transação

Para os clientes enviarem uma transação na rede para outros participantes precisam de fornecer o identificador do destinatário e a quantia a transferir. Neste processo, supõe-se que existe um mecanismo capaz de fazer a descoberta do endereço do destinatário a partir

```
Resultado: Envio da terceira mensagem para o SP
msg3_json = msg3;
// Pedido HTTP POST assíncrono da terceira mensagem ao SP
sp_client_.request(POST, msg3_json).then((http_response msg4)
{
    // Extrair resposta do SP (msg4)
    msg4_json = msg4;
    // Verificar o certificado de SP da chave pública do enclave
    X509SignatureVerify(msg4.pub_key_cert);
    se msg4.status == Trusted então
        // Enclave confiável. Obter hash das chaves MK e SK
        // entrando no enclave para provar o segredo partilhado com
           SP
        // Obter MK entrando no enclave com a função sgx_ra_get_keys
        mk_hash = RemoteAttestationKeyHash(enclave_id, ra_dhke_context,
         SGX_RA_KEY_MK);
        // Obter SK entrando no enclave com a função sgx_ra_get_keys
        sk_hash = RemoteAttestationKeyHash(enclave_id, ra_dhke_context,
         SGX_RA_KEY_SK;
    senão
        // Enclave não confiável. Obter informação do porquê do
           enclave não ser confiável
        print(sgx_report_attestation_status(msg4.platform_info_blob));
    fim
});
// Libertar contexto de atestação e de DHKE entrando no enclave
ReleaseRaDhkeContext(enclave_id, ra_dhke_context);
```

Algoritmo 6: Serialização da msg3 para envio e processamento da msg4

do seu identificador. Sendo assim, o cliente para gerar uma nova transação recorre ao seu enclave para preencher localmente os seguintes requisitos:

- Evitar o forjamento da assinatura da transação;
- Evitar o gasto duplo de uma quantia do saldo.

Como é o enclave que assina/verifica uma nova transação e que controla o saldo é garantido que o cliente não tem qualquer impacto na geração das transações. Embora o enclave permita impor estas condições, quando uma transação é enviada remotamente esta pode ser replicada e enviada novamente, o que deixa o sistema ainda suscetível ao problema do gasto duplo. Para defender este tipo de ataque cada transação é acoplada com um *nonce* gerado dentro do enclave. Dado que o enclave guarda a última associação válida entre o emissor de uma transação e o respetivo *nonce*, para cada nova transação basta verificar se a associação se repete. Em caso positivo, o enclave sabe que a transação é duplicada, caso contrário, é a primeira vez que a recebe. Isto apenas é possível porque

quando a aplicação é encerrada o respetivo enclave sela todos os seus dados em disco, logo armazena todas as associações entre diferente execuções. Visto que a assinatura da transação é controlada totalmente pelo enclave, mesmo que durante o trajeto do emissor para o recetor um atacante forge o *nonce* da transação, a validação da assinatura no recetor falha e a mesma é rejeitada.

A estrutura de uma transação apresentada na tabela 4.5 é apenas um molde e depende das necessidades do serviço que implementa o protocolo, no entanto, o *nonce* e a assinatura são elementos imprescindíveis.

Tabela 4.5: Estrutura de uma transação

Elemento	**Significado**	**Tamanho**
sender	*String* do identificador do emissor da transação	32 *bytes*
reciever	*String* do identificador do destinatário da transação	32 *bytes*
amount	Quantia da transação	4 *bytes*
nonce	*Nonce* para impedir ataques de *replaying*	32 *bytes*
signature	Assinatura ECDSA de sender || reciever || amount || nonce	64 *bytes*

Admitindo que qualquer integrante da rede aloja um *endpoint url*, como por exemplo *http://host:port*, a sua API REST necessita apenas de uma função para receber transações de outros pares. A implementação desta funcionalidade está descrita detalhadamente na tabela 4.6. Um cliente para enviar uma transação realiza as seguintes etapas:

1. Criar e assinar a transação entrando no enclave;
2. Anexar o certificado, enviado pelo SP na msg4, da chave pública do seu enclave;
3. Serializar a transação e o certificado;
4. Efetuar o envio.

Sempre que um cliente pertencente à rede recebe um pedido de receção de transação, o código executa as seguintes operações para validar a transação no *payload* (algoritmo 7):

1. Extrair o conteúdo do *payload* serializado em JSON;
2. Verificar o certificado do SP da chave pública do remetente da transação;
3. Efetuar chamada ao enclave para verificar a transação;
4. Responder ao remetente conforme a validade da transação.

```
Resultado: Saldo atualizado
// Verificar certificado do SP da chave pública do remetente
X509SignatureVerify(transaction_json["certificate"], sender_enclave_pub_key);
// Verificar transação entrando no enclave
verification_result = VerifyTransactionSignature(enclave_id, transaction,
 sender_enclave_pub_key);
se verification_result == SGX_EC_VALID então
    // Transação válida
    transaction_post.reply(OK)
senão
    // Transação inválida
    transaction_post.reply(BadRequest)
fim
```

Algoritmo 7: Receção de uma transação

Tabela 4.6: Detalhes da API de receção transações

Pedido		
Método HTTP	POST	
Recurso HTTP	http::://host:port/transactions	

Corpo do Pedido	Transação Serializada em JSON:
	{
	"sender":"<id_emissor>",
	"reciever":"<id_recetor>",
	"amount":"<quantia_transação>",
	"nonce":"<valor_hexadecimal>",
	"signature":"<assinatura_ecdsa_hexadecimal>",
	"certificate":"<x509_formato_pem>",
	}	

Parâmetros	N/A	
Cabeçalhos do Pedido	**Cabeçalho**	**Valor**
	Content-Type	"application/json"
Resposta		
Códigos de *status* HTTP	**Código**	**Descrição**
	200 OK	Operação de receção de transação bem sucedida. Assinatura da transação validada com sucesso
	400 Bad Request	*Payload* do pedido inválido ou assinatura da transação inválida. O cliente não deve repetir a solicitação sem modificações.
Corpo da Resposta	Vazio	

4.3 Enclave

4.3.1 Modelo de Programação

O *software* Intel SGX, que inclui um sistema de *runtime* Intel SGX, pode ser desenvolvido usando ferramentas e ambientes de desenvolvimento padrão. Embora o paradigma de programação seja muito semelhante ao *software* convencional, existem algumas diferenças em como o *software* Intel SGX é projetado, desenvolvido e depurado para aproveitar a tecnologia. Antes de detalhar como está programado o enclave, compara-se o modelo de programação disponível para o desenvolvimento de enclaves com modelos de programação mais familiar. Existem algumas semelhanças que reduzem a barreira de entrada para adotar a tecnologia Intel SGX. No entanto, deve-se estar ciente das diferenças de como o *software* Intel SGX é projetado, desenvolvido e depurado para criar enclaves robustos. Ao entender a tecnologia, bem como o modelo de programação que envolve, é possível extrair o maior benefício da mesma. Os seguintes princípios para desenvolver enclaves corretamente são fundamentais para não criar uma vulnerabilidade de segurança que pode ser explorada posteriormente:

- Um enclave é uma entidade de *software* monolítica que permite definir o TCB de uma aplicação para um sistema de *runtime* confiável, código e bibliotecas confiáveis de terceiros. Um erro numa componente pode comprometer as propriedades de segurança do enclave;
- O domínio não confiável controla a ordem em que as funções da interface do enclave são chamadas;
- Ao chamar um enclave, é o domínio não confiável que seleciona o *Trusted Thread Context* a ser usado no enclave;
- Não há garantia de que os parâmetros de entrada de uma chamada a um enclave (ECall) ou os parâmetros de retorno de uma chamada fora de um enclave (OCall) sejam o que o enclave espera, porque é o domínio não confiável que os fornece;
- A função não confiável chamada durante uma OCall pode não executar as operações esperadas pelo enclave;
- Qualquer pessoa pode carregar um enclave. Além disso, um invasor pode carregar um enclave com um programa desenvolvido especificamente para expor vulnerabilidades desse enclave.

Em alto nível, o *software* de suporte às Intel SGX oferece um modelo de programação semelhante ao que se está habituado ao desenvolver aplicações para os sistemas operativos mais comuns, ou seja, no caso do *Windows*, exposto através de uma DLL. Um arquivo regular DLL contém seções de código e dados correspondentes às funções e/ou métodos,

bem como às variáveis e/ou objetos implementados na biblioteca partilhada. O sistema operativo aloca um *heap* quando o processo que usa a biblioteca partilhada é carregado e uma pilha para cada *thread* gerada no processo. Da mesma forma, um arquivo de biblioteca do enclave contém seções confiáveis de código e dados que serão carregadas na memória protegida *Enclave Page Cache* (EPC) quando o enclave for criado. No arquivo de um enclave, também existe uma estrutura de dados específica das Intel SGX: os metadados do enclave. Os metadados não são carregados na EPC, em vez disso, são usados pelo carregador não confiável para determinar como carregar corretamente o enclave na EPC. Os metadados definem vários *trusted thread contexts* confiáveis, que incluem a pilha confiável e um *heap* confiável inicializado pelo *trusted runtime system* confiável na inicialização do enclave. *Thread contexts* confiáveis e um *heap* confiável são necessários para suportar um ambiente de execução confiável. Os metadados também contêm a estrutura de assinatura do enclave que é um certificado vital de autenticidade e origem de um enclave. Mesmo que um enclave possa ser entregue como um arquivo de biblioteca partilhada, a definição de que código e dados são colocados dentro do enclave e o que permanece fora na aplicação não confiável é um aspecto essencial. Os enclaves, independentemente do número de *threads* confiáveis definidos, não devem ser projetados com a suposição de que a aplicação não confiável invoca as funções da interface do enclave seguindo uma ordem específica. Assim que o enclave é inicializado, um invasor pode chamar qualquer função da interface, organizar as chamadas em qualquer ordem e fornecer quaisquer parâmetros de entrada. Estas informações são fundamentais para impedir ataques a um enclave.

A primeira etapa no design de uma aplicação habilitada para Intel SGX é identificar os ativos que precisa de proteger, as estruturas de dados em que os ativos estão contidos e o código que opera nessas estruturas de dados para depois colocá-lo numa biblioteca confiável. Deve-se realizar uma análise de segurança da aplicação e particioná-la adequadamente, tomando a decisão sobre que código e dados são colocados no enclave. O código de um enclave não é diferente do código que existe como parte de uma aplicação regular, no entanto o código do enclave é carregado de uma maneira especial, de modo que, uma vez inicializado o enclave, o código privilegiado e o resto da aplicação não confiável não possam ler diretamente os dados que residem no ambiente protegido ou alterar o comportamento do código no enclave sem detecção. Por este motivo, apesar de identificar os componentes de processamento secreto e os recursos que usam seja uma etapa importante em qualquer processo seguro de desenvolvimento de *software*, para o uso das Intel SGX é uma atividade essencial. Particionar uma aplicação em componentes confiáveis e não confiáveis tem implicações adicionais do ponto de vista de segurança. É geralmente aceite que um espaço menor de memória (código e dados menores) geralmente implica uma menor chance de apresentar defeitos no produto final. Também implica análise de segurança mais simples e *software* mais seguro, pois é exposta uma superfície de ataque. Portanto, embora seja possível mover a maioria do código da aplicação para um enclave, na maioria dos casos isso não é desejável. O tamanho da TCB deve ser um fator a ser considerado ao projetar o que se passa dentro de um enclave. Deve-se tentar minimizar o

tamanho do enclave, mesmo que a arquitetura Intel SGX proteja o conteúdo do enclave quando o sistema operacional, o VMM ou o BIOS estiverem comprometidos.

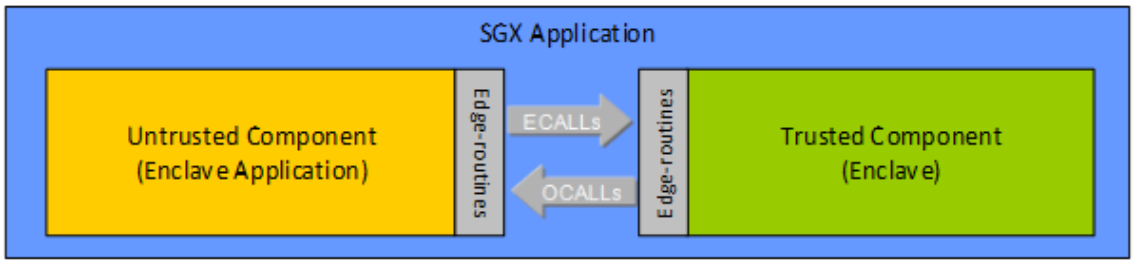


Figura 4.1: Aplicação Intel SGX

Depois de definir os componentes confiáveis (enclave) e não confiáveis (aplicação) de uma aplicação habilitada para Intel SGX, deve-se definir cuidadosamente a interface entre a aplicação não confiável e o enclave. O código confiável é executado nos seguintes cenários:

- A aplicação não confiável faz explicitamente uma chamada para uma função da interface do enclave dentro do enclave, por exemplo, a aplicação faz uma ECall.
- Depois de uma chamada feita de dentro do enclave para a aplicação externa (OCall) retornar.
- Após o retorno de uma interrupção, o código do enclave também é executado. No entanto, a arquitetura Intel SGX garante que a execução no enclave continue como se a interrupção nunca tivesse ocorrido.

Um enclave deve expor uma API para as aplicações acederem (ECalls) e anunciarem que serviços fornecidos pelo domínio não confiável são necessários (OCalls). Como as ECalls expõe a interface que uma aplicação não confiável pode usar, deve-se reduzir a superfície de ataque do enclave limitando o número de ECalls. Também se deve estar ciente de que um enclave não tem controlo sobre qual ECall é executada ou a ordem na qual ECalls são invocadas. Assim, um enclave não pode depender de ECalls que ocorrem numa determinada ordem. Por outro lado, as funções da interface podem ser invocadas somente após a inicialização do enclave, o que significa que:

- Qualquer *re-basing* de endereços necessária é realizada com sucesso;
- Dados globais confiáveis, incluindo dados centrados na segurança são inicializados com sucesso;
- O *trusted thread context*, incluindo dados centrados na segurança do *thread* confiável em que a função esta a ser executada, foi inicializado com sucesso;
- As funções implícitas de inicialização confiável são executadas até completarem.

As *inputs* do enclave (e neste caso, as *outputs* do enclave) podem ser observadas e modificadas pelo código não confiável. Nunca se deve confiar em nenhuma informação proveniente do domínio não confiável e deve verificar-se sempre os parâmetros de entrada das ECall, bem como os valores de retorno das OCall. Ao aceitar *inputs* de fora do enclave, as suposições sobre o tamanho e o tipo dos valores transmitidos devem ser verificadas pelo *software* do enclave para garantir o comportamento correto. Depois de identificar a origem e/ou destino, deve-se decidir se é necessário aplicar proteção de integridade e/ou criptografia com verificações de *anti-replay* e de *liveness* para proteger as informações que em algum momento estão expostas ao domínio não confiável. Quando uma função da interface é invocada:

- Os argumentos da função e quaisquer dados *marshaled* dos parâmetros passados por referência estão dentro do ambiente confiável e não são acessíveis aos atacantes;
- Uma operação de leitura e/ou escrita nos argumentos, no valor de retorno e na referência *marshaled*, de acordo com as definições de parâmetro especificadas, não compromete a confidencialidade e a integridade do código/dados.
  - O argumento, o valor de retorno e os dados *marshaled* são alocados e geridos pelo *run-time* confiável, sem sobrepor nenhum código ou dados.
  - O tamanho de um argumento, o valor de retorno e a referência *marshaled* são os especificados (por exemplo, o tamanho do *buffer* dos dados *marshaled* referenciados por um parâmetro apontador é especificado por uma constante, outro parâmetro ou um campo no porção fixa dos dados reais).

Deve-se manipular referências ou apontadores com cuidado especial. Uma aplicação pode passar um apontador referenciando um local de memória dentro do limite do enclave, o que pode fazer com que o enclave substitua inadvertidamente código ou dados do enclave. Da mesma forma, se o *software* do enclave não estiver ciente de que um apontador faz referência a um local não confiável, o enclave pode vazar segredos. Para evitar estes problemas, o *software* do enclave deve determinar se a região de memória (especificada por um apontador e tamanho) está dentro ou fora da faixa linear do enclave antes de desreferenciar o apontador. Além disso, o enclave deve garantir que os dados não possam ser modificados após a verificação. Devem passar-se apenas pela interface limite do enclave apontadores para objetos de escopo conhecido dentro do enclave. Portanto, apontador para estruturas de dados C são razoáveis, mas indicadores para objetos C++ não são.

Os enclaves não podem aceder diretamente aos serviços fornecidos pelo sistema operativo. Em vez disso, um enclave deve fazer uma OCall para uma rotina de interface na aplicação não confiável. Embora efetuar esta chamada para fora adicione uma sobrecarga de desempenho, não há perda de confidencialidade. No entanto, a comunicação com o sistema operacional exige a libertação de dados ou a importação de dados não secretos, que precisam de ser tratados adequadamente. Mesmo que às vezes as OCalls

sejam necessárias, são chamadas fora do enclave e, portanto, associam alguns riscos de segurança:

- Operações de enclave que requerem uma OCall, como sincronização de *threads* e I/O, são expostas ao domínio não confiável. Um enclave deve ser projetado de forma a impedir que se liberte informações *side-channel* que permita a um invasor, que observa as funções não confiáveis chamadas pelo enclave, obter informações sobre os segredos do enclave.

- Um enclave deve estar preparado para lidar com o cenário em que a função OCall não é executada. O valor de retorno de uma OCall, que é um *input* do enclave, vem do domínio não confiável e não deve ser confiado. Pode parecer que uma OCall foi concluída com êxito quando não o foi. Por exemplo, um invasor pode descartar a solicitação de um enclave para escrever dados selados no disco e informar ao enclave que o arquivo foi gravado com êxito.

Quando uma função dentro do enclave chama uma OCall:

- A OCall expõe apenas os argumentos da função OCall (incluindo os dados referenciados) e o valor de retorno ao domínio não confiável

- Quando a OCall retornar, o valor de retorno e qualquer dado *marshaled* dos parâmetros de saída de passagem por referência estarão dentro do ambiente confiável (portanto, não acessível a um invasor) e os argumentos da função somente de entrada (incluindo os dados referenciados) não serão alterados. Quando o valor de retorno é um apontador, apenas a referência estará dentro do ambiente confiável. O *software* do enclave deve verificar o *buffer* de dados referenciado pelo apontador retornado como qualquer outra referência passada para o enclave.

- Quando a OCall retorna, o *trusted thread context* é o mesmo que o anterior à OCall, exceto os registos voláteis e os dados de saída na pilha confiável.

4.3.2 Desenvolvimento

Para desenvolver o enclave foram seguidas as seguintes etapas que são a norma para qualquer desenvolvimento de uma aplicação habilitada para SGX:

1. Gerar um projeto de enclave através do *plug-in* do IDE usado (*Microsoft Visual Studio*);

2. Definir a interface entre a aplicação não confiável e o enclave em arquivos EDL e os segredos que o enclave guarda. Os arquivos EDL (*Enclave Definition Language*) descrevem funções e tipos confiáveis e não confiáveis do enclave usados nos protótipos

das funções. A ferramenta *Edger8r* usa esses arquivos para criar funções C e respetivos cabeçalhos para exportações de enclave (usadas por ECALLs) e importações (usadas por OCALLs);

3. Implementar as funções da aplicação e do enclave;

4. Compilar a aplicação e o enclave. No processo de compilação, a ferramenta *Edger8r* gera as funções ponte confiáveis e não confiáveis. A ferramenta de assinatura do enclave gera os metadados e a assinatura do enclave;

5. Executar e depurar a aplicação em modo *hardware*.

Na perspetiva da aplicação, fazer uma chamada ao enclave (ECALL) é como qualquer outra chamada de função ao usar a função ponte não confiável. As funções do enclave são funções em C/C++ com várias limitações. Apenas se pode escrever funções de enclave em C e C++ (nativamente). Outros idiomas não são suportados. As funções do enclave podem contar com versões especiais das bibliotecas de *runtime* C/C++, *Standard Template Library*, sincronização e várias outras bibliotecas confiáveis que fazem parte do SDK das Intel SGX. Estas bibliotecas confiáveis são desenhadas especificamente para serem usadas dentro de enclaves.

Para estruturar as funcionalidades do enclave para a atestação remota e para as transações foram criados dois ficheiros EDL: *remote_attest_flow.edl* e *transactions.edl*. No primeiro, define-se a interface das funções de entrada no enclave (ECALLs) que são necessárias para suportar o fluxo de atestação. No segundo, são também exibidas ECALLs que o enclave expõe quer para gerar as chaves usadas para assinar e verificar transações quer os respetivos mecanismos de autenticação. Existe um terceiro ficheiro EDL gerado automaticamente no ato de criação do projeto, Encalve.edl, que é usado como ficheiro/interface de topo, pois importa os dois ficheiros indicados.

Um ficheiro EDL segue normalmente o formato mostrado na seguinte listagem:

Listagem 4.1: *Template* de um ficheiro EDL

```
1  enclave {
2      // Include files
3
4      // Import other edl files
5
6      // Data structure declarations to be used as parameters of the
7      //function prototypes in edl
8
9      trusted {
10         //Include header files if any
11         //Will be includedd in enclave_t.h
12
13         //Trusted function prototypes
14     };
15
```

```
16      untrusted {
17          //Include header files if any
18          //Will be included in enclave_u.h
19
20          //Untrusted function prototypes
21      };
22  };
```

Tendo em conta este formato, o conteúdo do ficheiro *remote_attest_flow.edl* é apresentado na listagem 4.2. Na linha 5 desta listagem importa-se do ficheiro *sgx_tkey_exchange.edl* oferecido pelo SDK três ECALLs usadas durante o fluxo de atestação remota:

- *sgx_ra_get_ga*: calcula o segredo DHKE do lado do cliente;
- *sgx_ra_proc_msg2_trusted*: processa a msg2, ou seja, verifica a assinatura do SP e a lista de assinaturas revogadas;
- *sgx_ra_get_msg3_trusted*: constrói a msg3 que contém o *quote* usado para atestar o enclave.

O cabeçalho *sgx_key_exchange.h* contem tipos específicos para a troca de chaves remota. Deste são importados o tipo de dados do descritor de propriedades de segurança do serviço de plataforma opaco para os utilizadores (*sgx_ps_sec_prop_desc_t*), do contexto retornado pela biblioteca de troca de chaves (*sgx_ra_context_t*), das chaves de 128 *bits* usadas (*sgx_key_128bit_t*), do tipo de chave SK e MK (*sgx_ra_key_type_t*) e das mensagens msg1, msg2 e msg3 (*sgx_ra_msg1_t*, *sgx_ra_msg2_t* e *sgx_ra_msg3_t* respetivamente).

Já o cabeçalho *sgx_ukey_exchange.h*, contem os protótipos para as APIs que a biblioteca confiável da troca de chaves expõe: *sgx_ra_get_msg1* usada para obter a mensagem 1 do protocolo para enviar ao SP e *sgx_ra_proc_msg2* usada para processar a mensagem 2 do protocolo enviada pelo SP e gerar a mensagem 3 para enviar ao mesmo.

Por fim, o cabeçalho *sgx_tkey_exchange.h* permite ao enclave ter acesso à função *sgx_ra_init* que cria um contexto para o protocolo, à função *sgx_ra_get_keys* usada para obter as chaves de sessão negociadas e à função *sgx_ra_close* para liberar o contexto do protocolo após a conclusão do processo e este não ser mais necessário.

Listagem 4.2: Ficheiro *remote_attest_flow.edl*

```
1  /* remote_attest_flow.edl - Remote attestation edl */
2
3  enclave {
4
5    from "sgx_tkey_exchange.edl" import *;
6
7    include "sgx_key_exchange.h"
8    include "sgx_ukey_exchange.h"
9
10     trusted {
```

```
11      include "sgx_tkey_exchange.h"
12
13      public sgx_status_t EcallGenerateRaDhkeContext(
14          [out] sgx_ra_context_t *ra_dhke_context);
15
16      public sgx_status_t EcallRemoteAttestationGetKeyHash(
17          [out] sgx_status_t *get_keys_status,
18          sgx_ra_context_t ra_dhke_context,
19          sgx_ra_key_type_t type,
20          [out] sgx_sha256_hash_t *hash);
21
22      public sgx_status_t EcallReleaseRaDhkeContext(
23          sgx_ra_context_t ra_dhke_context);
24
25      };
26
27      untrusted {
28
29      };
30  };
```

Nos protótipos das funções dos ficheiros EDL, os apontadores devem ser decorados explicitamente com um atributo de direção *in*, *out* ou um atributo *user_check*. A explicação de como funcionam estes atributos está na tabela 4.7.

Tabela 4.7: Comportamento dos atributos *in*, *out* e *user_check*

	ECALL	**OCALL**
user_check	Apontador não é verificado. Utilizador deve programar verificação e/ou cópia.	O mesmo que as ECALLs.
in	*Buffer* copiado da aplicação para o enclave. Posteriormente, as alterações afetam apenas o *buffer* dentro do enclave. Seguro mas lento.	*Buffer* copiado do enclave para a aplicação. Obrigatório se apontador aponta para dados do enclave.
out	A função confiável aloca um *buffer* a ser usado pelo enclave. Após o retorno, esse *buffer* será copiado para o *buffer* original.	O *buffer* não confiável é copiado para o enclave pela função confiável. Seguro mas lento.
in, *out*	Combina o comportamento de *in* e *out*. Os dados são copiados para a frente e para trás.	O mesmo que as ECALLs.

As três ECALLs identificadas no ficheiro *remote_attest_flow.edl* completam as funções necessárias dentro do enclave para suportar o protocolo de atestação, uma vez que o SDK já fornece à partida funções para criar a msg1, processar a msg2 e criar a msg3. Assim, faltava apenas estas três funções confiáveis que implementam a criação do contexto

de atestação e troca de chaves, a obtenção do *hash* das chaves de sessão acordadas no protocolo com o SP e a libertação do contexto no fim do mesmo.

De seguida, é possível observar com mais detalhe a explicação de cada uma das ECALLs no ficheiro *remote_attest_flow.edl*:

- **EcallConstructRemoteAttestationContext:** esta ECALL utiliza a função *sgx_ra_init* para começar o protocolo de atestação criando o contexto que mantém os cálculos intermédios da troca de chaves. Este método do SDK recebe como parâmetro a chave pública de curvas elíticas do provedor de serviços baseada na curva NIST P-256, *hardcoded* no código do enclave, e devolve num parâmetro de saída o contexto para as computações subsequentes do protocolo.

- **EcallRemoteAttestationGetKeyHash:** esta ECALL utiliza a função *sgx_ra_get_keys* para obter as chaves negociadas com o SP. Esta é chamada assim que o SP responde positivamente à msg3 que o cliente envia. Como estas chaves não devem vazar do ambiente seguro do enclave, é calculado um *hash* SHA-256 das chaves que a função devolve. Este método recebe como parâmetros o contexto de atestação e o tipo de chave que se pretende. No terceiro parâmetro, de saída, é colocada a chave requisitada.

- **EcallReleaseRaDhkeContext:** esta ECALL liberta o contexto do protocolo de autenticação através do método *sgx_ra_close*. Este método recebe como parâmetro o contexto que se pretende libertar quando o protocolo encerra.

No ficheiro *transactions.edl* é definida a estrutura de uma transação e três ECALLs relativas ao protocolo de transferência. O conteúdo deste ficheiro pode ser visto na listagem 4.3. Logo no início do documento define-se uma estrutura que representa as transações geradas no enclave. O cabeçalho *sgx_trts.h* permite utilizar uma função do SDK (*sgx_read_rand*) para tirar partido das instruções do processador para gerar um número aleatório verdadeiro a partir do *hardware* dentro do enclave. Este método é usado para criar os *nonces* que seguem na transações para evitar a sua repetição. Depois de importar este cabeçalho, são definidas as três chamadas ao enclave que formam a base para as transações confiáveis produzidas no cliente. Eis o significado de cada uma delas:

- **EcallGenerateEcc256KeyPair:** esta ECALL origina o par de chaves de curvas elíticas baseadas na curva NIST P-256 responsável pela autenticação das transações. É a chave pública deste par que é certificada pelo provedor de serviços e que serve de prova da autenticidade das transações de um cliente. A chave privada é usada para assinar, através do método ECDSA, as transações criadas no enclave. Para suportar as operações de curvas elíticas dentro do enclave é aberto um contexto mantido no enclave para as respetivas computações (*ecc_crypto_context*).

- **EcallGenerateSignedTransaction:** esta ECALL constrói uma nova transação e assina-a. Para isto, começa por verificar se a quantia requisitada pelo cliente é suficiente de acordo com o seu saldo armazenado no enclave. Depois, preenche os campos da estrutura da transação com o destinatário, *nonce* acabado de gerar, emissor e quantia. Por último, utiliza a função do SDK *sgx_ecdsa_sign* para efetuar uma assinatura via ECDSA da estrutura da transação.

- **EcallVerifyTransactionSignature:** esta ECALL verifica a assinatura duma transação e incrementa o saldo do enclave do cliente de acordo com o resultado da verificação efetuada. Em primeiro lugar, verifica se não é repetida através da estrutura de anti-repetição. Em segundo lugar, usa a função do SDK *sgx_ecdsa_verify* para verificar a assinatura da transação através da chave pública previamente aprovada no código não confiável do cliente. No final, incrementa o saldo do enclave do cliente se a operação de verificação sucedeu sem problemas e devolve o resultado à aplicação não confiável.

Listagem 4.3: Ficheiro *transactions.edl*

```
/* transactions.edl - transactions functionality edl file */

enclave {

    /* Transaction structure */
    struct Transaction {
    char sender[32];
    char reciever[32];
    int amount;
    char nonce[32];
    sgx_ec256_signature_t signature;
  };

  trusted {
    include "sgx_trts.h"

    public sgx_status_t EcallGenerateEcc256KeyPair(
        [out] sgx_ec256_public_t *enclave_pub_key_out);

    public sgx_status_t EcallGenerateSignedTransaction(
        [in, out] Transaction *transaction);

    public sgx_status_t EcallVerifyTransactionSignature(
        [in] Transaction *transaction,
        [in] sgx_ec256_public_t *peer_pub_key,
        [out] uint8_t *verification_result);

  };

    untrusted {

    };
};
```

4.4 Provedor de Serviços

O provedor de serviços está ininterruptamente em escuta por novos pedidos de atestação por parte de novos clientes que entram na rede. O propósito deste servidor é garantir que qualquer cliente corre o código do enclave compilado que reconhece como genuíno e impor uma política de aceitação de acordo com as informações que chegam durante os processos de atestação. É também o SP que cria os certificados das chaves públicas que os clientes autênticos anexam às suas transações como prova de que este os aprovou como enclaves corretos. No que toca ao fluxo de atestação, o provedor de serviços processa a mensagem 1, cria a mensagem 2, processa a mensagem 3 e finalmente cria a mensagem

4 que encerra o fluxo. Ao contrário dos clientes que já possuem a maior parte das ferramentas, através do SDK, para mais facilmente processarem as mensagens e criarem as respetivas respostas, o SP implementa esses mecanismos através da biblioteca OpenSSL tornando o seu código mais complexo.

Para o SP conseguir reconhecer um cliente como verdadeiro tem que recorrer ao servidor fornecido pela Intel, o IAS, que possuiu uma API especializada inteiramente para aprovar entidades de enclaves. Este serviço da Intel está disponível para registo em https://api.portal.trustedservices.intel.com/, onde se obtêm as chaves de API e o SPID. No momento do registo, escolhe-se o tipo de política de atestação, que no caso deste projeto, foi escolhida a política *linkable* por razões de segurança. Estas informações são embutidas na inicialização do programa do SP, assim como o endereço onde escuta por pedidos e as suas chaves pública e privada que necessita para atestar os enclaves dos clientes.

Para o SP ser mais eficiente, processa os pedidos das mensagens de atestação assincronamente de modo a conseguir processar múltiplos pedidos concorrentemente. Ao receber a msg1 do cliente, o provedor de serviços começa por desserializar a mesma e posteriormente verificar os valores na solicitação, gerar o seu próprio parâmetro DHKE e enviar uma consulta ao IAS para obter a lista de revogação de assinaturas (SigRL) do Intel EPID GID enviado pelo cliente. Para processar msg1 e gerar msg2, o provedor de serviços segue estas etapas:

1. Gerar uma chave de curvas elíticas aleatória usando a curva P-256. Esta chave denomina-se Gb.

2. Derivar a chave de derivação de chaves (KDK) a partir de Ga e Gb:

   a) Computar o segredo partilhado usando a chave de sessão pública do cliente, Ga, e a chave de sessão do provedor de serviços (obtida em 1.), Gb. O resultado desta operação será a coordenada x de Gab, denotada Gab_x.

   b) Converter Gab_x para a ordem de *bytes little-endian*.

   c) Executar um AES-128 CMAC na forma *little-endian* de Gab_x usando um bloco de 0x00 *bytes* para a chave.

   d) O resultado do passo anterior é o KDK. KDK é protegida em memória para impedir inspeção trivial.

3. Derivar a SMK executando um AES-128 CMAC na sequência de bytes: 0x01 || SMK || 0x00 || 0x80 || 0x00, usando o KDK como chave.

4. Definir o tipo de *quote* que é solicitado ao cliente com o valor 0x1 (*linkable*).

5. Definir o valor de KDF_ID com o valor 0x1.

6. Calcular a assinatura ECDSA de: Gb_x || Gb_y || Ga_x || Ga_y.

7. Calcular o AES-128 CMAC de: Gb || SPID || Quote_Type || KDF_ID || SigSP usando a chave SMK (derivada em 3.).

8. Consultar o IAS para obter o SigRL para o Intel EPID GID do cliente. Os EPID SigRLs são gerados pela Intel e armazenados no IAS. Estes são usados para verificar o estado de revogação da plataforma e do *Quoting Enclave*. A respetiva API para as obter é mostrada na figura 4.2.

9. Responder com a msg2 à msg1 do cliente.

A estrutura da msg2 resultante destes passos está representada na tabela 4.8.

Tabela 4.8: Estrutura da msg2

Elemento			**Tamanho**
A	Gb	Gb_x	32 *bytes*
		Gb_y	32 *bytes*
	SPID		16 *bytes*
	Quote_Type		2 *bytes*
	KDF_ID		2 *bytes*
	SigSP	$Sig(Gb\_Ga)_x$	32 *bytes*
		$Sig(Gb\_Ga)_y$	32 *bytes*
	$CMAC_{SMK}$(A)		16 *bytes*
SigRL	SigRL_Size		4 *bytes*
	SigRL_data		SigRL_size *bytes*

Os valores inteiros sem sinal (Quote_Type, KDF_ID e SigRL_Size), as coordenadas x e y de SigSP e as coordenadas x e y de Gb estão na ordem de *bytes little-endian*.

A algoritmia que suporta os passos da manipulação da msg1 e da criação da msg2 por parte do SP é apresentada no algoritmo 8.

Ao receber a msg3 do cliente, o provedor de serviços deve fazer o seguinte por ordem:

1. Verificar se Ga da msg3 corresponde a Ga da msg1.

2. Verificar $CMAC_{SMK}$(M).

3. Verificar se os primeiros 32 *bytes* dos dados do corpo do relatório do *quote* correspondem ao *hash* SHA-256 de Ga || Gb || VK. VK é derivado ao executar um AES-128 CMAC na seguinte sequência de *bytes*, usando o KDK como chave: 0x01 || "VK" || 0x00 || 0x80 || 0x00

4. Verificar a evidência de atestação fornecida pelo cliente:

   a) Extrair o *quote* da msg3.

Request		
HTTP method	GET	
HTTP resource	/attestation/v3/sigrl/{gid} *Note: No trailing slash.*	
Request body	N/A	
Request headers	**Header**	**Value**
	Ocp-Apim-Subscription-Key	Subscription Key that provides access to the API (copied as-is from the API portal).
Parameters	{gid} – Base 16-encoded representation of the EPID group ID provided by the platform, encoded as a Big Endian integer.	
Response		
HTTP status	**Status code**	**Description**
	200 OK	Operation successful.
	401 Unauthorized	Failed to authenticate or authorize request.
	404 Not Found	{gid} does not refer to a valid EPID group ID.
	500 Internal Server Error	Internal error occurred.
	503 Service Unavailable	Service is currently not able to process the request (due to a temporary overloading or maintenance). This is a temporary state – the same request can be repeated after some time.
Response headers	Request-ID	Random generated identifier for each request.
Response body	Base 64-encoded SigRL for EPID group identified by {gid} parameter. If {gid} refers to a valid EPID group but there is no SigRL for this group, then the response body shall be empty and the value of Content-Length response header shall be equal to 0. In any other case (error) the response body will be empty, HTTP status code will define the problem and Request-ID header will be returned to allow further troubleshooting.	

Figura 4.2: API do IAS para obtenção da lista de assinaturas revogadas (retirado de [26])

```
Resultado: Envio da msg2
// Extrair msg0, msg1 e chave pública do enclave
msg0 = msg0_json.extract_json();
msg1 = msg1_json.extract_json();
// Verificar se o GID estendido recebido é suportado
se msg0.extended_epid_gid != 0 então
    reply(BadRequest);
    return;
fim
// Gerar chave aleatória usando a curva P-256 (Gb)
gb = GenerateRandomEcKeyP256Curve();
// Derivar KDK da chave Ga e de Gb
kdk = DeriveKdk(gb, msg1.g_a);
// Derivar SMK de KDK
smk = Cmac128(kdk, "\x01SMK\x00\x80\x00");
// Converter chave em big-endian para little-endian
gb = KeyToSgxEc256(msg2.g);
// Definir SPID, tipo do quote e KDF_ID
msg2.spid = spid_.id;
msg2.quote_type = SGX_LINKABLE_SIGNATURE;
msg2.kdf_id = 1;
// Criar Gab
gb_ga = gb + ga;
// Calcular a assinatura ECDSA
msg2.sign_gb_ga = EcdsaSign(gb_ga, sp_private_ec_key);
// Converter assinatura para little-endian
BigEndianToLittleEndian(msg2.sign_gb_ga.x, SGX_ECP256_KEY_SIZE);
BigEndianToLittleEndian(msg2.sign_gb_ga.y, SGX_ECP256_KEY_SIZE);
// Obter lista de revogação de assinaturas solicitando o IAS
ias_connection.RetrieveSigRl(msg1.gid, sigrl, msg2.sig_rl_size);
// AES-128 CMAC da msg2 com chave SMK
msg2.mac = Cmac128(smk, msg2);
msg2_json["msg2"] = msg2;
msg0_msg1.reply(msg2);
```

Algoritmo 8: Processamento da msg1 e construção para envio da msg2

b) Enviar o *quote* ao IAS, chamando a respetiva função API para verificar evidências de atestação.

c) Validar o certificado de assinatura recebido na resposta.

d) Validar a assinatura do relatório usando o certificado de assinatura.

5. Se o *quote* for validada com êxito em 3., executa-se o seguinte:

a) Extrair o estado de atestação para o enclave.

b) Examinar a identidade do enclave (MRSIGNER), versão de segurança, ID do produto e código do enclave (MRENCLAVE).

c) Examinar o atributo *debug* e verificar que não está definido (num ambiente de produção).

d) Decidir se confia ou não no enclave.

6. Derivar as chaves de sessão, SK e MK, que devem ser usadas para transmitir mensagens futuras entre o cliente e o servidor durante a sessão. O cliente pode simplesmente chamar *sgx_ra_get_keys*, mas o servidor deriva-as manualmente executando um AES-128 CMAC nas seguintes sequências de bytes, usando o KDK como chave: MK: 0x01 || "MK" || 0x00 || 0x80 || 0x00 e SK: 0x01 || "SK" || 0x00 || 0x80 || 0x00

7. Gerar a msg4 e enviá-la para o cliente.

A verificação das evidências de atestação exige que o provedor de serviços envie o *quote* para o IAS e obtenha um relatório de atestação. Este relatório é assinado pela chave privada de assinatura de relatórios do IAS e o SP deve validar esta assinatura usando a chave pública de assinatura de relatórios do IAS.

O próprio relatório contém cabeçalhos de resposta e um *payload* que contém o estado de atestação para o enclave. Um valor de "OK" significa que o componente é de confiança. Qualquer outro valor significa que o componente não é confiável ou é confiável se determinadas ações forem tomadas (como uma atualização de *software* ou BIOS). Cabe ao SP passar estas informações para o cliente. Tudo isto depende da política que em modo de produção estiver a vigorar no SP. A API do IAS para verificação de *quotes* do IAS está exposta na figura 4.3.

Se o estado de atestação não for "OK", o IAS envia um PIB (*Platform Info Blob*) que o SP encaminha ao cliente. O cliente passa este PIB para a função *sgx_report_attestation_status* para obter informações mais detalhadas sobre o que falhou.

O formato da msg4 depende sempre das políticas internas de um SP em ambiente de produção, mas deve conter obrigatoriamente:

- Se o enclave é ou não confiável.
- O PIB caso não seja.
- O certificado da chave pública do cliente que se atestou.

Opcionalmente pode conter também:

- Outros segredos a fornecer ao enclave. Quaisquer segredos devem ser encriptados usando as chaves SK ou MK
- Tempo limite para nova reatestação do enclave
- O *timestamp* atual
- Chaves públicas de outros servidores em que o cliente precisa de confiar

Request		
HTTP method	POST	
HTTP resource	/attestation/v3/report *Note: No trailing slash.*	

Request body	Attestation Evidence Payload serialized to JSON:
	{
	"isvEnclaveQuote":"<encoded_quote>",
	"pseManifest":
	"<encoded_SGX_Platform_Service_Security_Property_Descriptor><optional>",
	"nonce":"<custom_value_passed_by_caller><optional>"
	}	

Request headers	**Header**	**Value**
	Content-Type	"application/json"
	Ocp-Apim-Subscription-Key	Subscription Key that provides access to the API (copied as-is from the API portal).
Parameters	N/A	
Response		
HTTP status code	**Status code**	**Description**
	200 OK	Operation successful.
	400 Bad Request	Invalid Attestation Evidence Payload. The client should not repeat the request without modifications.
	401 Unauthorized	Failed to authenticate or authorize request.
	500 Internal Server Error	Internal error occurred.
	503 Service Unavailable	Service is currently not able to process the request (due to a temporary overloading or maintenance). This is a temporary state – the same request can be repeated after some time.
Response headers	X-IASReport-Signature	Base 64-encoded Report Signature. This header is present only if HTTP status code is 200.
	X-IASReport-Signing-Certificate	URL encoded Attestation Report Signing Certificate Chain in PEM format (all certificates in the chain, appended to each other). This header is present only if HTTP status code is 200.
	Advisory-URL	URL to Intel® Product Security Center Advisories page that provides additional information on SGX-related security issues. IDs of advisories for specific issues that may affect the attested platform are conveyed in Advisory IDs header. This header is present only if HTTP status code is 200 and isvEnclaveQuoteStatus in Attestation Verification Report is equal to GROUP_OUT_OF_DATE or CONFIGURATION_NEEDED.
	Advisory-IDs	Comma-separated list of Advisory IDs (e.g. "INTEL-SA-00075, INTEL-SA-00076") that can be searched on a page indicated by URL included in Advisory-URL header. Advisory IDs refer to articles providing insight into SGX-related security issues that may affect attested platform. This header is present only if HTTP status code is 200 and isvEnclaveQuoteStatus in Attestation Verification Report is equal to GROUP_OUT_OF_DATE or CONFIGURATION_NEEDED.
	Request-ID	Random generated identifier for each request.
Response body	Attestation Verification Report serialized to a JSON string format: {	
	"id":"<report_id>", "timestamp":"<timestamp>", "version":<version>, "isvEnclaveQuoteStatus":"<quote_status>", "isvEnclaveQuoteBody":"<quote_body>", "revocationReason":<revocation_reason><optional>, "pseManifestStatus":"<pse_manifest_status><optional>", "pseManifestHash":"<pse_manifest_hash><optional>", "platformInfoBlob":"<platform_info_blob><optional>", "nonce":"<custom_value_passed_by_caller><optional>", "epidPseudonym":"<epid_pseudonym_for_linkable><optional>" } In case of an error during processing, the response body will be empty (an appropriate HTTP status code will define the problem and Request-ID header returned in case additional troubleshooting actions are required).	

Figura 4.3: API do IAS para obtenção do relatório de verificação de atestação (retirado de [26])

```
Resultado: Envio da msg4
// Extrair msg3
msg3 = msg3_json.extract_json();
// Assegurar que o Ga da msg3 corresponde ao da msg1
compare(msg3.g_a, msg1.g_a);
// Validar CMAC da msg3
VerifyCmac128(smk, msg3.mac);
// Verificar os primeiros 32 bytes dos dados do relatório do quote
VerifyQuoteReport(quote.report_body)
// Derivar VK
vk = Cmac128(kdk,"\x01VK\x00\x80\x00");
// Verificar estado de atestação do quote
ias_connection_.RetrieveAttestationReport(quote_base64, atestation_report, pib);
// Derivar MK e SK obtendo o seu hash SHA-256
mk = Cmac128(kdk, "x01MK\x00\x80\x00");
sk = Cmac128(kdk,"\x01SK\x00\x80\x00");
Sha256Digest(mk);
Sha256Digest(sk);
// Gerar certificado assinado pelo SP da chave pública do enclave
pub_key_cert_signed = X509(sp_private_ec_key, enclave_pub_key);
// Criar msg4
msg4 = { attestation_report_value, pib , pub_key_cert_signed };
msg4_json["msg4"] = msg4;
msg3.reply(msg4);
```

Algoritmo 9: Processamento da msg2 e construção para envio da msg4

Este processo pode ser observado programaticamente no algoritmo 9.

O SP valida se o relatório do enclave é de um enclave que reconhece. Ou seja, que o MRSIGNER corresponde à sua chave de assinatura e o *hash* MRENCLAVE corresponde ao enclave compilado que reconhece como aquele que deve ser os dos seus clientes. Isto depende sempre da política utilizada, mas outras decisões de política podem incluir examinar o ISV_SVN para impedir que *software* desatualizado/obsoleto seja bem-sucedido e garantir que o TCB não está desatualizado. Um provedor de serviços num ambiente real de produção pode permitir vários valores ISV_SVN, mas para este molde, é permitido apenas o enclave que é compilado.

Capítulo 5

Avaliação

Ao longo deste capítulo é descrita a avaliação do sistema proposto no capítulo anterior. Inicialmente, é explicada a abordagem utilizada para perceber as vantagens e as desvantagens da arquitetura. Depois, é indicado o objetivo principal dos testes aos quais o sistema foi submetido. Por fim, são apurados os resultados produzidos pela avaliação realizada de modo a tirar conclusões sobre as características do projeto.

5.1 Abordagem

No sentido de estudar detalhadamente o sistema produzido definiu-se uma estratégia para validar o que foi implementado. Desde o princípio que o intuito deste trabalho é produzir uma alternativa que simplifica os algoritmos utilizados pela *blockchain* para manter íntegras as transações. No entanto, a vertente *blockchain* desta dissertação não foi conseguida pelas dificuldades sentidas nos conceitos adjacentes ao longo do projeto, embora o processamento isolado tenha sido desenvolvido. Assim, o foco da avaliação do sistema é compreender se é mais eficiente depositar a confiança das transações no processamento isolado do cliente através das SGX, ou num servidor centralizado sem qualquer tipo de tecnologia semelhante que é o ponto essencial de confiança do sistema de transações. Ou seja, pretende-se entender se é mais rápido o cliente gerar as transações e enviá-las diretamente para os outros pares da rede *peer-to-peer* ou, se por outro lado, um servidor central a gerar as transações por eles e enviá-las a quem de direito consegue uma melhor eficiência.

Desta forma, a abordagem para avaliar o que foi feito divide-se em dois grupos de testes. No primeiro grupo, analisa-se o desempenho de um cliente com um processador com *hardware* especializado SGX. Este está pré-autenticado pelo provedor de serviços, que comunica com o IAS, para enviar uma transação gerada nesse mesmo *hardware* para

um outro cliente que já se autenticou, que se encontra na mesma rede e que verifica a transação no seu processador. No outro grupo de testes, examina-se o desempenho de um cliente que pretende enviar uma transação para outro membro da rede, mas que não tem responsabilidade na geração da transação, logo tem que efetuar um pedido ao SP para lhe criar a transação e enviá-la ao destinatário da mesma. O primeiro teste permite avaliar o desempenho de um sistema transacional distribuído e cujas transações dependem inteiramente da segurança da tecnologia do processador dos clientes. O segundo teste permite avaliar um sistema transacional tradicional, tal como um banco no mundo real, por onde passam todas as transações. Ao comparar os resultados obtidos em ambos os testes, consegue-se compreender qual o sistema transacional mais eficaz e rápido.

5.2 Testes

Idealmente, os testes indicados eram executados sobre uma infraestrutura previamente oferecida pelo projecto desta dissertação, mas dado que as ferramentas escolhidas para produzir o sistema não utilizavam *JavaScript* não foi possível integrá-lo. Dada esta incompatibilidade, a execução dos testes é inevitavelmente local o que significa que quaisquer variáveis de latência remota entre os membros da rede e o SP são insignificantes e não são abrangidas pelos testes. Por outro lado, os pedidos do SP ao IAS são reais pelo que o processo de autenticação é avaliado com sucesso e simula um cenário real. Assim, os testes focam maioritariamente a parte computacional da criação das transações ao invés do tempo que demoram a atravessar a rede. Mesmo com estes factos, as garantias do protocolo de autenticação e do protocolo transacional continuam asseguradas.

Os testes para um sistema composto pelo SP, IAS e clientes de uma rede *peer-to-peer* com processadores SGX calculam o tempo entre o instante em que um cliente envia uma transação até ao instante em que recebe a resposta de receção desta pelo cliente destinatário. Por outro lado, os testes para um sistema composto pelo SP e clientes sem SGX calculam a duração de envio de uma transação que é solicitada ao SP. Quando o cliente solicita o envio de uma transação para outro cliente ao SP incluem-se o tempo de geração desta transação no provedor, o seu envio ao respetivo destinatário, a resposta do destinatário em como a recebeu e finalmente o encaminhamento desta resposta para o cliente que solicitou a transação.

O primeiro teste é aplicável diretamente no código que foi produzido para demonstrar a arquitetura projetada. Contudo, o mesmo não acontece para o teste dois que impôs a criação de um programa de testes dedicado. As diferenças principais do segundo teste são as seguintes:

- Os clientes não possuem qualquer *hardware* para criar as transações em ambiente protegido, logo é o servidor central que fica com essa obrigação.
- O SP gera todas as transações do sistema. É a este servidor que os clientes solicitam as transações. Este assina-as e reencaminha-as ao seu destinatário.

- O IAS deixa de existir e supõe-se um mecanismo de autenticação prévio pelo qual os clientes se autenticam antes de qualquer pedido ao SP.

5.3 Resultados

O ambiente de avaliação foi a máquina Intel NUC Kit NUC8i7BEH que contém um processador Intel Core i7-8559U que suporta as Intel SGX, possui 16 GB RAM e um disco M.2 SSD 2280 com PCIe X4 com 500 GB. Cada teste foi executado três vezes e foi feita a média dos tempos que se obtiveram.

O primeiro teste calculou o tempo dos seguintes procedimentos: criar o enclave, inicializá-lo, autenticá-lo e efetuar uma transação através deste. Estas etapas estão detalhadas da seguinte maneira:

- **Criar o enclave:** tempo necessário para o cliente criar o enclave.
- **Inicializar o enclave:** tempo necessário para o processador carregar o enclave em memória controlada pelo próprio.
- **Autenticar o enclave:** duração total do processo de autenticação, desde que o cliente envia a primeira mensagem até que fecha o contexto de atestação. Esta fase simula um cenário real em que o provedor de serviços do jogo comunica com o servidor de autenticação da Intel, de resto, as latências da comunicação do cliente com o SP são insignificantes pois o teste é local nesse aspeto.
- **Efetuar uma transação:** duração total do protocolo de transferência. Tempo que medeia a entrada no enclave para o emissor da transação a criar, o envio em *localhost* da transação, e a verificação da mesma pelo destinatário, respondendo depois com a aprovação de receção da transação. Neste passo, apenas são realmente testados o tempo das computações de entrar no enclave, sair do enclave e assinar uma transação no enclave.

A duração média dos três primeiros passos é 1.7643 segundos, embora para se gerar a transação entrando e saindo do enclave o tempo médio necessário seja 141933.3 nanossegundos. Já a duração média do tempo de envio de uma transação é 1 milissegundo. No primeiro valor é contabilizado tudo o indicado acima, exceto as comunicações por REST HTTP entre o cliente e o servidor central. De resto, incluem-se o arranque do enclave, os processos criptográficos exigidos pelo protocolo de autenticação tanto no servidor como no cliente e a criptografia inerente a uma assinatura de uma transação no enclave.

Já o segundo teste calculou o tempo dos seguintes procedimentos: autenticação do cliente, solicitação da transação ao servidor, processamento deste pedido, geração da respetiva transação, envio desta para o destinatário e reencaminhamento da resposta do destinatário para o solicitante da transação. Estas etapas incluem:

- **Solicitação da transação ao SP:** Latência do pedido de uma transação ao servidor central. Esta latência não é contabilizada, pois tanto os clientes como o SP executam na mesma máquina e na mesma rede local.
- **Gerar a transação:** Processos que ocorrem para criar a transação, validá-la e assiná-la de acordo com o solicitado pelo cliente. Aqui é testado o tempo computacional que o servidor precisa para gerar transação. O servidor mantém os saldos de todos os clientes, portanto verifica se o saldo do cliente que pretende emitir uma transação permite a quantia que este pretende enviar.

A duração média do envio de uma transação foi 3 milissegundos. Neste número é contabilizado tudo o indicado acima, exceto as comunicações por REST HTTP entre os clientes e o servidor central. De resto, incluem-se todos os mecanismos criptográficos exigidos para o SP assinar e validar a transação.

Os números obtidos revelam-se bastante baixos precisamente pelo facto de não terem em conta qualquer comunicação remota que aconteceria numa ambiente real. Tirando o primeiro teste em que é contabilizado a comunicação entre SP e IAS, tudo o resto é fundamentalmente o tempo de processamento das máquinas. Em termos de escalabilidade, à priori e sem olhar aos resultados obtidos, era expectável obter um melhor resultado para o primeiro teste, porque ao centralizar os pedidos no SP o tempo de processamento aumentaria. Além disto, se as transações forem centradas numa só entidade, existem muitas operações que esta tem de efetuar para realmente produzir uma transação. Sempre que existe um pedido de transação esta tem que realizar novamente todos os passos para verificar, gerar a transação e coordenar com os respetivos emissor e destinatário.

Mesmo admitindo que a duração do protocolo de autenticação para ambos os testes é a mesma, no que toca à produção de transações, o SP acarreta maior sobrecarga quando é ele a única entidade com essa responsabilidade. Por outro lado ao comparar os resultados obtidos, pode-se analisar que ao distribuir o processamento dessas transações nos clientes da rede o tempo diminui. Tendo em conta que cada cliente gera as suas transações em ambiente seguro, e estando na mesma rede que os outros, o envio é direto. Os passos mais pesados de verificação das próprias é todo gerido pelos processadores dos clientes o que torna a carga do sistema mais distribuída. A grande vantagem desta abordagem é que os clientes necessitam apenas de garantir ao SP que são legítimos, porque a integridade e autenticidade das transferências são mantidas pelos seus enclaves. Se for o SP a produzi-las tem todo o peso computacional sobre ele.

Embora os resultados não simulem na totalidade o que seria esperado num ambiente de produção real, estes dão uma perspetiva de que ao gerarmos transações nos processadores dos clientes conseguimos agilizar o processo de criação de transações comparativamente a uma solução mais tradicional baseada num servidor central que mantém o estado. A diferença de tempos entre um teste e outro pode ser justificada pelo número de entradas no enclave por cliente ser bastante reduzido. Todos os procedimentos inerentes na abstração de um enclave (criação, entradas e saídas do mesmo) implicam um *trade-off*

de desempenho versus segurança. Para além disto, também contam os parâmetros que são copiados entre os limites do enclave e a aplicação. Mas visto que uma transação é gerada apenas com uma entrada no enclave e uma verificação de uma transação também com uma entrada, o *overhead* não é elevado. Se se admitir uma transação completamente preenchida, são apenas 164 *bytes* que o enclave copia para aplicação quando gerada e assinada, e mais 164 no destinatário para copiar a transação para o enclave para ser verificada.

Capítulo 6

Conclusão

O sistema concebido permite uma nova forma de transacionar dinheiro incumbindo a geração das transações nos elementos de uma rede *peer-to-peer*. Através das novas instruções dos processadores Intel acima da quinta geração, as Intel SGX, consegue-se garantir que o código responsável pela geração das transações não é de todo alterado e cuja integridade permanece mesmo com um sistema operativo vulnerável.

Aliando esta nova tecnologia ao mundo das transferências de dinheiro é possível produzir aplicações seguras com a confiança depositada no *hardware* dos clientes ao invés de numa entidade central. Apesar de algo recente, está em constante crescimento, com vulnerabilidades ainda a serem descobertas e cuja metodologia de programação não difere muito da atual utilizada em ambientes sem qualquer segurança, é possível construir sistemas seguros, eficientes e escaláveis.

Embora as garantias de segurança de código e dados sejam nítidas, existe ainda assim um processo de atestação prévio que é fundamental e obrigatório para conseguir utilizar esta tecnologia em ambiente de produção. Ainda que se tenham sentido dificuldades em aprender como realmente se podia utilizar esta tecnologia a favor do objetivo principal desta dissertação, no final obteve-se um protótipo que implementa as ideias principais do sistema.

A avaliação deste protótipo mostra que gerar uma transação num ambiente protegido pelo processador da máquina é mais eficiente do que fazê-lo num ambiente não controlado num servidor central. Para além disto, embora a avaliação não o prove, é esperado que a escalabilidade de um sistema que aplique esta estratégia para distribuir a responsabilidade da geração de transações seja superior a um que as controle através de uma entidade centralizada.

6.1 Trabalho futuro

O sistema desenvolvido cumpre a maior parte dos requisitos definidos mas possui aspetos relevantes que podem ser trabalhados futuramente. Atualmente, o sistema está somente preparado para situações em que uma transação é enviada e não chega ao seu destinatário. Neste caso, como o cliente emissor da transação não recebeu nenhuma resposta positiva em tempo útil acerca da sua receção, procede normalmente com o reenvio da mesma. O problema surge quando o destinatário desta transação, por algum motivo, já não pertence à rede. Assim, reenviar a transação não tem qualquer significado e é necessário algo mais do que um simples reenvio. É neste caso que uma vertente *blockchain* para o sistema se torna útil e pode ser aplicada para resolver esta situação. Desta forma, torna-se fundamental explorar como se pode utilizar uma cadeia de blocos para manter o estado das transações para evitar perdas de dinheiro quando acontecem casos excepcionais como estes.

Outro dos pontos pertinentes que pode ser melhorado no sistema é abstrair totalmente numa biblioteca o protocolo de autenticação. Qualquer aplicação que recorre às SGX para fornecer um serviço e pretende verificar a entidade e código dos seus clientes precisa sempre de executar o protocolo de autenticação que foi apresentado no capítulo 3. Deste modo, qualquer serviço teria apenas que integrar esta biblioteca e usar a respetiva API para efetuar de forma mais cómoda a atestação dos seus clientes.

A funcionalidade de selagem dos dados do enclave em disco após o fim da execução da aplicação tem uma lacuna de segurança que pode ser resolvida futuramente. Como sempre que o enclave é destruído todos os dados que armazena são apagados foram usadas as operações de selagem dos processadores Intel para os persistir em memória não volátil. Ainda que os dados estejam encriptados, estes correm o risco de ser copiados do disco, o que torna possível replicar o estado do enclave entre diferentes execuções. Assim, é possível retomar o saldo da carteira do enclave para um estado anterior, revogando qualquer transação que tenha efetuado desde então. Para resolver esta situação, existem ferramentas que o SDK da Intel fornece que podem ser exploradas para impedir o *replaying* dos estados de um enclave.

Bibliografia

[1] A. Adamski. *Overview of Intel SGX - Part 1, SGX Internals*. Disponível em: https://blog.quarkslab.com/overview-of-intel-sgx-part-1-sgx-internals.html. Jul. de 2018.

[2] A. Adamski. *Overview of Intel SGX - Part 2, SGX Externals*. Disponível em: https://blog.quarkslab.com/overview-of-intel-sgx-part-2-sgx-externals.html. Ago. de 2018.

[3] I. Anati, S. Gueron, S. Johnson e V. Scarlata. "Innovative technology for CPU based attestation and sealing". Em: *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*. Vol. 13. ACM New York, NY, USA. 2013.

[4] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell et al. "{SCONE}: Secure Linux Containers with Intel {SGX}". Em: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 689–703.

[5] L. Baird. "The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance". Em: *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep.* (2016).

[6] B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara e S. Thorne. *The Site Reliability Workbook: Practical Ways to Implement SRE*. "O'Reilly Media, Inc.", 2018. Cap. 11 - Case Study 1: Pokémon GO on GCLB.

[7] E. Brickell e J. Li. "Enhanced Privacy ID from Bilinear Pairing." Em: *IACR Cryptology ePrint Archive* 2009 (2009), p. 95.

[8] V. Buterin. "Ethereum white paper: a next generation smart contract & decentralized application platform". Em: *First version* (2014).

[9] M. Castro, B. Liskov et al. "Practical Byzantine fault tolerance". Em: *OSDI*. Vol. 99. 1999, pp. 173–186.

[10] B. Charron-Bost, F. Pedone e A. Schiper. "Replication". Em: *LNCS* 5959 (2010), pp. 19–40.

[11] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin e T. H. Lai. "Sgxpectre attacks: Leaking enclave secrets via speculative execution". Em: *arXiv preprint arXiv:1802.09085* (2018).

[12] K. Christidis e M. Devetsikiotis. "Blockchains and Smart Contracts for the Internet of Things". Em: *IEEE Access* 4 (2016), pp. 2292–2303. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2016.2566339.

[13] E. Commission. "Working Document on Trusted Computing Platforms and in particular on the work done by the Trusted Computing Group (TCG group)". Em: *Article 29 Data Protection Working Party* (jan. de 2004).

[14] G. Coulouris, J. Dollimore, T. Kindberg e G. Blair. *Distributed Systems: Concepts and Design*. 5th. USA: Addison-Wesley Publishing Company, 2011, 662, 665–668. ISBN: 0132143011, 9780132143011.

[15] J. Cowling, D. Myers, B. Liskov, R. Rodrigues e L. Shrira. "HQ replication: A hybrid quorum protocol for Byzantine fault tolerance". Em: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 177–190.

[16] C. Feijoo, J.-L. Gómez-Barroso, J.-M. Aguado e S. Ramos. "Mobile gaming: Industry challenges and policy implications". Em: *Telecommunications Policy* 36.3 (2012), pp. 212–221.

[17] P. Fips. "186-2. digital signature standard (dss)". Em: *National Institute of Standards and Technology (NIST)* 20 (2000), p. 13.

[18] M. J. Fischer, N. A. Lynch e M. S. Paterson. *Impossibility of distributed consensus with one faulty process*. Rel. téc. MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1982.

[19] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf e S. Capkun. "On the security and performance of proof of work blockchains". Em: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 3–16.

[20] G. Greenspan. "Multichain private blockchain-white paper". Em: *URl: http://www.multichain. com/download/MultiChain-White-Paper. pdf* (2015).

[21] G. Greenspan. *Smart contract showdown: Hyperledger Fabric vs MultiChain vs Ethereum vs Corda*. Dez. de 2018. URL: https://www.multichain.com/blog/2018/12/smart-contract-showdown/.

[22] T. C. Group. *TPM 1.2 Main Specification*. Mar. de 2011. URL: https://trustedcomputinggroup.org/resource/tpm-main-specification/.

[23] T. C. Group. *TPM 2.0 Library Specification*. Mar. de 2013. URL: https://trustedcomputinggroup.org/resource/tpm-library-specification/.

[24] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade e J. Del Cuvillo. "Using innovative instructions to create trustworthy software solutions." Em: *HASP@ ISCA* 11.10.1145 (2013), pp. 2487726–2488370.

[25] A. Holdings. *ARM Security Technology, Building a Secure System using TrustZone Technology*. 2009.

[26] Intel. *Attestation Service for Intel® Software Guard Extensions (Intel® SGX): API Documentation*. Disponível em: https://api.trustedservices.intel.com/documents/sgx-attestation-api-spec.pdf, versão 6.0.

[27] Intel. *Strengthen Enclave Trust With Attestation*. Disponível em: https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/attestation-services.html.

[28] S. Johnson, V. Scarlata, C. Rozas, E. Brickell e F. Mckeen. "Intel® Software Guard Extensions: EPID Provisioning and Attestation Services". Em: *White Paper* 1 (2016), pp. 1–10.

[29] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher et al. "Spectre attacks: Exploiting speculative execution". Em: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 1–19.

[30] E. M. Koruyeh, K. N. Khasawneh, C. Song e N. Abu-Ghazaleh. "Spectre returns! speculation attacks using the return stack buffer". Em: *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*. 2018.

[31] L. Lamport, R. Shostak e M. Pease. "The Byzantine generals problem". Em: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 382–401.

[32] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin et al. "Meltdown: Reading kernel memory from user space". Em: *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, pp. 973–990.

[33] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue e U. R. Savagaonkar. "Innovative instructions and software model for isolated execution." Em: *HASP@ ISCA* 10.1 (2013).

[34] J. MICHAEL, A. COHN e J. R. BUTCHER. "Blockchain Technology". Em: *The Journal* (2018).

[35] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss e F. Piessens. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX". Em: *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*. 2020.

[36] S. Nakamoto et al. "Bitcoin: A peer-to-peer electronic cash system". Em: (2008).

[37] V. Scarlata, S. Johnson, J. Beaney e P. Zmijewski. "Supporting third party attestation for intel® sgx with intel® data center attestation primitives". Em: *White Paper* (2018).

[38] L. Stone. *Bringing Pokémon GO to life on Google Cloud*. Set. de 2016. URL: https://cloud.google.com/blog/products/gcp/bringing-pokemon-go-to-life-on-google-cloud.

[39] S. Weiser e M. Werner. "Sgxio: Generic trusted i/o path for intel sgx". Em: *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. 2017, pp. 261–268.

[40] G. Wood. "Ethereum: A secure decentralised generalised transaction ledger". Em: *Ethereum project yellow paper* 151 (2014), pp. 1–32.

ANEXO I

Atestação e Selagem baseadas no CPU

Para a construção deste capítulo, foram adaptadas determinadas seções de [3] fulcrais para a compreensão dos mecanismos utilizados na implementação do sistema detalhado no capítulo 3. Neste capítulo, são discriminados os componentes necessários para gerar uma atestação baseada no processador do *software* em execução dentro de um enclave e os meios para um enclave selar/encriptar dados e exportá-los para fora (por exemplo, para memória não volátil), de tal maneira que somente o mesmo *software* seja capaz de importá-los desencriptados novamente.

I.1 Modelo de Segurança SGX

De acordo com o modelo de segurança das Intel SGX, a TCB encarregue de proteger dados secretos são o *firmware*, o *hardware* do processador e apenas o *software* no interior de um enclave. O desenvolvedor de um enclave pode usar os meios próprios das Intel SGX para selar dados e para atestação, pois asseguram as afirmações abaixo e provam a um terceiro que os segredos são protegidos de acordo com o nível de segurança pretendido:

- O Intel SGX permite instanciar enclaves capazes de solicitar uma asserção segura da plataforma/máquina/computador acerca da sua identidade.
    - O Intel SGX também permite que estes vinculem dados efémeros seus à asserção.
- O Intel SGX permite aos enclaves verificarem asserções provenientes de outras instâncias de outros enclaves na mesma plataforma/máquina/computador.
- O Intel SGX permite que uma entidade remota verifique asserções de uma instância de um enclave.

- O Intel SGX permite que uma instância de um enclave obtenha chaves vinculadas à plataforma/máquina/computador e ao mesmo.
    - O Intel SGX impede o acesso do *software* a chaves de outros enclaves.

I.2 Instruções Intel SGX

A arquitetura Intel SGX fornece as instruções de hardware *EREPORT* e *EGETKEY* para suportar atestação e para selar dados. Os portadores de dados secretos que aceitam o modelo de segurança do SGX podem confiar nestas instruções para depositar no TCB os seus segredos.

O *software* não confiável usa instruções Intel SGX para criar o ambiente do enclave e calcular uma medição criptográfica do ambiente produzido.

Para habilitar a atestação e a selagem de dados, o *hardware* fornece duas instruções adicionais:
EREPORT e *EGETKEY*. A primeira fornece uma prova sob a forma de uma estrutura que é criptograficamente vinculada ao *hardware* e para consumo pelos verificadores da atestação. A segunda fornece ao *software* de um enclave acesso às chaves *"Report"* e *"Seal"* usadas no processo de atestação e selagem, respetivamente.

I.3 Medições

A arquitetura das Intel SGX é responsável por estabelecer identidades para a atestação e a selagem de dados. Para cada enclave, fornece dois registos de medição: MRENCLAVE e MRSIGNER. O MRENCLAVE fornece uma identidade do código e dos dados de um enclave quando é compilado. O MRSIGNER fornece a identidade de uma autoridade sobre um enclave.

Estes valores são registados enquanto o enclave é construído e finalizados antes do início da execução do mesmo. Somente o TCB escreve nestes registos, a fim de garantir a precisão destas identidades nos processos de atestação e selagem.

I.3.1 MRENCLAVE - Identidade do Enclave

A "Identidade do Enclave" é o valor MRENCLAVE, um SHA-256 *digest* de um *log* interno que regista toda a atividade realizada enquanto o enclave é compilado. O *log* contem as seguintes informações:

- O conteúdo das páginas (código, dados, pilha, *heap*);
- A posição relativa das páginas no enclave;
- Quaisquer *flags* de segurança associados às páginas.

Assim que a inicialização do enclave é concluída, através da instrução EINIT, o MRENCLAVE não sofre mais atualizações. O valor final do MRENCLAVE é um SHA-256 *digest* que identifica criptograficamente o código, dados e pilha colocados dentro do enclave, a ordem e posição em que as páginas do enclave foram colocadas, e as propriedades de segurança de cada uma. Qualquer alteração em qualquer uma destas variáveis leva a um valor diferente no MRENCLAVE.

I.3.2 MRSIGNER - Identidade de Selagem

O enclave possui uma segunda identidade usada para proteger dados denominada "Identidade de Selagem". A Identidade de Selagem inclui uma "Autoridade de Selagem", um identificador do produto e um número de versão. A Autoridade de Selagem é uma entidade que assina o enclave antes de ser distribuído, geralmente pelo desenvolvedor do enclave. O desenvolvedor do enclave apresenta ao *hardware* um certificado de enclave assinado por RSA (SIGSTRUCT), visível na figura I.1, que contém o valor esperado da identidade do enclave, MRENCLAVE, e a chave pública da Autoridade de Selagem. O *hardware* verifica a assinatura no certificado, usando a chave pública contida nele e, em seguida, compara o valor do MRENCLAVE que computou com o da versão assinada. Se estas verificações forem aprovadas, um *hash* da chave pública da Autoridade de Selagem é armazenado no registo MRSIGNER. É essencial destacar que se vários enclaves forem assinados pela mesma Autoridade de Selagem todos terão o mesmo valor de MRSIGNER. Deste modo, o valor da Identidade de Selagem pode ser usado para selar dados de tal forma que os enclaves da mesma Autoridade de Selagem (por exemplo, versões diferentes do mesmo enclave) conseguem partilhar e migrar os dados que selaram.

I.4 Atestação

A atestação é o processo de demonstrar que um *software* foi instanciado corretamente numa plataforma/máquina/computador. No Intel SGX, é o mecanismo pelo qual outrem pode obter confiança de que o *software* correto está a ser executado com segurança num enclave e numa plataforma habilitada. Para conseguir isto, a arquitetura Intel SGX produz uma asserção de atestação (mostrada na Figura I.1) que transmite as seguintes informações:

- As identidades do ambiente de *software* a serem atestadas.
- Detalhes de qualquer estado não mensurável (por exemplo, o modo em que o ambiente de *software* está a executar).
- Dados que o ambiente de *software* deseja associar a si mesmo.
- Uma associação criptográfica ao TCB da plataforma que cria a asserção.

Tabela I.1: SIGSTRUCT

Seção	Nome
Header	HEADERTYPE HEADERLEN HEADERVERSION TYPE MODVENDOR DATE SIZE KEYSIZE MODULUSSIZE ENPONENTSIZE SWDEFINED RESERVED
Signature	MODULUS EXPONENT SIGNATURE
Body	MISCSELECT MISCMASK RESERVED ISVFAMILYID ATTRIBUTES ATTRIBUTEMASK ENCLAVEHASH RESERVED ISVEXTPRODID ISVPRODID ISVSVN
Buffer	RESERVED Q1 Q2

A arquitetura Intel SGX fornece um mecanismo para criar uma asserção autenticada entre dois enclaves em execução na mesma plataforma (atestação local) e outro mecanismo para estender a atestação local para fornecer asserções a terceiros fora da plataforma (atestação remota). Finalmente, para obter a máxima confiabilidade no sistema, a chave de atestação deve ser vinculada apenas a um específico TCB de uma plataforma. Se o TCB da plataforma mudar, por exemplo, devido a uma atualização de microcódigo, a chave de atestação da plataforma deve ser substituída para representar adequadamente a confiabilidade da TCB.

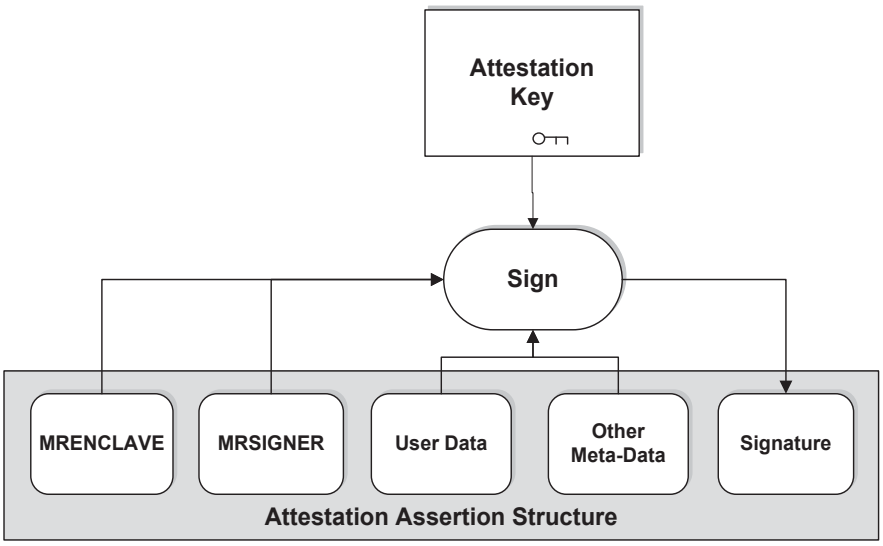


Figura I.1: Criação de Asserção de Atestação (retirada de [3])

I.5 Atestação Intra-Plataforma

Os desenvolvedores de aplicações podem desejar conceber enclaves que cooperam entre si, logo precisam de um método de autenticação entre os enclaves. Para satisfazer esta necessidade a arquitetura Intel SGX fornece a instrução EREPORT.

Quando um enclave invoca a instrução EREPORT, é criada uma estrutura assinada conhecida como REPORT. A estrutura REPORT contém as duas identidades do enclave, os atributos associados ao enclave (atributos identificam modos e outras propriedades estabelecidas durante a instrução ECREATE), a confiabilidade da TCB do *hardware* e informações adicionais que o desenvolvedor do enclave deseja transmitir ao enclave destino e uma MAC *tag*. O enclave destino é aquele que verifica o MAC no REPORT, sendo assim possível determinar se o enclave que criou o REPORT foi executado na mesma plataforma.

O MAC é fabricado com uma chave intitulada "Chave de Relatório". Conforme exposto na Tabela I.2, a Chave de Relatório é conhecida apenas pelo enclave destino e pela instrução EREPORT. O enclave destino pode obter a sua própria Chave de Relatório usando a instrução EGETKEY. A instrução EGETKEY fornece chaves aos enclaves, entre elas a Chave de Relatório, úteis para criptografia simétrica e para autenticação. O enclave destino usa a Chave de Relatório para recalcular o MAC sobre a estrutura de dados REPORT e verificar se esta foi produzida pelo enclave que atesta. A arquitetura Intel SGX usa o algoritmo AES128-CMAC para construir o MAC.

Cada estrutura REPORT também inclui um campo de 256 bits para Dados do Utilizador. Este campo vincula os dados que estão dentro do enclave à identidade do enclave (conforme expresso pelo REPORT). Este campo pode ser usado para estender o REPORT com dados auxiliares populando-o com o *hash digest* dos dados auxiliares, que é anexado

Tabela I.2: Acesso às chaves de relatório e de *seal*

Chave	Instrução	Chave associada
Chave *seal*	EGETKEY	Enclave corrente
Chave de relatório	EREPORT	Enclave destino
	EGETKEY	Enclave corrente

ao REPORT. O uso deste campo permite a um enclave construir um protocolo para formar um canal seguro com outra entidade.

Um exemplo da utilização deste campo, consiste em dois enclaves trocarem REPORTs cujos Dados do Utilizador contêm/autenticam chaves públicas *Diffie-Hellman*, geradas aleatoriamente dentro dos enclaves usando parâmetros mutuamente acordados, o que viabiliza a geração de um segredo partilhado autenticado que pode ser usado para proteger comunicações adicionais entre ambos.

I.6 Atestação Inter-Plataforma

O mecanismo de autenticação para atestação de um enclave intra-plataforma usa um sistema de chave simétrica, onde apenas o enclave que verifica a estrutura REPORT e a instrução EREPORT que a cria têm acesso à chave de autenticação. Criar uma atestação que pode ser verificada fora da plataforma requer o uso de criptografia assimétrica. O Intel SGX possui um enclave especial, chamado *Quoting Enclave*, dedicado à atestação remota. O QE verifica REPORTs de outros enclaves na plataforma usando o método de atestação intra-plataforma descrito acima e, em seguida, substitui o MAC sobre esses por uma assinatura criada com uma chave assimétrica (privada) específica da máquina. O resultado deste processo é designado de *quote*. Um *quote* inclui os seguintes dados:

- Medição do código e dados no enclave.
- Um *hash* da chave pública no certificado apresentado no momento de inicialização do enclave.
- O identificador do produto e o número da versão de segurança do enclave.
- Atributos do enclave, por exemplo, se o enclave está a ser executado em modo *debug*.
- Dados do utilizador incluídos pelo enclave na parte de dados da estrutura do relatório.
- Um bloco de assinatura dos dados acima, assinado pela chave EPID do grupo.

Os dados do enclave contidos no *quote* são apresentados ao SP remoto no final de um processo de atestação, com os quais o SP vai comparar com uma configuração confiável para decidir se deve prover o serviço ao enclave.

I.6.1 *Intel Enhanced Privacy ID*

A Intel introduziu uma extensão do esquema DAA usado pelo TPM [22] e [23] nomeado Intel *Enhanced Privacy ID* [7], usado pelo QE para assinar atestações de enclaves, dadas as preocupações de privacidade [13] levantadas na atestação utilizando esquemas de assinatura assimétrica padrão com um pequeno número de chaves ao longo da vida de um sistema. EPID é um esquema de assinatura de grupo que permite a uma plataforma assinar objetos sem identificar unicamente a plataforma ou vincular diferentes assinaturas. Deste modo, cada assinante pertence a um "grupo" e quem averigua as assinaturas usa a chave pública do grupo para verificá-las. O EPID suporta dois modos de assinaturas: um totalmente anónimo e outro pseudónimo. No modo totalmente anónimo do EPID, um quem verifica a assinatura não pode associar uma determinada assinatura a um membro específico do grupo. No modo pseudónimo, quem verifica a assinatura EPID consegue determinar se já conferiu anteriormente a plataforma que assinou.

No que toca ao SGX, a Intel define um grupo como um conjunto de processadores do mesmo tipo (por exemplo, core i3, i5 ou i7) que estão todos no mesmo TCB. Um tamanho típico para um grupo totalmente preenchido é de um milhão a alguns milhões de plataformas.

I.6.2 *Quoting Enclave*

O *Quoting Enclave* é um enclave fornecido pela Intel que verifica os relatórios que foram criados para o seu MRENCLAVE e depois os converte e assina usando uma chave assimétrica específica do dispositivo, a chave Intel EPID. O resultado deste processo é chamada de *quote* e pode ser verificado fora da plataforma. Somente o QE tem acesso à chave Intel EPID quando o sistema de enclave está operacional. Portanto, a o *quote* surge do próprio *hardware*, mas a chave nunca é exposta fora da plataforma.

I.7 Selagem

Quando um enclave é instanciado o *hardware* fornece confidencialidade e integridade aos seus dados, quando é mantido dentro dos seus limites. No entanto, quando o processo do enclave termina, este é destruído assim como todos os dados que protege. Se os dados tiverem que ser reutilizados posteriormente, o enclave deve armazenar os dados fora da sua fronteira.

A Tabela I.2 acima assinala que a instrução EGETKEY fornece o acesso a Chaves de Selagem persistentes que o *software* de enclave pode usar para encriptar e proteger a integridade dos dados. O Intel SGX não tem restrições no esquema de criptografia usado pelo enclave. A proteção de *replay* dos dados também é possível quando combinada com outro serviço oferecido pela plataforma, como contadores monotónicos.

I.8 Políticas de Selagem Intel SGX

Ao invocar a instrução EGETKEY, um enclave seleciona uma política para estabelecer que enclaves na mesma plataforma podem aceder à chave de selagem. Estas políticas são úteis para controlar a obtenção de dados confidenciais em futuras versões do enclave na mesma máquina. O Intel SGX oferece duas políticas para Chaves de Selagem: selagem para a Identidade de Enclave e selagem para a Identidade de Selagem.

Selagem para a Identidade de Enclave produz uma chave que está apenas disponível para instâncias de enclaves com exatamente a mesma Identidade de Enclave. Isto significa que somente enclaves com o mesmo *software* acedem aos segredos. Selagem para a Identidade de Selagem produz uma chave que está disponível para enclaves assinados pela mesma Autoridade de Selagem. Isto pode ser usado para permitir que enclaves mais recentes acedam aos dados armazenados por enclaves de versões anteriores.

Só uma instância decorrente de um enclave, que executa a instrução EGETKEY com a mesma política, pode recuperar a Chave de Selagem e desencriptar os dados que foram selados usando essa chave por uma instância anterior.

I.8.1 Selagem para a Identidade de Enclave

Ao selar para a Identidade de Enclave, a instrução EGETKEY produz uma chave baseada no valor de MRENCLAVE do enclave. Qualquer alteração que afete a medição do enclave produzirá uma chave diferente. Isto resulta numa chave diferente para cada enclave, fornecendo isolamento total entre enclaves. Como resultado do uso desta política, versões diferentes de um enclave têm chaves de selagem diferentes, o que impede a migração de dados *offline* entre eles.

Esta política é útil quando os dados antigos não devem ser usados depois de descoberta uma vulnerabilidade. Por exemplo, se os dados são uma credencial de autenticação, o SP pode revogar essas credenciais e fornecer novas. O acesso à credencial antiga pode ser prejudicial.

I.8.2 Selagem para a Identidade de Selagem

Ao selar para a Identidade de Selagem do enclave, a instrução EGETKEY produz uma chave baseada no valor de MRSIGNER e da versão do enclave. O MRSIGNER espelha a chave/identidade da Autoridade de Selagem que assinou o certificado do enclave.

A vantagem de selar para a Autoridade de Selagem sobre selar para a Identidade do Enclave é que possibilita a migração *offline* de dados selados entre as versões do enclave. A Autoridade de Selagem pode assinar vários enclaves e permitir que recuperem a mesma chave de selagem, por isso estes podem aceder transparentemente aos dados que foram selados por outros.

Ao selar para uma Autoridade de Selagem, o *software* mais antigo não deve ter permissão para aceder as dados criados por *software* mais recente. Isto deve acontecer quando o

motivo do lançamento do novo *software* é corrigir problemas de segurança. Para simplificar nestes casos, a Autoridade de Selagem tem a opção de prescrever um *Security Version Number* como parte da Identidade de Selagem. A instrução EGETKEY permite que o enclave especifique qual o SVN a usar ao produzir a chave de selagem, contudo apenas permitirá que indique SVNs para a sua identidade de selagem ou anteriores. Quando o enclave sela dados, tem a opção de definir o SVN mínimo dos enclave autorizados a aceder à chave de selagem. Isto protege os segredos futuros contra o acesso de *software* vulnerável antigo, porém permite uma transição contínua onde todos os segredos passados estão disponíveis após cada atualização.

O SVN não é o mesmo que o número da versão do produto. Um produto pode ter várias versões, com funcionalidades diferentes, mas com o mesmo SVN. É da responsabilidade do desenvolvedor do enclave comunicar aos seus clientes, se necessário, que versões do produto têm a mesma equivalência de segurança.

I.9 Remover Segredos da Plataforma

A arquitetura propõe um mecanismo, conhecido como *OwnerEpoch*, que permite ao proprietário da plataforma alterar todas as chaves do sistema alterando um único valor. Como o *OwnerEpoch* é incluído automaticamente ao solicitar chaves através da instrução EGETKEY, os dados que foram selados com um valor de *OwnerEpoch* específico podem ser desbloqueados se o só se o *OwnerEpoch* estiver definido com o mesmo valor.

O propósito principal deste mecanismo é permitir que um proprietário de uma plataforma negue o acesso a todos os segredos selados na plataforma, num simples e recuperável passo, antes de transferir a plataforma para outra pessoa permanentemente ou temporariamente. Antes de transferir a plataforma, o proprietário pode alterar o *OwnerEpoch* para um valor diferente usando os mecanismos fornecidos pelo OEM. No caso de uma transferência temporária, como para manutenção da plataforma, o proprietário pode restaurar o *OwnerEpoch* para o seu valor original assim que a plataforma for devolvida e restaurar o acesso aos segredos selados.

ANEXO II

Provisionamento EPID e Serviços de Atestação

Tal como no anexo I, para construir este capítulo foram adaptadas certas seções de [28] fundamentais para entender os mecanismos utilizados na implementação do sistema detalhado no capítulo 3. Neste capítulo, é descrito como a chave de atestação SGX é remotamente fornecida às plataformas habilitadas para Intel SGX, as primitivas de *hardware* usadas para suportar o processo de atestação e o IAS que torna mais fácil verificar atestações SGX através de uma API REST. Para além disto, detalha um pouco mais o Intel EPID, o algoritmo de assinatura usado pela arquitetura de atestação Intel SGX.

II.1 Atualização da TCB

A arquitetura SGX está planeada para caso certas classes de vulnerabilidades forem descobertas na mesma estas possam ser removidas com um *upgrade* à plataforma. Este processo é denominado por recuperação de TCB. É desejável nestes casos que o novo TCB seja aplicado nas atestações da plataforma. A atestação atualizada permite que um terceiro confiável verifique se a vulnerabilidade foi removida. O novo TCB será refletido nas atestações que ocorrem após a substituição da chave de atestação. Se uma plataforma individual for revogada permanentemente devido a um ataque irrecuperável a plataforma não deve receber uma nova chave de atestação.

II.2 Requerimentos da Atestação Intel

1) Fornecer uma chave de atestação de plataforma que represente o TCB do *hardware* e do *software* das Intel SGX.

2) Substituir uma chave de atestação se a plataforma for atualizada para corrigir um problema de segurança identificado com o SGX.

3) Impedir que partes comprometidas obtenham chaves de atestação adicionais.

4) Verificar se a privacidade do utilizador não é exposta pelo atestação.

II.3 Vinculação da Chave TCB

Os requisitos de atestação das Intel SGX requerem um procedimento não só para identificar os componentes da TCB, através do uso de uma chave de atestação, mas também para permitir a substituição dessa chave, caso um dos componentes precise de ser substituído.

O SGX possui dois componentes básicos que compõem a sua TCB: o *hardware* do processador e os componentes de *software* usados na atestação. O processo de criação da chave específica da TCB que une os dois componentes possui duas partes. A primeira parte atualiza o *hardware* e a segunda liga os componentes de *software* da TCB à chave da TCB do *hardware*.

II.4 Derivação da TCB do *Hardware*

Uma atualização de microcódigo pode ser lançada para remediar certos problemas de segurança na lógica do processador. Cada atualização dessas, é assinada e recebe um valor incremental que representa a revisão de segurança (SVN). Para vincular uma versão específica à plataforma, o mecanismo *Root of Key Transformation* deriva uma chave TCB com base no SVN (Figura II.1). Este é uma função pseudo-aleatória criptográfica (PRF) que impossibilita o comprometimento da chave de *hardware* original se a chave derivada vazar.

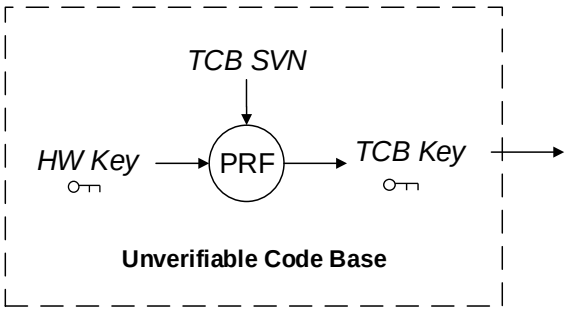


Figura II.1: Derivação de TCB básica (retirada de [28]))

Para permitir a fácil migração de dados, o *hardware* usa um mecanismo PRF cíclico (Figura II.2). Deste resultam uma sequência de chaves TCB que estão por ordem decrescente do SVN (Figura II.3). Chaves que possuem um SVN mais baixo são 'mais antigas', enquanto que as chaves de um SVN mais alto são 'futuras'. A vantagem desta abordagem é que o processador pode gerar dinamicamente chaves TCB anteriores.

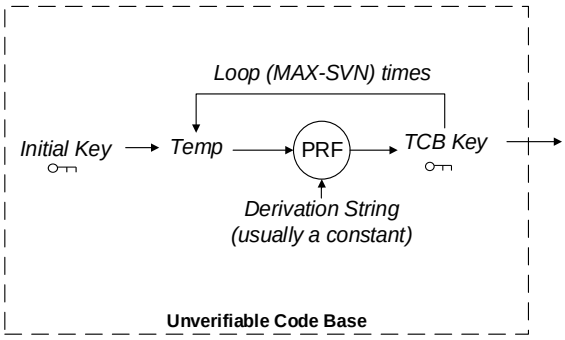


Figura II.2: Ciclo PRF TCB (retirada de [28]))

Este esquema cíclico constitui a base do processo de *Key Recovery Transformation* e ocorre no processador muito cedo durante a fase de *boot* do mesmo, antes do *fetch* da primeira instrução, onde o SVN é fixado. A *Root of Key Transformation* do *update loader* conta o número de ciclos que são executados, subtraindo ao SVN o número máximo de ciclos que ocorreu.

Figura II.3: Linha temporal das Chaves TCB (retirada de [28]))

II.5 Chaves Raiz

Cada processador habilitado para SGX contem dois valores estatisticamente únicos armazenados nos fusíveis. Estes são conhecidos como Chave de Provisionamento Raiz e Chave de Selagem Raiz. O processo *Key Recovery Transformation* opera na Chave de Provisionamento Raiz, como pode ser visto na Figura II.4. A Chave de Provisionamento Raiz é aleatoriamente criada e retida pela Intel. É a base de como o processador demonstra que é um processador Intel SGX genuíno num TCB específico. Esta Chave de Provisionamento Raiz é gerada por uma instalação/sistema de geração de chaves *offline* e é entregue à rede da fábrica da Intel.

A Chave de Selagem Raiz é criada durante a fabricação do processador e não é retida pela Intel. Como mostrado pela Figura II.4, todas as chaves exceto a Chave de Provisionamento Raiz incluem a Chave de Selagem Raiz nas suas derivações. Consequentemente, estas chaves são desconhecidas pela Intel.

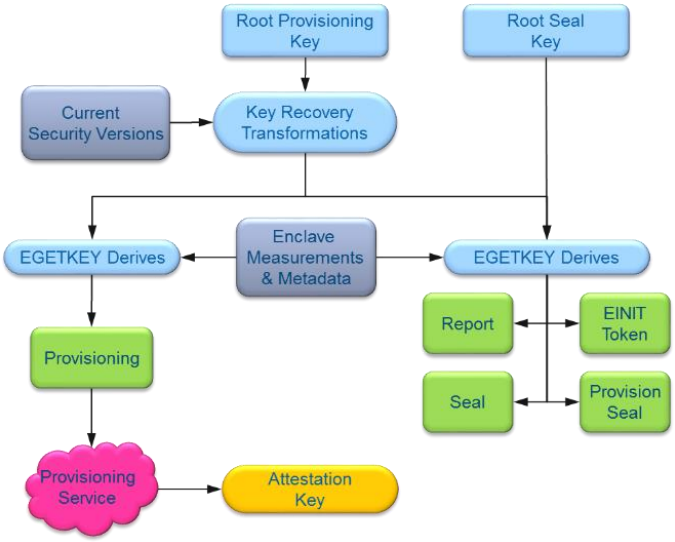


Figura II.4: Hierarquia de Chaves SGX (retirada de [28])

II.6 Derivação da TCB do *Software*

A derivação da segunda chave tem como base a chave TCB do *hardware* transformada e inclui os elementos de *software*. Esta segunda derivação ocorre na instrução EGETKEY. A Tabela II.1 identifica as propriedades do ambiente de *software* que podem entrar na segunda derivação de chave.

A instrução EGETKEY possibilita a derivação de um número de chaves diferentes (Tabela II.2) ao combinar os valores da Tabela II.1 usando outro mecanismo PRF.

A Chave de Provisionamento e a Chave de Selagem de Provisionamento não contêm o *OwnerEpoch* para permitir que a Intel identifique uma plataforma como genuína e manter seguras as chaves de atestação independentemente do seu proprietário. A Chave de Selagem de Provisionamento inclui a Chave de Selagem Raiz, o que a torna também desconhecida pela Intel.

Para cada TCB que a Intel emite chaves EPID, a instalação/sistema de geração de chaves deriva a Chave de Provisionamento que a instrução EGETKEY produz. Esta chave é usada no protocolo de provisionamento EPID com o serviço de provisionamento da Intel.

II.7 Propriedades do EPID

O EPID partilha as propriedades do aumento da privacidade do DAA e possui um modo de revogação adicional: revogação baseada em assinatura.

As assinaturas EPID são anónimas. Dada uma assinatura, ninguém, incluindo o emissor do grupo, pode determinar que chave foi usada para a estabelecer. Isto difere da maioria dos esquemas de assinatura de grupo cujas assinaturas são anónimas aos verificadores, mas o emissor das chaves tem a capacidade de abrir qualquer uma e determinar

Tabela II.1: Algumas propriedades do *Software* usadas na Derivação de Chaves (retirada de [28])

Chave	Propósito
MRENCLAVE	SHA256 *hash* da medição do enclave calculada durante a compilação do enclave
MRSIGNER	SHA256 *hash* da chave pública usada para assinar a SIGSTRUCT do enclave
CPUSVN	um conjunto de SVN de componentes de *firmware* no TCB. No caso em que mais do que um componente atualizável está incluído no TCB
ISVSVN	o SVN do componente de *software* no TCB atribuído pelo assinante do enclave através da SIGSTRUCT
ISVPRODID	um identificador do produto, designado pelo assinante do enclave, através da SIGSTRUCT, usado para dividir o espaço de chaves
OwnerEpoch	um valor fornecido pela plataforma, criado quando um novo proprietário toma posse da plataforma

que chave foi usada para a criar.

II.8 Modos de Assinatura EPID

Embora o EPID possua dois modos de assinatura com diferentes propriedades, para qualquer assinatura EPID é escolhida uma base. Se a base selecionada for diferente para duas assinaturas, estas são *unlinkable*, assim, ninguém pode determinar se ambas foram geradas pela mesma chave ou por chaves diferentes. Porém, se a base for a mesma para para duas assinaturas, é fácil determinar se ambas foram geradas pela mesma chave ou não. Não é possível determinar qual a chave que gerou as assinaturas, apenas se é ou não a mesma.

Os dois modos de assinatura EPID são o modo de base aleatória e o modo baseado em nome. No primeiro, é escolhida uma base aleatória diferente para cada assinatura tornando-as *unlinkable*. No segundo, melhor para proteger contra comprometimentos, é escolhido um nome para a base a ser usada na assinatura. Neste modo, duas assinaturas geradas com o mesmo nome são *linkable*, todavia duas assinaturas geradas com nomes diferentes não são *linkable*.

Tabela II.2: Chaves SGX (blocos verdes na Figura II.4) (retirada de [28])

Chave	Propósito
EINIT Token	chave de criação do *EINIT Token*
Report	chave de verificação usada na instrução EREPORT
Seal	Protege os segredos do enclave que necessitam de ser expostos fora deste para persistência a longo prazo
Provision Seal	Chave usada pelo enclave para proteger as chaves de atestação para persistência a longo prazo fora do enclave
Provisioning	Chave usada pelo enclave no protocolo de provisionamento para provar que a plataforma está no TCB que reivindica

Para entender o porquê do modo baseado em nome ser preferível, considere-se um banco que usa atestação para o registo de *tokens* de autenticação. Suponha-se que uma chave EPID está comprometida e um adversário cria um *malware* que a usa. O adversário pode levar muitos utilizadores a usar o *malware* durante o registo sem se aperceberem. Se o modo de base aleatória for empregue, o banco não sabe que todas as atestações foram geradas a partir de uma única chave EPID devido à propriedade *unlinkable* deste modo. Por outro lado, se o banco usar o modo baseado em nome for usado, pode perceber que muitas contas estão a ser registadas com a mesma chave. Em seguida, o banco deixa de aceitar atestações com essa chave e notifica os clientes que usaram a chave da infeção com *malware* e que devem limpar as suas plataformas e registar-se novamente.

II.9 Modos de Revogação EPID

O EPID suporta quatro modos diferentes de revogação:

- **Revogação de chave privada:** Se a Intel receber uma chave privada EPID em claro, esta pode ser colocada numa lista de revogação que é processada pelo serviço de atestação durante a verificação de uma assinatura. Como a chave privada deve existir apenas sobre proteção SGX e numa única plataforma deve ser claramente revogada.
- **Revogação local de verificador:** Com os algoritmos tradicionais de assinatura (por exemplo, RSA ou ECDSA), se uma parte confiável perceber que o comportamento de uma chave específica indica que foi comprometida, a parte confiável pode revogar a

chave localmente usando um identificador de chave no certificado de chave pública (por exemplo, Emissor e Número de Série ou *hash* da chave pública). Quando um verificador usa EPID com o modo baseado em nome, este mesmo tipo de revogação local está disponível. Isto é possível, porque neste modo as assinaturas são *linked*.

- **Revogação baseada em assinatura:** Com os algoritmos tradicionais de assinatura (por exemplo, RSA ou ECDSA), se uma parte confiável perceber que o comportamento de uma chave específica indica que foi comprometida, a parte confiável pode fornecer essa evidência a uma autoridade de revogação e se esta estiver convencida de que a chave foi realmente comprometida, pode adicionar o certificado correspondente à lista de revogações. Isto assegura que é efetivamente revogado para todas as partes confiantes, não apenas para a que constatou o comportamento anormal.

  O mecanismo EPID correspondente é conhecido como revogação de assinatura e está disponível se um terceiro confiável usar o modo de base aleatória ou o modo baseado em nome. O método de verificação de revogação para o EPID difere dos algoritmos tradicionais: quando um terceiro confiável noticia um comportamento estranho vinculado a uma assinatura, pode informar uma autoridade de revogação e esta pode determinar se há evidências suficientes para revogar a chave que gerou a assinatura. Se o fizer, coloca a assinatura numa lista de revogação, o que faz com que qualquer verificador que esteja a usar essa lista de revogação jamais aceite uma assinatura dessa chave.

  Uma plataforma de atestação ao criar uma assinatura, calcula uma prova não revogada para cada item na lista de revogação de assinaturas (SigRL). A prova não revogada é um conjunto de valores matemáticos que demonstra ao verificador que uma assinatura no SigRL não foi gerada pela plataforma de atestação.

- **Revogação baseada em grupo:** Permite a um terceiro confiável identificar que um grupo não está atualizado por algum motivo, por exemplo a chave emissora mestra do grupo EPID foi comprometida.

II.10 Serviços das Infraestruturas SGX

A Figura II.5 ilustra amplamente como o sistema de provisionamento de chaves e os serviços que a Intel fornece se integram entre si. A instalação *offline* de geração de chaves é responsável pela produção da chave de provisionamento raiz aleatória. Esta é transmitida com segurança ao sistema de fabrico de grande volume que integra a chave de provisionamento raiz.

A instalação *offline* de geração de chaves também emite chaves de provisionamento específicas da TCB derivadas e chaves mestras EPID para o servidor de provisionamento. Isto é realizado na criação de novos processadores e durante a recuperação da TCB. A instalação também permite que incidentes no serviço de provisionamento sejam tratados

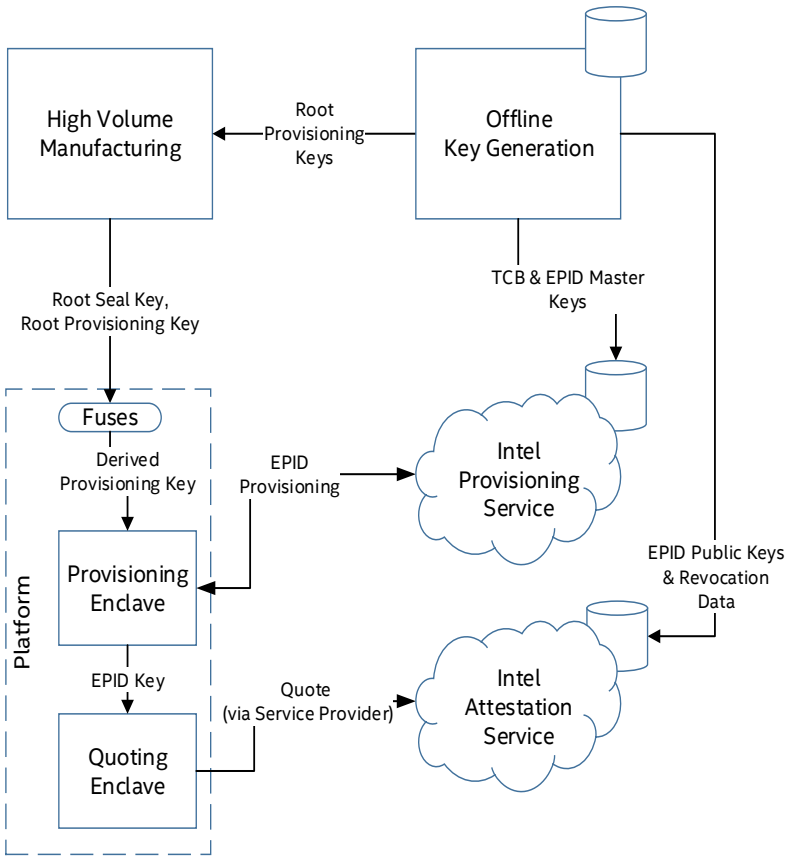


Figura II.5: Serviços das Infraestruturas SGX (retirada de [28])

através de um processo de recuperação da TCB. Finalmente, as chaves públicas EPID e as informações de revogação são emitidas para o serviço de verificação de atestação Intel (IAS).

II.11 Chaves e Dados de Revogação

Além de gerar a chave de provisionamento raiz, a instalação *offline* de geração de chaves é responsável por criar os dados que suportam a arquitetura de atestação EPID. A Figura II.6 mostra como todos os diferentes elementos do sistema EPID se relacionam.

- **Chave de Assinatura Intel (ISK):** é a principal chave de assinatura usada pela Intel para autenticar os objetos que produz e é Semelhante à chave raiz de uma autoridade de certificação.
- **Certificado de Chave de Assinatura Intel:** contém a chave pública de verificação de assinatura emparelhada com a chave de assinatura Intel e é similar ao certificado raiz de uma autoridade de certificação.
- **Chave Mestra de Grupo EPID:** cada grupo tem uma chave mestra secreta usada na criação da chave pública do grupo e no processo de entrada de um membro no

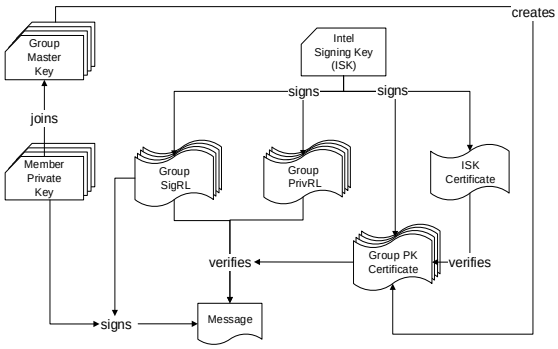


Figura II.6: Elementos da Infraestrutura EPID (retirada de [28])

grupo.

- **Chave Privada de Membro EPID:** cada membro de um grupo EPID possui a sua própria chave privada que é desconhecida pela Intel. A chave é gerada aleatoriamente através de protocolo de entrada às cegas entre o enclave de provisionamento na plataforma e a Intel.
- **Lista de Revogação de Chaves Privadas EPID (PrivRL):** lista de chaves privadas EPID que a Intel sabe que estão comprometidas, porque a Intel as recebeu.
- **Lista de Revogação de Assinaturas EPID (SigRL):** lista de assinaturas EPID que são assinadas por chaves EPID e que são suspeitas de estarem comprometidas.
- **Certificado de Chave Pública de Grupo EPID:** forma certificada da Chave Pública de Grupo EPID, assinado pela chave de assinatura Intel.

II.12 Cenários de Provisionamento

Para suportar os requisitos para atestação, o serviço de provisionamento EPID da Intel implementa três cenários:

1° **Primeira geração de chave:** Quando o *software* SGX é implantado pela primeira vez, a plataforma precisa de estabelecer uma chave de atestação. Para a estabelecer, a plataforma demonstra que é um *hardware* genuíno da Intel em execução no TCB que alega e depois entra num grupo EPID selecionado pelo servidor de provisionamento EPID. A plataforma também encripta uma cópia da sua chave EPID privada para *backup* usando a chave de selagem de provisionamento. A chave de selagem de provisionamento é conhecida meramente pela plataforma.

2° **Nova geração de chave após atualização da TCB:** Caso a plataforma sofra uma recuperação da TCB (por exemplo, recebendo uma atualização do *BIOS*), a plataforma pode solicitar uma nova chave EPID para o TCB atualizado. Antes de permitir que

a plataforma ingresse num novo grupo EPID, o processo de provisionamento necessita de garantir que qualquer chave emitida anteriormente para a plataforma não foi revogada.

3° **Recuperando uma chave anterior:** Se a plataforma perder a sua chave de atestação, o processo de recuperação de *backup* reabilita a capacidade da plataforma realizar atestações. Ao fornecer este serviço de recuperação garante-se que todas as plataformas têm acesso às chaves anteriores para demonstrarem que não foram revogadas anteriormente.

II.13 Fluxo de Provisionamento

O protocolo do serviço de provisionamento EPID consiste em quatro mensagens (Figura II.7). O servidor de provisionamento possui dois componentes principais: um motor de protocolo que lida com o fluxo principal das mensagens e um ambiente de processamento seguro que verifica as provas criptográficas e protege a chave EPID mestra.

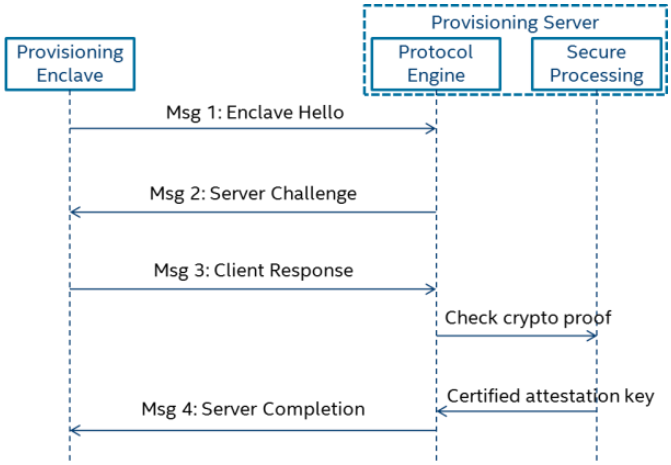


Figura II.7: Fluxo de Provisionamento (retirada de [28])

A primeira mensagem contém um PPID (*Platform Provisioning ID*), derivado da primeira chave de provisionamento disponível para o enclave de provisionamento, e uma versão TCB. Esta mensagem é encriptada usando a chave pública do servidor de provisionamento da Intel e, para manter a privacidade, apenas os enclaves autorizados podem aceder à sua chave de provisionamento derivada.

Em resposta à primeira mensagem (mensagem 2), o servidor de provisionamento usa o PPID para determinar se a plataforma já foi provisionada. Se ainda não foi, o servidor de provisionamento seleciona um grupo para a plataforma ingressar e retorna a chave pública do grupo e um desafio de *liveness* (por exemplo, um *nonce*).

Na mensagem 3 o enclave de provisionamento demonstra que está a executar dentro de um enclave numa plataforma Intel com um TCB específico.

De seguida, o enclave de provisionamento efetua o protocolo EPID de entrada às cegas com a Intel, incluindo o desafio de*liveness* emitido na mensagem 2. Ao concluir este protocolo, o enclave de provisionamento tem uma chave EPID privada e a Intel não sabe qual. Como parte deste fluxo, a plataforma também efetua uma cópia *backup* encriptada da chave EPID de membro privada. Esta encriptação é feita com a chave de selagem de provisionamento desconhecida pela Intel.

Antes de gerar a mensagem 4, a prova de TCB da plataforma e o protocolo de entrada são verificados e a chave de membro certificada. Como estas operações são muito sensíveis em termos segurança, o seu processamento ocorre dentro de um ambiente seguro. Os resultados do processo de certificação vão na mensagem 4 juntamente com uma cópia do *backup* da chave encriptada.

O protocolo do serviço de provisionamento EPID impede entidades comprometidas de obterem chaves de certificação, através da aplicação da lista de revogação atual antes de emitir uma nova chave. Se a plataforma já foi provisionada com uma chave de atestação, a mensagem 2 incluirá a lista de revogação atual, que deve ser satisfeita via provas de não-revogação na mensagem 3.

II.14 Serviço de Atestação da Intel

A Intel fornece dois serviços, um serviço de "*development*" para os desenvolvedores testarem os seus serviços e um serviço de "produção" para os provedores de serviços usarem no modo de produção.

Para utilizar o IAS para atestação remota, é preciso obter chaves de API do serviço de atestação Intel SGX utilizando EPID. Pode-se optar por uma política de atestação *linkable* ou *unlinkable*(seção II.8). É fornecido um identificador de SP com as respetivas chaves de API. Este identificador é usado pelo SP ao comunicar com o IAS.

O IAS faculta duas interfaces [26]:

- **GetSigRL[GID]** → devolve a lista de revogação de assinaturas (SigRL) para o grupo EPID identificado por GID (identificador de grupo EPID).
- **VerifyQuote[QUOTE]** → retorna o resultado da verificação da assinatura EPID na estrutura do *quote*.

A interface **GetSigRL** é útil para que terceiros confiáveis que solicitam atestações possam obter facilmente a lista de revogação mais recente do grupo EPID da plataforma com a qual comunicam.

II.15 Verificação do *Quote*

Quando um *quote* é submetido para verificação o IAS executa os seguinte passos:

1) Desencripta o *quote* se necessário.

2) Verifica se o QUOTE.SPID é o mesmo SPID do utilizador que solicita a verificação (apenas para assinaturas *linkable*).

3) Verifica se o QUOTE.GID é um grupo atualizado.

4) Obtém do armazenamento a PrivRL, a SigRL e a chave pública EPID para o QUOTE.GID.

5) Verifica se a porção do *quote* respetiva à assinatura é assinada por um membro do grupo.

6) Verifica se a assinatura não foi produzida por uma chave revogada.

7) Verifica se a parte não revogada da assinatura usou uma SigRL atualizada e verifica se cada membro da SigRL é uma entrada correta.

8) Devolve o resultado da operação ao utilizador do serviço.