Engenharia de Segurança

Universidade do Minho

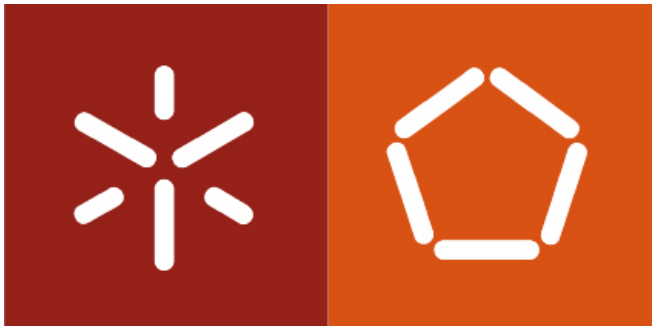
Mestrado Integrado em Engenharia Informática

# Projeto em Identificação *mobile*

## mDL (*mobile Driving License*)

*Autores:*
A77531 - Daniel Maia
A78034 - Diogo Costa
A77364 - Mafalda Nunes

19 de Junho de 2019



**Resumo**

O presente trabalho aborda a temática da identificação *mobile*, mais especificamente da *mobile Driving License* (`mDL`).

Este documento divide-se em duas partes fundamentais, sendo estas a análise do ISO de desmaterialização da Carta de Condução e a implementação da estrutura que dá suporte à `mDL`, sem perder de vista os algoritmos, as primitivas criptográficas e os *workflows* que garantem a segurança do `mDL`.

# Conteúdo

# 1 Introdução

Este projeto é desenvolvido no âmbito da unidade curricular de Engenharia de Segurança, do Mestrado Integrado em Engenharia Informática, da Universidade do Minho.

Um dos principais objetivos deste trabalho é a investigação e análise do *standard* ISO de desmaterialização da Carta de Condução, mais especificamente o "`ISO/IEC CD 18013-5 Information technology – Personal identification – ISO compliant driving licence – Part 5: Mobile driving licence application (mDL)`". Pretende-se dar especial atenção à estrutura de dados requerida e aos vários algoritmos, primitivas criptográficas e *workflows* que garantem a segurança da mDL. Por fim, deverá apresentar-se uma implementação da mDL, de acordo com o ISO, através da utilização de bibliotecas *open-source*.

De facto, esta desmaterialização de documentos, que se baseia em técnicas e algoritmos criptográficos, torna possível o acesso aos mesmos através de dispositivos móveis, que são comummente utilizados na atualidade. Assim, começa a surgir a tendência de substituir os documentos de identificação, como hoje os conhecemos (em papel ou *smartcard*), por documentos desmaterializados.

# 2 Contextualização

O *standard* ISO/IEC (*International Organization for Standardization / International Electrotechnical Commission*) 18013 é caracterizado pelo título geral `Personal Identification – ISO Compliant Driving Licence` e é constituído pelas seguintes partes:

- **Parte 1** – `Physical Characteristics and Basic Data Set` [1]: descreve as características físicas, o conjunto básico de elementos de dados, o *layout* visual e as capacidades de segurança física (recursos legíveis pelo ser humano) de uma *ISO-compliant driving licence* (IDL);
- **Parte 2** – `Machine-Readable Technologies` [2]: descreve as tecnologias, legíveis por máquina, que podem ser utilizadas por este *standard*, incluindo a estrutura de dados lógica e o mapeamento de dados por cada tecnologia;
- **Parte 3** – `Access Control, Authentication and Integrity Validation` [3]: descreve as capacidades de segurança eletrónica que podem incorporar este *standard*, incluindo mecanismos para controlo de acesso aos dados, verificação da origem de uma IDL e confirmação da integridade dos dados;
- **Parte 4** – `Test Methods`: descreve métodos de teste que podem ser utilizados para determinar se uma IDL está de acordo com os requisitos das tecnologias legíveis por máquinas especificadas na parte 2 e com as capacidades de segurança eletrónica especificadas na parte 3.

Este *standard* cria uma base comum para a utilização internacional e reconhecimento mútuo da IDL, sem impedir que países ou estados apliquem as suas regras de privacidade e que autoridades nacionais/comunitárias/regionais de trânsito tratem das suas necessidades específicas.

A **Parte 5** do ISO/IEC 18013 – `Mobile Driving Licence` [4] – pretende estabelecer um *standard* de especificações de interface para a implementação de cartas de condução associadas a dispositivos móveis (*Mobile Driving License* - mDL). Assim, esta parte descreve a interface e requisitos físicos e funcionais associados, que possibilitam a utilização de dispositivos móveis pelo titular da carta de condução, para a fornecer a um verificador, facilitando o acesso do mesmo a informação da carta de condução.

Neste contexto, considera-se que dispositivos móveis são os dispositivos eletrónicos com interface de utilizador e a capacidade de armazenar informação da mDL e de a partilhar com um leitor, após instrução do titular – *smartphones*, *wearables*, entre outros. Um leitor mDL é um dispositivo portátil ou computador, que pode trocar dados com uma mDL, enquanto que o titular da mDL é o indivíduo para quem a mDL é emitida, isto é, o titular legítimo dos privilégios de condução refletidos na mDL.

O objetivo do ISO/IEC 18013-5 é permitir que verificadores não associados à autoridade de emissão da mDL, como outras autoridades de emissão ou entidades verificadoras de outros países, ganhem acesso à informação para a qual o titular da mDL providenciar consentimento, conseguindo autenticá-la. Para o conjunto de informações disponibilizado pelo titular da mDL, estas entidades deverão poder:

1. Utilizar uma máquina para obter a informação da mDL;
2. Estabelecer a conexão entre a mDL e o seu titular, com um grau aceitável de confiança;
3. Autenticar a origem da informação da mDL;
4. Verificar a integridade da informação da mDL.

Salienta-se a utilidade do titular poder aceder e facultar dados da sua carta de condução através de um dispositivo móvel, sendo este tipo de dispositivos muito utilizado atualmente. Outra vantagem das mDL em relação às cartas de condução físicas é a capacidade de atualizar informação com mais frequência e autenticá-la com um nível de confiança superior.

Existem três interfaces fulcrais para esta parte do *standard*, que são explicadas de seguida:

1. Interface entre a mDL e a autoridade emissora, que permite controlar, entre outros, como a mDL é fornecida e como são efetuadas atualizações. Esta interface não é o foco desta parte do ISO/IEC 18013, uma vez que a interoperabilidade entre autoridades emissoras não é requerida para as funcionalidades pretendidas.
2. Interface entre a mDL e o leitor, que tem de funcionar em tempo real e é descrita na parte 5 do ISO/IEC 18013.
3. Interface entre a autoridade emissora e a entidade de verificação, que facilita a troca de informação requerida para permitir a um leitor confirmar a autenticidade da informação da mDL e, em alguns casos, ler alguma informação da mesma. Esta interface é estabelecida preferencialmente entre a entidade verificadora e a autoridade emissora (diretamente ou através de intermediários), em vez de diretamente entre o leitor e a autoridade de emissão. Para além disso, não precisa de funcionar em tempo real e pode ser usada pela própria autoridade emissora, em leitores sob o seu controlo. Esta interface é descrita nesta parte do ISO/IEC 18013.

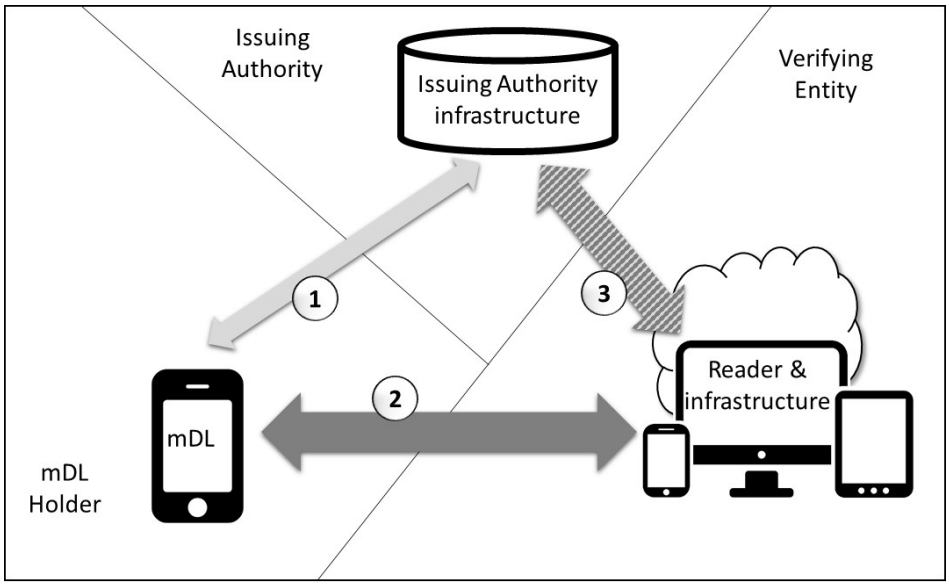



Figura 1: Ecossistema mDL, incluindo as interfaces associadas

# 3 Requisitos

Os requisitos funcionais abrangidos por esta parte do *standard* para a solução do mDL incluem:

- Capcidade de funcionar durante verificação num ambiente *offline* (leitor mDL *offline* e mDL *offline*).
- Capcidade de funcionar durante verificação num ambiente *online* (leitor mDL *online* e mDL *online*).
- Mecanismos ou uma arquitetura que permite a partes interessadas na mDL (titular, aplicador da lei ou entidade privada) estabelecer confiança na informação providenciada pela mDL, isto é, ter garantias de que a mDL foi emitida pela alegada autoridade de emissão e que informação não foi alterada.
- Confirmar a ligação entre uma mDL e um titular de mDL.
- Transmitir privilégios de condução.
- Permitir a leitura de informação entre autoridades emissoras.
- Permitir que um titular de mDL autorize a libertação de informação especificamente selecionada da mDL para um leitor mDL.

Existem ainda requisitos técnicos relativos à interface entre uma mDL e um leitor mDL, que são especificados nesta parte do ISO/IEC 18013, nomeadamente:

- Estrutura de dados lógicos com as informações da mDL, quando transferidas entre uma mDL e um leitor mDL, deve respeitar os seguintes aspetos:

  - Elementos de dados considerados no ISO/IEC 18013-2:
    * DG1: elementos de texto (obrigatório).
    * DG2: detalhes do titular da licença (opcional).
    * DG3: detalhes da autoridade de emissora (opcional).
    * DG4: imagem do retrato do titular da licença (opcional).
    * DG5: assinatura / imagem de marca habitual (opcional).
    * DG6: modelo biométrico facial (opcional).
    * DG7: modelo biométrica do dedo (opcional).
    * DG8: modelo biométrico da íris (opcional).
    * DG9: outro modelo biométrico (opcional).
    * DG10: reservado para uso futuro.
    * DG11: dados domésticos (opcional).
  - Elementos de dados adicionais:
    * Inclusão obrigatória da imagem facial do titular.
    * Elementos de dados adicionais para "Up to Date info".
    * Identificador adicional que indica o fator de forma.
    * Grupos de dados mDL, utilizados para transferência de informação seletiva (inclui novos elementos de dados).

- Protocolo de comunicação para troca de dados mDL, entre uma mDL e um leitor:
  - Camada de transmissão:
    * ISO/IEC 14443 e/ou ISO/IEC 18092 (NFC).
    * Interface visual (câmara).
    * Wi-Fi *Aware*.
    * Internet.
    * *Bluetooth Low Energy* (BLE).
  - Camada de apresentação:
    * Comandos ISO/IEC 7816-4 e ISO/IEC 7816-8 (Parte 2 e 3) para o equivalente a *Standard Encoding* para mDL.
    * Códigos de barras 2D (para estabelecimento de conexão entre dispositivos e o equivalente a *Compact Encoding* para mDL, na transferência de dados da mDL).

- Mecanismos de proteção de dados para serem aplicados, tendo em conta o ISO/IEC 18013-3 - preservar confidencialidade, integridade e autenticação de dados mDL.

Especificam-se ainda alguns requisitos funcionais relativos a uma aplicação de leitores de mDL, para assegurar a verificação fiável de uma mDL:

- Disponibilidade de verificação de dados (e.g. certificados digitais) de autoridades emissoras, incluindo a definição do modelo de confiança utilizado para uma mDL.

- Sequência de leitura para dados de uma mDL.

- Sequência de verificação para dados de uma mDL.

Assim sendo, todos estes requisitos são necessários para assegurar a competência do sistema.

# 4 Estrutura de Dados Lógica

A estrutura de dados mDL é codificada como um conjunto de objetos de dados BER-TLV.

As *Basic Encoding Rules* (BER) especificam o formato da estrutura de dados, recorrendo a um método de *enconding type-length-value* (TLV). A codificação de dados consiste em quatro elementos que se apresentam na seguinte ordem:

1. O octeto *Type*, que distingue os dados de outros membros.
2. O octeto *Length*, que define o número de bytes que constituem o objeto.
3. O octeto *Value*, que contém o conteúdo do elemento de dados.

Deste modo, a estrutura mDL pode ser apresentada em dois formatos: *standard encoding* e *compact encoding*.

## 4.1 *Standard encoding*

O *standard encoding* do mDL é constituído por três componentes primários: A estrutura de ficheiros, o conjunto de comandos e os grupos de dados que o constituem.

### 4.1.1 Estrutura de ficheiros

A estrutura de dados lógica do mDL é constituída por um conjunto de ficheiros elementares, cada um deles contendo um ou mais grupos de dados. Cada um destes pode ser classificado como obrigatório, opcional ou condicional (dependendo do suporte providenciado) na implementação do mDL. Relativamente à permissão de acesso a um dado ficheiro, é necessário indicar se o consentimento explícito é requerido do titular mDL.

Na tabela 1 são identificados os vários conjuntos de ficheiros elementares que constituem a estrutura de dados lógica do mDL.

Tabela 1: Conjunto de ficheiros elementares.

| **Ficheiro Elementar** | **Presença** | **Consentimento** |
|---|---|---|
| Data group 1 | Obrigatória | Explícito |
| Data group 2-4 | Opcional | Explícito |
| Data group 5 | Opcional<br>(Não recomendado) | Explícito |
| Data group 6 | Obrigatória | |
| Data group 7-9 | Opcional | Explícito |
| Data group 10 | Obrigatória | |
| Data group 11 | Opcional | Explícito |
| Data group 13 | Condicional<br>(Obrigatória se Active Authentication é suportada) | |
| Data group 14 | Condicional<br>(Obrigatória se autenticação PACE e/ou Chip é suportada) | |
| Data group 32-127 | Opcional | Explícito |
| EF.COM | Obrigatória | |
| EF.SOD | Obrigatória | |
| EF.CardAccess | Condicional<br>(Obrigatória se PACE é suportada) | |
| EF.GroupAccess | Obrigatória | |

#### 4.1.2 Comandos

Os comandos de uma aplicação mDL cumprem a norma ISO/IEC 18013-2. Cada comando toma a forma de uma mensagem que será transmitida a um recipiente, sendo constituída por um cabeçalho e um corpo.

O cabeçalho é constituído por quatro *bytes*, cada um dos quais indicando um campo, presentes na seguinte ordem:

- O *byte Class* (CLA) que, como o nome indica, especifica a classe, interindústria ou proprietária, do comando a executar. Indica também, caso se trate de um comando de classe interindústria, se se pretende executar *chaining* de comandos e respostas (e.g. transmissão de uma *string* demasiado longa para um único comando). Adicionalmente, indica se se pretende utilizar um canal seguro para a transmissão de dados e o respetivo formato. Por fim, é indicado o canal lógico sobre o qual a transmissão será efetuada.

- O *byte Instruction* (INS), que especifica exatamente qual comando será processado. Existe uma variedade de comandos providenciados pela norma ISO/IEC 18013-2. Para além dos comandos detalhados na norma, é especificado um comando adicional *UPDATE BINARY*, que atualiza o ficheiro EF.GroupAccess.

- Os *bytes Parameter* 1 e 2 (P1 e P2), que indicam controlos e opções para o processamento do comando.

### 4.1.3 Grupos de dados

Os dados mDL são organizado em 11 grupos de dados, de acordo com a norma ISO/IEC 18013-2, com algumas alterações.

O primeiro grupo de dados (DG 1) é responsável por guardar o conjunto mínimo de dados essenciais para identificação internacional, com a exceção da assinatura e foto do indivíduo. Os grupos de dados 6 e 10, tornam-se obrigatórios na implementação do mDL.

Tabela 2: Grupos de dados do mDL.

| Ficheiro Elementar | Conteúdo |
|---|---|
| Data group 1 | Elementos obrigatórios |
| Data group 2 | Detalhes do titular |
| Data group 3 | Detalhes da autoridade emissora |
| Data group 4 | Foto do titular |
| Data group 5 | Assinatura |
| Data group 6 | Biométrica da face |
| Data group 7 | Biométrica do dedo |
| Data group 8 | Biométrica da íris |
| Data group 9 | Outras biométricas |
| Data group 10 | Dados mDL obrigatórios |
| Data group 11 | Dados domésticos |

Para além destes, são introduzidos os grupos de dados opcionais 32 a 127, que permitirão a autorização seletiva de informação mDL para ser fornecida ao leitor. Quaisquer destes grupos que contenha dados é introduzido no elemento EF.SOD. É incluído também o elemento EF.GroupAccess, que contém informação sobre que grupos de dados são disponibilizados ao leitor mDL.

Por motivos de simplicidade, e tendo em conta que se pretende implementar apenas os campos obrigatórios do mDL, serão apenas aprofundadas as constituições dos elementos DG1, DG6, DG10, EF.COM, EF.SOD e EF.GroupAcess.

### 4.1.4 Data group 1

Este ficheiro elementar contém os dados demográficos do titular, bem como as categorias de veículo para os quais este tem qualificações e é identificado pela *tag* `'61'` e o identificador `'01'`, seguido pelo comprimento total desto grupo. Como a totalidade da informação contida neste é obrigatória e encontra-se numa ordem fixa, a informação demográfica encontra-se concatenada num único objeto com a *tag* `'5F1F'` seguida do comprimento da mesma.

De tal modo, sabendo que cada campo de tamanho variável é precedido pelo respetivo valor de comprimento, os dados demográficos seguem a seguinte estrutura:

Tabela 3: Dados armazenados no DG1.

| Nome | Tamanho Variável/ Fixo | Formato | Exemplo |
|---|---|---|---|
| Nome Próprio | Variável | Até 36 letras e/ou símbolos | Smithe-Williams |
| Apelidos | Variável | Até 36 letras e/ou símbolos | Alexander George Thomas |
| Data de nascimento (yyyymmdd) | Fixo | 8 números | 19700301 |
| Data de emissão (yyyymmdd) | Fixo | 8 números | 20020915 |
| Data de expiração (yyyymmdd) | Fixo | 8 números | 20070930 |
| País emissor | Fixo | 3 letras | JPN |
| Autoridade emissora | Variável | Até 65 letras, números e/ou símbolos | HOKKAIDO PREFECTURAL POLICE ASAHIKAWA AREA PUBLIC SAFETY COMMISSION |
| Número de licença | Variável | Até 65 letras e/ou números | A290654395164273X |

Por sua vez, cada uma das qualificações de veículo é codificada com a *tag* `'7F63'`, seguida pelo comprimento total da lista e pelo número de entradas da mesma, codificado com a *tag* `'02'`. Cada elemento da lista é codificado com a *tag* `'87'` seguida pelo comprimento do elemento.

#### 4.1.5 Data group 6

Este grupo é responsável pelo armazenamento de uma variedade de identificadores biométricos do titular. No entanto, por motivos de simplicidade, assume-se que apenas a foto estará presente. Este ficheiro é identificado pela *tag* `'75'` e o identificador `'06'`, seguido pelo comprimento total do grupo. A seguir segue a seguinte formatação:

Tabela 4: Dados armazenados no DG6.

| Tag | Comprimento | Valor |
|---|---|---|
| ’7461’ | Variável | Informação dos subgrupos biométricos |
| ’02’ | Variável | Número de subgrupos biométricos do grupo<br><br>(no contexto do Projeto, 1) |
| ’7F60’ | Variável | Subgrupo biométrico |
| ’A1’ | Variável | Header do subgrupo biométrico (BHT) |
| ’80’ | 02 | Versão da header |
| ’86’ | 02 | BDB product owner, product type<br>(2 números positivos concatenados) |
| ’87’ | 02 | BDB format owner<br>(1 número positivo) |
| ’88’ | 02 | BDB format type<br>(1 número positivo) |
| ’5F2E’ | Variável | Bloco de dados biométrico<br>(formato definido pelos dois anteriores) |

#### 4.1.6 Data group 10

Este grupo é responsável por guardar os dados obrigatórios mDL, sendo identificado pela *tag* `’62’` e o identificador `’0A’`, seguido pelo comprimento total do grupo. Seguem-se então os seguintes parâmetros:

Tabela 5: Dados armazenados no DG10.

| Tag | Comprimento | Valor |
|---|---|---|
| 02 | 01 | Versão do mDL |
| ’5F28’ | 07 | Data do último update. Timestamp em UTC,<br>no formato YYYYMMDDhhmmss |
| ’5F2B’ | 07 | Data de expiração do mDL. Timestamp em UTC,<br>no formato YYYYMMDDhhmmss |
| ’5F38’ | 07 | Data do próximo update. Timestamp em UTC,<br>no formato YYYYMMDDhhmmss |
| ’5F39’ | Variável | Informação de gestão mDL para autoridades,<br>string de octetos |

#### 4.1.7 EF.COM

Este grupo de dados é responsável pelo armazenamento da versão da estrutura de dados lógica e a lista de *tags* correspondentes aos grupos de dados presentes na implementação do mDL. O EF.COM é identificado pela *tag* `’60’` e o identificador `’1E’`, seguido pelo comprimento total do grupo. Seguem-se então os seguintes parâmetros:

Tabela 6: Dados armazenados no EF.com

| Tag | Comprimento | Valor |
|---|---|---|
| '5F01' | Variável | Número da versão em formato 'aabb', no qual 'aa' define o maior nível de revisão e 'bb' define o nível de lançamento. Ambos são numéricos e codificados como 2 bytes em formato BCD. |
| '5C' | Variável | A lista de tags de grupos de dados. |

#### 4.1.8 EF.SOD

O elemento EF.SOD é um documento de segurança identificado pela *tag* `'77'` e o identificador `'1D'`. Trata-se de uma assinatura digital gerada através da concatenação das assinaturas de todos os restantes elementos presentes, em formato DER.

#### 4.1.9 EF.GroupAccess

O grupo EF.GroupAccess contem a informação de quais dados estão disponíveis a um leitor mDL, sendo identificado pelo identificador `'18'`. Este contém apenas a *tag* `'04'`, o comprimento dos dados e uma *string* de octetos correspondentes aos grupos de dados a que o leitor tem acesso, bem como as respetivas *tags*. Como este elemento é dinâmico, a sua *hash* não está presente no ficheiro EF.SOD.

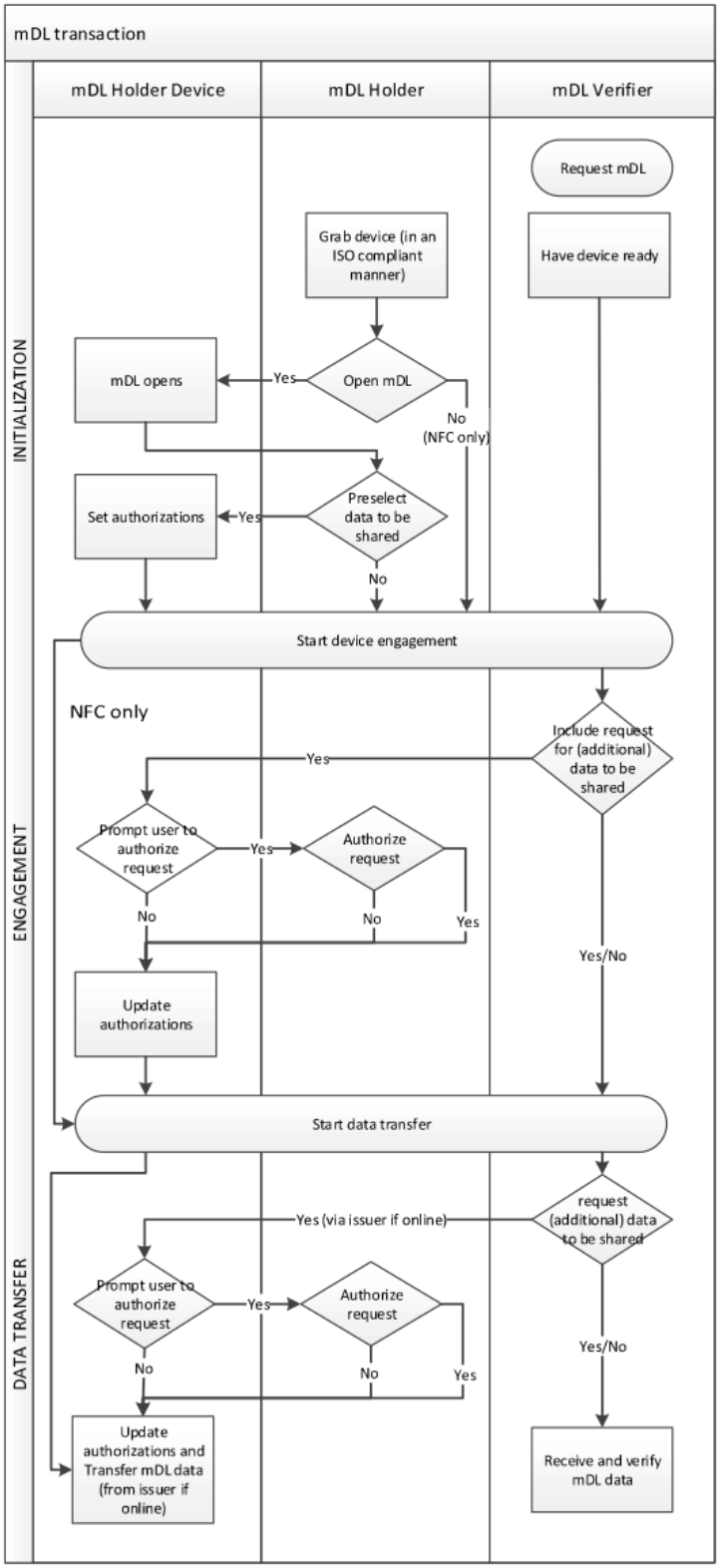
### 4.2 *Compact encoding*

O *Compact encoding* é o esquema de dados utilizado na transferência de informação por meio de uma interface ótica, tais como códigos de barras ou fitas magnéticas. Estes requerem um espaço de armazenamento entre 300 B e 5 kB. Devido a esta limitação, o número de grupos de dados é restrito, bem como o espaço permitido para cada um.

O esquema do *compact encoding* providencia espaço para os grupos de dados 1, 6 e 10 obrigatoriamente, bem como a possibilidade da utilização dos grupos 2, 3, 4, 7 e 11, caso necessário.

## 5 Transferência de Dados da mDL

Uma transação mDL consiste em três fases, sendo estas a inicialização, a conexão entre dispositivos e a transferência de dados. Estas fases são bem expressas no diagrama que se segue:




**Inicialização**

Durante a inicialização, uma mDL é aberta pelo utilizador ou, potencialmente, ativada pelo NFC. O utilizador pode, opcionalmente, pré-autorizar a partilha de certos elementos de dados.

**Conexão entre Dispositivos**

Durante o estabelecimento da conexão entre dispositivos, é utilizado NFC ou um código QR (*Quick Response*) para transferir a estrutura de conexão de dispositivos, de forma a configurar o passo seguinte de transferência de dados. Os leitores mDL devem suportar tanto a interface ótica (código QR) como as tecnologias NFC. Quando é utilizado NFC, é possível a comunicação nos dois sentidos, possibilitando que o leitor mDL solicite acesso a elementos adicionais para partilha.

**Transferência de Dados**

A transferência de dados pode utilizar um método *offline* ou *online*. Em qualquer caso, a conexão pode ser utilizada para solicitar acesso a elementos de dados (adicionais) ao leitor mDL. Uma mDL deverá suportar qualquer um dos seguintes métodos de transferência de dados *offline*: NFC, *Bluetooth Low Energy* (BLE) ou código de barras 2D. Os leitores mDL deverão suportar, obrigatoriamente, essas três tecnologias, bem como, opcionalmente, a *Wi-Fi Aware*.

Figura 2: Processo de transferência de dados de uma mDL

# 6 Mecanismos de Proteção de Dados da mDL

## 6.1 Controlos de segurança

### 6.1.1 Autenticação Passiva (ISO/IEC 18013-3)

A autenticação passiva tem como objetivo confirmar que a *machine-readable data* não foi alterada desde que a IDL (ISO-compliant driving license) foi emitida.

Na prática, este mecanismo é implementado usando criptografia de chave pública (assimétrica) para produzir assinaturas digitais sobre *machine-readable data*.

De facto, se for usado *standard encoding*, a *machine-readable data* deve incluir:

- uma *message digest* de cada grupo de dados presente na IDL;
- uma assinatura digital relativa à coleção de todas as *message digest*. Esta assinatura deve usar a chave privada da autoridade emissora (IA).

Posteriormente, quando uma autoridade de leitura (RA) tentar aceder ao conteúdo da IDL, esta deve realizar os seguintes passos:

- verificar a assinatura digital utilizando a chave pública da IA;
- computar o *message digest* para cada um dos grupos de dados que sejam de interesse e comparar o seu resultado com os respetivos *message digest* guardados no *machine-readable data* do IDL.

Desta forma, após os passos anteriores estarem concluídos, a RA pode considerar que os grupos de dados que quer ler são autenticos se:

- a assinatura digital verifica;
- os *message digest* calculados correspondem aos armazenados na *machine-readable data*;
- a RA está confiante que a chave pública, usada para verificar a assinatura, pertence efetivamente à IA que emitiu a IDL.

No entanto, caso algum dos passos anteriores não seja válido, então significa que pelo menos um dos seguintes aspetos aconteceu:

- a assinatura digital não foi verificada com sucesso;
- a chave pública usada não era a correta;
- os dados na IDL foram alterados.

**Funções de Hash - Codificação *standard***

Para *standard encoding* a IA deve utilizar funções de hash presentes na seguinte lista:

- SHA-1 (apenas por compatibilidade)
- SHA-224
- SHA-256 (recomendada)
- SHA-384
- SHA-512

Um *message digest* é calculado para cada grupo de dados presente na IDL e armazenado na *machine-readable data*.

A mesma função de hash deve ser usada para todos os grupos de dados.

O cálculo do *message digest* deve ser aplicado à concatenação de todos os elementos de dados, presentes num dado grupo de dados, pela ordem especificada no ISO/IEC 18013-2.

**Funções de Hash - Codificação compacta**

Neste tipo de codificação não é realizado o cálculo do valor de hash para cada grupo de dados em particular. Desta forma, não é necessário o uso de funções de hash.

**Método de assinatura - Codificação *standard***

A assinatura digital do IDL deve ser gerada sobre a concatenação dos *message digest* dos grupos de dados presentes. Para tal, a IA pode utilizar dois métodos de assinatura diferentes:

- ECDSA
- RSA

Por um lado, caso a IA opte pelo uso do ECDSA, então deve:

- usar ANSI X9.62
- incluir, de forma explícita, na chave pública os parâmetros de domínio da curva elíptica usados para gerar o par de chaves ECDSA. Assim, esta informação deve ser do tipo `ECParameters` (sem nomes de curvas e sem parâmetros implícitos) e deve incluir o *cofactor* opcional.
- garantir que os `ECPoints` estão no formato descompactado.
- garantir que o tamanho mínimo da ordem do ponto base seja 160 bits.

Por outro lado, caso a IA opte pelo uso do RSA, então deve:

- seguir o RFC 4055, ou seja, escolher o mecanismo RSASSA-PSS (recomendado) ou o RSASSA-PKCS1-v1_5.
- garantir que o tamanho mínimo do *modulus* (n) é 1024 bits.

Assim sendo, a IA, além do `EF.COM` e dos grupos de dados mencionados no ISO/IEC 18013-2 (DG1 até ao DG11), deve adicionar um SOD para incluir as hashes individuais de cada grupo de dados e a assinatura digital no IDL. O SOD deve ser do tipo `SignedData` (RFC 3369) e deve ser produzido no formato DER.

**Método de assinatura - Codificação compacta**

Neste tipo de codificação, a assinatura digital é gerada sobre a totalidade dos dados, ou seja, desde o DG1 até ao DG12 sem contar com o DG.SOD e o cabeçalho. O valor é depois armazenado no DG.SOD.

O método usado para assinar deve ser o ECDSA com uma ordem de tamanho mínimo de 224 bits. Além disso, de forma a reduzir o armazenamento necessário para os parâmetros de domínio deve ser usada uma das seguintes curvas:

- P-224
- P-256
- P-384
- P-521
- brainpoolP224r1
- brainpoolP224t1
- brainpoolP256r1
- brainpoolP256t1
- brainpoolP320r1
- brainpoolP320t1
- brainpoolP384r1
- brainpoolP384t1
- brainpoolP512r1
- brainpoolP512t1

Na prática, o elemento DG.SOD deve ser composto pelos seguintes campos:

- DG.SOD.1: assinatura digital (codificada em DER)

```
SEQUENCE ::= { r INTEGER, s INTEGER }
```

- DG.SOD.2: chave pública
- DG.SOD.3: identificador da curva

Posteriormente, este campo deve ser adicionado aos restantes na seguinte ordem:

```
[header] x [Data Group 1] x [Data Group 2] x [Data Group 3] x
[Data Group4] x [Data Group 7] x [Data Group 11] x
[Data Group 12] x [DG.SOD.1 length] [digital signature] x
[DG.SOD.2 length] [public key] x [DG.SOD.3: named curve]
```

### 6.1.2 Outros mecanismos

Além dos mecanismos de segurança anteriormente mencionados, existem ainda outros, como:

- **Autenticação ativa**. Tem como objetivo confirmar que o *secure integrated chip* (SIC) foi emitido juntamente com a *machine-readable data*.
- **Proteção de acesso básico (BAP)**. Tem como objetivo verificar que o IS tem acesso ao *proximity integrated circuit card* (PICC) antes de aceder aos dados guardados no seu interior. Adicionalmente, o BAP assegura que a comunicação entre o IS e o PICC é protegida.
- **Proteção de acesso extendido (EAP)**. Tem como objetivo permitir o acesso condicional autenticado a certos grupos de dados.

Apesar destes mecanismos se encontrarem no documento ISO, utilizam dispositivos físicos em algum momento do seu processo. Dado que o objetivo é focar num sistema completamente digital optou-se por não aborda-los.

## 6.2 Controlos de privacidade

### 6.2.1 Consentimento do utilizador

O acesso, por parte de um dispositivo de leitura, aos dados guardados numa mDL, apenas deve ser permitido depois do consentimento (implícito ou explícito) do titular do mDL.

Os métodos para o consentimento do utilizador não fazem parte dos temas abrangidos neste documento.

Com o objetivo de proteger a privacidade do titular do mDL, aplicações do governo não devem rastrear os movimentos de indivíduos através de *geo tagging*.

# 7 Implementação da mDL (ISO *compliant*)

## 7.1 Parser ASN.1

Para implementar o armazenamento dos dados de uma mDL foi necessário desenvolver-se um *parser* capaz de guardar os mesmos como objetos de dados BER-TLV (*Basic Encoding Rules - Tag, Length, Value*), em hexadecimal, bem como recuperar esses dados para uma estrutura adequada.

O principal objetivo deste *parser* é automatizar o processo de conversão dos dados entre uma estrutura de dados e texto hexadecimal, e vice-versa. Para isso, basta indicar uma configuração básica ao *parser*, escrita em `JSON`, sendo esta a estrutura seguida aquando da conversão.

A título ilustrativo, apresenta-se de seguida um exemplo básico deste tipo de configuração.

```
{
    "tag": "00",
    "length": "var",
    "content": {
        "var_name": {
            "tag": "01",
            "length": 4,
            "constraints": "[$N]{8}",
            "encode": "BCD"
        },

        "list_name": {
            "tag": "02",
            "length": "var",
            "content": {
                "number_of_entries": {
                    "tag": "03",
                    "length": "var",
                    "constraints": "(\\d{2}|1\\d{2})",
                    "decode": "int"
                },
```

```
                "list_var_name": {
                    "size_list": "number_of_entries",
                    "tag": "04",
                    "length": "var",
                    "constraints_func": "myFunction($DATA)"
                }
            }
        }
    }
}
```

Salienta-se a utilização das chaves `tag` e `length` que indicam, respetivamente, a *tag* e o comprimento dos dados incluídos nessa configuração. O comprimento dos dados pode ser fixo (caso em que é indicado um inteiro correspondente à quantidade de *bytes*) ou variável (`var`). No primeiro caso o comprimento é omitido da representação em hexadecimal, enquanto que no segundo caso esse valor tem de ser indicado. A `tag` é sempre apresentada na representação em hexadecimal e serve para identificar o elemento de dados que se está a ler quando se pretende converter essa representação para uma estrutura de dados.

Para além disso, a chave `content` indica a presença de variáveis das estruturas de dados (elementos de dados básicos) no respetivo valor. A variável é encontrada quando se alcança uma chave (diferente de `content`) cujo conteúdo não inclua nenhum `content`. É nesse ponto que os dados são, de facto, convertidos entre formatos. Relativamente ao exemplo de configuração ilustrativo apresentado, a respetiva estrutura de dados seria a apresentada na figura 3, como *input*.

Existem ainda outros elementos de validação que podem ser indicados relativamente aos elementos de dados básicos:

- `constraints` – Formato que o elemento de dados, como *string*, deverá ter, recorrendo a expressões regulares. Este formato é validado, sendo apresentado um erro caso as restrições não se verifiquem. Salienta-se a possibilidade de utilização de alguns símbolos, que facilitam a representação de determinados tipos de caracteres:
  - `[$A]` : Caracter alfabético.
  - `[$N]` : Caracter numérico.
  - `[$S]` : Símbolo.
  - `[$D]` : Delimitador.

  Estes caracteres podem ser combinados de várias formas. Por exemplo, pode-se representar um caracter alfanumérico ou espaço simples da seguinte forma: `[$A$N ]`.

- `constraints_func` – Invocação da função a utilizar para validar as restrições do elemento de dados associado. Note-se que esta função tem de ser implementada para cada caso específico. Possibilita-se a utilização da etiqueta `$DATA` para aceder aos dados que se pretende validar.

- `encode` – Formato utilizado para a codificação em hexadecimal, pelo que este componente poderá ser mais específico que os restantes. Pode tomar um dos seguintes valores (caso não se apresente nenhum destes, utiliza-se a codificação por defeito):

  - `BCD`: *Binary Coded Decimal* (utilizado sobre inteiros).
  - `NO_ENCODE`: Sem qualquer codificação (elemento de dados já se encontra no formato hexadecimal).
  - `CATEGORIES_ENCODE`: Codificação utilizada para as categorias de veículos.
  - `DG_TAGS_ENCODE`: Codificação utilizada para uma lista de identificadores e *tags* dos grupos de dados.
  - `DG_TAGS_LIST_ENCODE`: Codificação utilizada para uma lista de *tags* dos grupos de dados.
  - `BDB_ENCODE`: Codificação de um *Biometric Data Blocks*.
  - `VERSION_ENCODE`: Codificação da versão de um *Biometric Data Block*.
  - `DEFAULT`: Codificação por defeito (inteiro ou *string* é convertido diretamente em *string* hexadecimal).

- `decode` – Formato utilizado para a descodificar de hexadecimal para *string* ou *inteiro*. Note-se que a descodificação é efetuada de acordo com a codificação, mas os elementos básicos podem precisar de ser distinguidos entre os dois tipos referidos. Assim, pode tomar um dos seguintes valores (caso não se apresente nenhum destes, utiliza-se a descodificação para *string*): `INT` e `STR`.

- `size_list` – Nome da chave (e da variável) que indica o número de elementos da respetiva lista. Note-se que a presença desta chave indica que o dado associado é uma lista, em que cada elemento tem o formato especificado nessa configuração.

Após esta análise do que é possível representar através da configuração `JSON`, é necessário perceber como esta é utilizada para converter estruturas de dados numa *string* hexadecimal e vice-versa.

Relativamente à fase de codificação, salienta-se a função principal `encode`, que recebe como argumento a estrutura de dados a codificar e o nome do ficheiro `JSON` com a configuração. Esta função simplesmente lê o conteúdo da configuração e invoca a função recursiva `asn1_encode`, passando como argumento a estrutura de dados e a respetiva configuração.

A função `asn1_encode` é recursiva devido ao facto da configuração poder ter bastantes níveis de conteúdo, sendo necessário aplicar a mesma função para efetuar a codificação associada. Em termos gerais, esta função funciona conforme apresentado na figura 3.

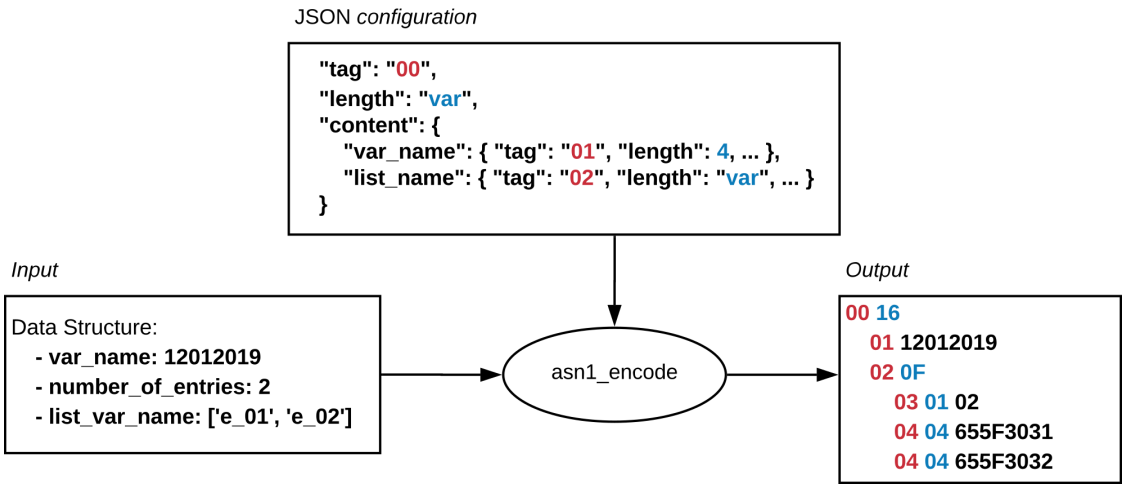



Figura 3: Representação do funcionamento geral da função `asn1_encode` (invocação principal).

Neste diagrama, bem como nos que se apresentarão de seguida, encontram-se realçadas a negrito as porções da configuração que são passadas como argumento à função indicada e as partes da estrutura de dados que são utilizadas. Para além disso, realçou-se as *tags* a vermelho e os comprimentos a azul, de forma a permitir a sua fácil localização no *output*. A formatação deste *output* (espaçamento e cor) serve apenas para facilitar a perceção do mesmo, isto é, o *output* real consiste apenas na *string* simples em hexadecimal.

Como é necessário conhecer o comprimento dos dados associados ao `content` para indicar o mesmo na representação em hexadecimal, primeiro tem de se efetuar o *parse* desse conteúdo. Assim, para cada chave dentro do `content`, a função `asn1_encode` volta a invocar-se, com essa configuração, para obter os respetivos elementos de dados convertidos em hexadecimal. Assim, a função `asn1_encode`, com os argumentos apresentados na figura 3, efetua uma invocação recursiva com cada um dos seus `content`, conforme apresentado nas figuras 4 e 5.

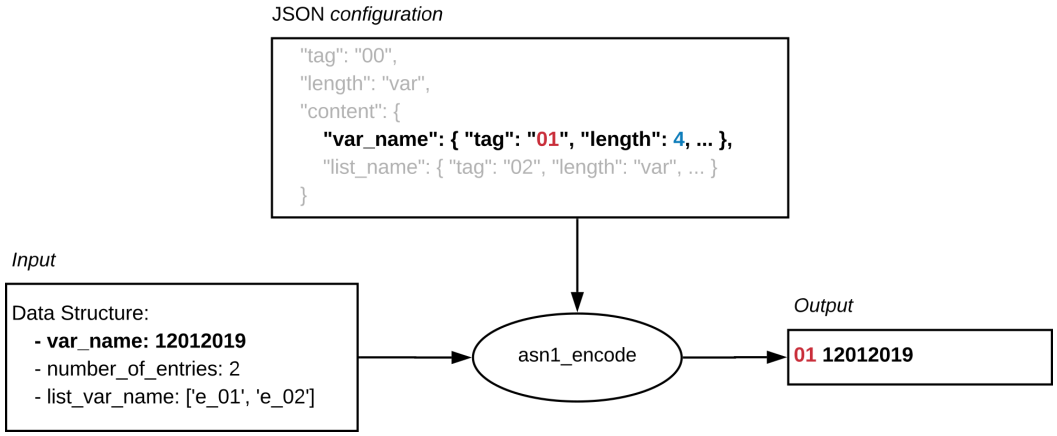



Figura 4: Representação do funcionamento geral da função `asn1_encode` (invocação para o conteúdo "var_name").

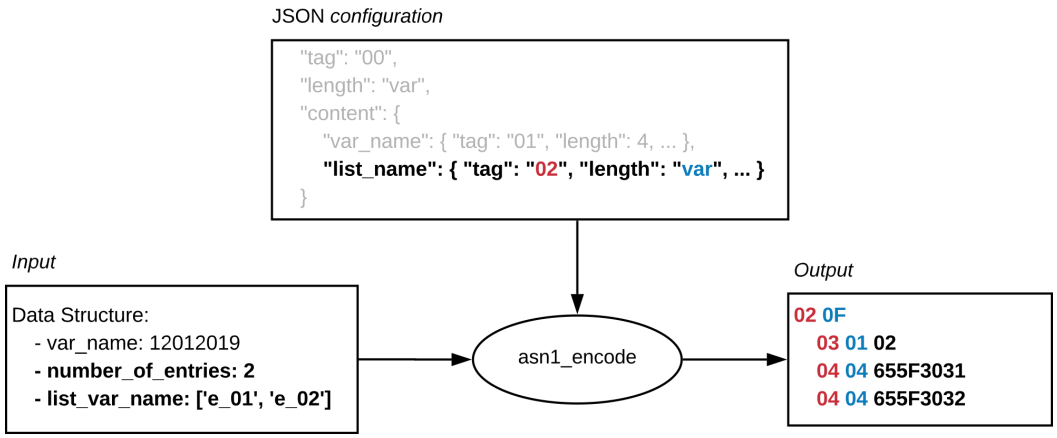



Figura 5: Representação do funcionamento geral da função `asn1_encode` ("list_name").

Relativamente à lista, processada na figura 5, é notória a necessidade de novas invocações recursivas da função `asn1_encode`. Para o conteúdo `number_of_entries`, o processamento é semelhante ao apresentado para a variável `var_name`. Relativamente ao conteúdo `list_var_name`, a configuração realçada na figura 6 tem de ser utilizada para converter cada elemento da lista `list_var_name` do *input*. Nessa figura apenas se representa a conversão para o primeiro elemento.

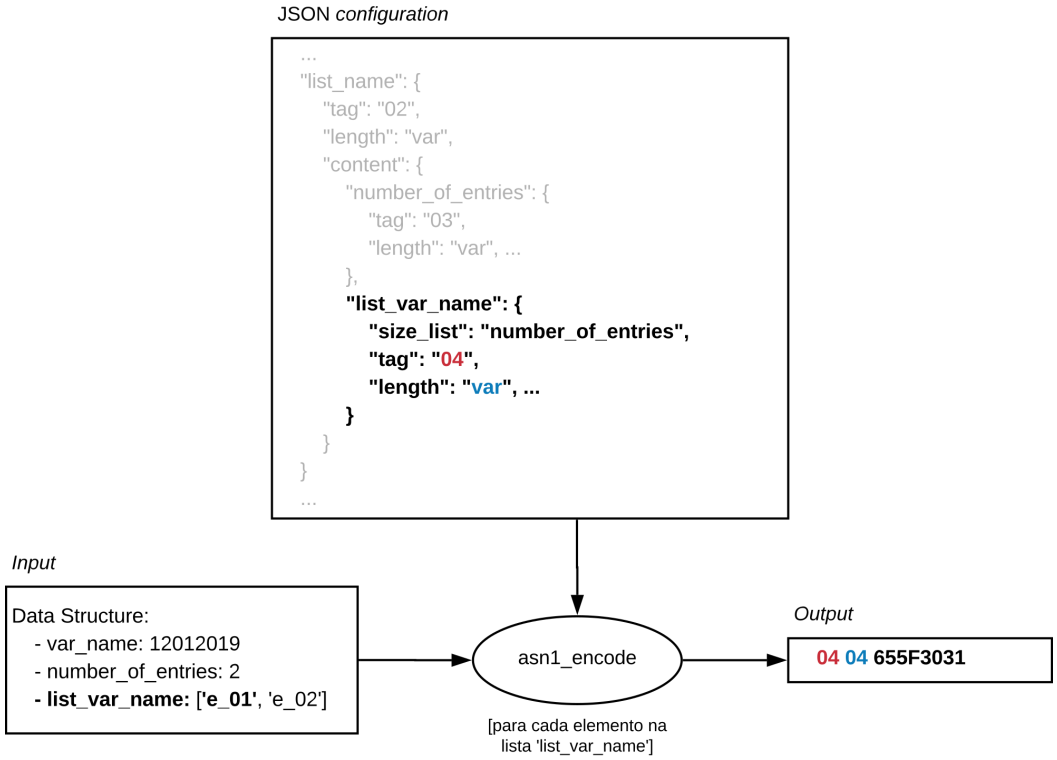



Figura 6: Representação do funcionamento geral da função `asn1_encode` (invocação para o conteúdo "list_var_name", dentro de "list_name").

Para além destas listas simples, também é necessário, por vezes, armazenar listas compostas, em que cada elemento contém um determinado conjunto de dados. Para isso, pode-se utilizar uma configuração semelhante à que se apresenta de seguida.

```
{
    ...
    "compost_list_name": {
        "tag": "05",
        "length": "var",
        "content": {
            "number_of_entries": {
                "tag": "03",
                "length": "var",
                "decode": "int"
            },
            "list_var_name": {
                "size_list": "number_of_entries",
                "tag": "06",
                "length": "var",
                "compost_content": {
                    "unused_name": {
                        "tag": "07",
                        "length": "var",
                        "content": {
                            "var_name_1": {
                                "tag": "08",
                                "length": 2,
                                "decode": "int"
                            }
                        }
                    },
                    "var_name_2": {
                        "tag": "09",
                        "length": "var"
                    }
                }
            }
        }
    }
    ...
}
```

Neste caso, utiliza-se a chave `compost_content`, em vez de `content`, na configuração relativa ao elemento da lista. Desta forma, o *parser* consegue distinguir se os dados associados correspondem a uma lista de elementos básicos (*strings* ou inteiros) ou a uma lista de estruturas compostas.

Relativamente à descodificação da *string* hexadecimal para uma estrutura de dados, esta é efetuada pela função principal `decode`, que recebe como argumentos a *string* hexadecimal e o nome do ficheiro `JSON` com a configuração. Esta função invoca a `asn1_decode` com a *string* hexadecimal e a respetiva configuração, retornando o seu resultado (dicionário em que as chaves correspondem aos nomes das variáveis da estrutura original e os valores aos respetivos dados).

A função `asn1_decode` também é recursiva, pelo motivo previamente apresentado em relação à função `asn1_encode`. Neste caso, começa-se por extrair do início da *string* hexadecimal a *tag* e o respetivo comprimento (se este não for fixo, de acordo com a configuração). De seguida, extrai-se esse comprimento de *bytes* do início da *string* resultante, à qual se aplicará a função *asn1_decode*, se tiver conteúdo, ou se efetuará diretamente a conversão dos dados, caso contrário. Caso o comprimento da *string* hexadecimal ultrapasse o comprimento esperado, é lançada uma exceção.

Assim, este é precisamente o processo inverso ao apresentado anteriormente, não se entrando, portanto, em tanto detalhe. Apenas se apresenta um diagrama representativo do funcionamento geral da função *asn1_decode*, na figura 7.

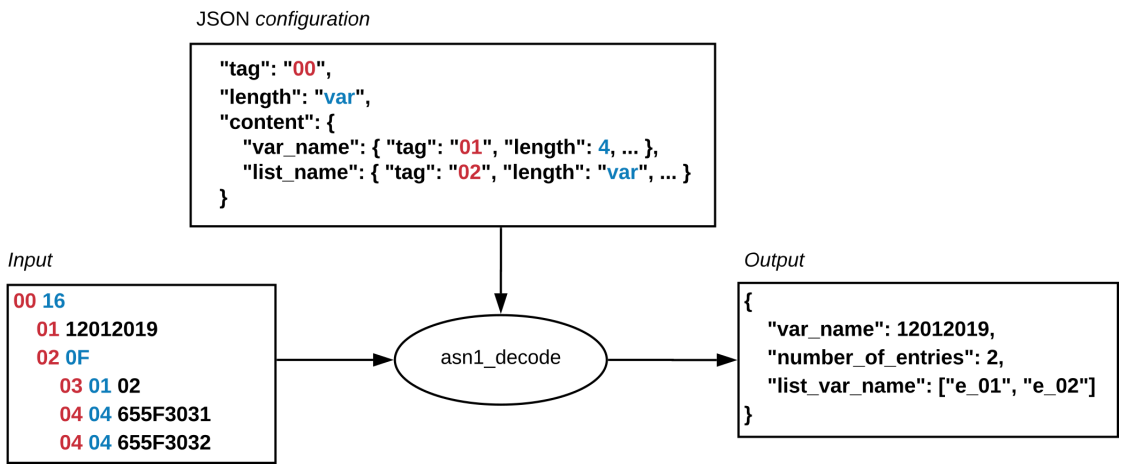



Figura 7: Representação do funcionamento geral da função `asn1_decode`.

Neste contexto, salienta-se apenas uma restrição relativamente ao comprimento dos dados armazenados para cada estrutura de dados, que não pode ultrapassar os 65 535 *bytes*. A implementação foi efetuada desta forma, tendo em conta as regras de codificação do ASN.1 *length*, especificadas no ISO/IEC 18013-2.

Por fim, tendo em conta toda a funcionalidade providenciada pelo *parser*, este permitirá converter, de acordo com a especificação do ISO/IEC 18013, os dados dos *data groups* em hexadecimal e vice-versa. Para além disso, também permite efetuar uma validação desses dados antes de os armazenar e após estes serem lidos.

## 7.2 Grupos de dados (DG's)

Para cada um dos ficheiros que compõe a estrutura de dados, foi implementada uma *script* com um conjunto de funcionalidades que as permitem criar e gerir os respetivos ficheiros. Estes são depois acedidos pelo *mDL* através da sua API, da qual as *scripts* partilham a capacidade

de carregar e escrever os dados do respetivo grupo de dados em ficheiro, bem como codificar o seu conteúdo para o formato ASN1:

```
def save(self, filename):
    with open(filename, 'w+') as fp:
        hex_data = asn1.encode(self, './data_groups/configs/dg1.json')
        fp.write(hex_data)

def load(self, filename):
    with open(filename, 'r') as fp:
        data = asn1.decode(fp.read(), './data_groups/configs/dg1.json')
    return data

def encode(self):
    return asn1.encode(self, './data_groups/configs/dg1.json')
```

Tanto o carregamento como a escrita destes é executada com a ajuda de um ficheiro JSON que permite alterações à estrutura da implementação sem que haja alterações no código em si. Cada um dos DG's tem o seu próprio JSON, no qual são guardados os seus campos, bem como os respetivos comprimentos em *bytes* e as suas *tags*. Para além disso, cada DG é convertido de e para o formato ASN1, consoante a leitura e escrita, respetivamente.

## 7.3 Data Groups 1, 6 e 10

Estes DG's são responsáveis por armazenar os dados do titular mDL, bem como os do próprio título mDL. Cada um destes contem um conjunto de funções que as permitem demonstrar o seu conteúdo conforme necessário. Cada um destes tem a sua implementação específica, ainda assim, seguem uma estrutura idêntica. Em particular, a função `get_data` preenche e devolve um dicionário com todos os valores do respetivo grupo de dados e a função `__str__` devolve o conteúdo do mesmo sob a forma de uma *string*. Por fim, é implementada uma função de *hash* que suporta um conjunto de diferentes algoritmos de acordo com o *standard* do mDL, entre os quais, o SHA-1, SHA-224, SHA-256 e SHA-512.

### 7.3.1 Data Group 1

Este DG é responsável por armazenar os dados do titular mDL, bem como alguma informação básica do próprio título mDL. Cada um destes contém um conjunto de variáveis de instância correspondentes aos respetivos dados explicitados na estrutura de dados:

```
def __init__(self, data):
    if isinstance(data, str):
        data = self.load(data)
    self.family_name = data['family_name']
    self.name = data['name']
    self.date_of_birth = data['date_of_birth']
    self.date_of_issue = data['date_of_issue']
    self.date_of_expiry = data['date_of_expiry']
```

```
    self.issuing_country = data['issuing_country']
    self.issuing_authority = data['issuing_authority']
    self.license_number = data['license_number']
    self.number_of_entries = data['number_of_entries']
    self.categories_of_vehicles = data['categories_of_vehicles']
```

### 7.3.2 Data Group 6

Este DG contém um conjunto de dados biométricos relativos ao titular, em particular, a sua foto. Para permitir uma melhor gestão deste ficheiro, foi criada uma classe auxiliar `BiometricTemplate`:

```
class BiometricTemplate:
    def __init__(self, version, bdb_owner, bdb_type, bdb):
        self.version = version if version is not None else '0101'
        self.bdb_owner = bdb_owner
        self.bdb_type = bdb_type
        self.bdb = bdb

    def __str__(self):
        return '(' + ', '.join([
            str(self.version),
            str(self.bdb_owner),
            str(self.bdb_type),
            str(self.bdb)[:20] + '...'
        ]) + ')'


class DG6:
    def __init__(self, data):
        if isinstance(data, str):
            data = self.load(data)
        self.biometric_templates = []

        for template in data['biometric_templates']:
            self.biometric_templates.append(
                BiometricTemplate(
                    template['version'],
                    template['bdb_owner'],
                    template['bdb_type'],
                    template['bdb']
                )
            )
        self.number_of_entries = data['number_of_entries']
```

Nesta classe auxiliar, é guardado um único tipo de identificador biométrico, quer seja uma foto, impressão digital, entre outros. Como tal, o DG6 trata-se de um conjunto destes.

### 7.3.3 Data Group 10

Este grupo de dados é responsável por armazenar os dados relativos à gestão do mDL, incluindo a sua data de validade e as datas do último e próximo *updates*.

```
    def __init__(self, data):
    if isinstance(data, str):
        data = self.load(data)
    self.version = data['version']
    self.last_update = data['last_update']
    self.expiration_date = data['expiration_date']
    self.next_update = data['next_update']
    self.management_info = data['management_info']
```

## 7.4 EF.COM

O ficheiro elementar COM é responsável por armazenar as *tags* relativas aos grupos de dados presentes numa dada implementação do mDL.

```
def __init__(self, data):
    if isinstance(data, str):
        data = self.load(data)
    self.version = data['version']
    self.tag_list = data['tag_list']
```

Como esta funciona puramente como um meio rápido de averiguar os dados disponíveis ao mDL, este contém apenas as funções de API básica, bem como uma função de exibição de dados semelhante aos grupos de dados anteriormente expostos.

## 7.5 EF.GroupAccess

Este ficheiro é responsável por guardar a lista de grupos aos quais um titular mDL dá permissões de leitura a um dado requerente mDL.

```
def __init__(self, data):
    if isinstance(data, str):
        self.allowed_data_groups = self.load(data)['allowed_data_groups']
    else:
        self.allowed_data_groups = {}
        for dg, tag in data.items():
            self.allowed_data_groups[int(dg)] = tag
```

Como este é utilizada sempre que é efetuada uma transação de dados, requer um conjunto de funções que permitam a leitura e gestão do seu conteúdo.

```
def set_permissions(self, allowed):
    self.allowed_data_groups = {}
    for dg, tag in allowed.items():
        self.allowed_data_groups[dg] = tag

def add_permissions(self, allowed):
    for dg, tag in allowed.items():
        self.allowed_data_groups[dg] = tag

def is_allowed(self, dg):
    return dg in self.allowed_data_groups
```

### 7.6 EF.SOD

Este ficheiro é responsável por gerir os valores de *hash* dos dados mDL e gerar assinaturas digitais. Este contém uma estrutura que varia significativamente dos restantes ficheiros e, como tal, contém o seu próprio *parser* denominado de `ef_sod_parser`. Este difere do *parser* utilizado pelos restantes no sentido em que recorre a uma biblioteca `pyasn1`, que se melhor ajusta à estrutura requerida. Como tal, o *encoding* para o formato ASN1 nas funções de leitura e escrita de ficheiros é executada por estes:

```
def sod_encode(signed_data):
    global TAG
    return TAG + encode(signed_data).hex()

def sod_decode(hex_signed_data):
    global TAG
    assert hex_signed_data.startswith(TAG), 'ERROR: Unknown or invalid tag.'
    signed_data, rest = decode(bytes.fromhex(hex_signed_data[2:]),
                               asn1Spec=SignedData())
    assert rest == b'', 'ERROR: "rest" not null.'

    return signed_data
```

Para que o mDL possa gerir os as suas assinaturas, este recebe uma API appropriada que permite a geração de assinaturas digitais baseadas nos conteúdos de dados que se pretende enviar a um requerente. Em particular deestacam-se as funções de *get* e *set* de *digests* e de assinaturas digitais, respetivamente:

- A função `get_digests` devolve um dicionário de dados composto pelas *hashes* guardadas na classe.
- A função `set_digests` permite o carregamento do conjunto de *hashes* dos grupos de dados a enviar numa instância da classe para posterior escrita e/ou envio.
- A função `get_signature` devolve a assinatura digital do respetivo EF.SOD.

- A função `set_signature` é responsável por assinar os dados do EF.SOD recorrendo a um algoritmo de assinatura e a uma chave privada correspondente ao certificado, ambos armazenados no mesmo.

## 7.7 Aplicação mDL

A aplicação mDL tem como objetivo incorporar todos os componentes anteriormente descritos, de forma a fornecer as funcionalidades que permitem animar a interação com o mDL.

### 7.7.1 Funcionalidades (API)

Primeiramente, é necessário inicializar o mDL (`__init__`). Para tal, à semelhança de outras classes da ferramenta, é possível passar um JSON com os dados, ou simplesmente deixar que o mDL carregue os dados que guardou na última utilização (`load`). Esta última opção é apenas possível se já tiver sido usada a função `save`, que persiste os dados (codificados em ASN1) do mDL em ficheiros. Na prática, tanto o `load` como o `save` realizam as suas operações delegando a tarefa a cada um dos grupos de dados.

```
def load(self):
    self.dg1 = dg1.DG1('./data_groups/asn1_hex_data/dg1.txt')
    self.dg6 = dg6.DG6('./data_groups/asn1_hex_data/dg6.txt')
    (...)
```

```
def save(self):
    self.dg1.save('./data_groups/asn1_hex_data/dg1.txt')
    self.dg6.save('./data_groups/asn1_hex_data/dg6.txt')
    (...)
```

Mais ainda, uma parte fundamental do mDL é a possibilidade de definir quais os grupos de dados aos quais se tem acesso. Para tal, são utilizadas as funcionalidades de `set_permissions` e `add_permissions`, oferecidas pela classe `ef_groupAccess`, para definir permissões ou permitir novos grupos de dados, respetivamente. Importa salientar que o documento ISO que serviu de referência para a realização deste projeto, não afirma de forma clara, a que é que se deve aplicar as permissões mencionadas anteriormente. No entanto, é referido no ISO que os valores de hash, utilizados na autenticação passiva, são sobre o DG completo (Pág. 20 da parte 3). Desta forma, a validação efetuada pelo verificador implica que este tenha acesso a todos os elementos do DG, de forma a reconstruir o *digest*, mesmo que só tenha permissão para alguns. Tendo em conta este caso, considerou-se que as permissões são estabelecidas por DG e não para os campos de cada DG individualmente.

```
def set_permissions(self, allow):
        self.ef_groupAccess.set_permissions(allow)
```

```
def add_permissions(self, allow):
        self.ef_groupAccess.add_permissions(allow)
```

Outra funcionalidade disponibilizada está relacionada com a necessidade de autenticar a origem da informação da mDL. Para tal, o processo de autenticação passiva incorpora um mecanismo de validação da assinatura dos dados do mDL. De forma a disponibilizar esta funcionalidade tem-se a função `get_signature` que devolve as informações necessárias para que o leitor mDL possa realizar a validação. Adicionalmente, o processo de autenticação passiva envolve a validação dos valores de hash dos dados de cada DG. Esses mesmo valores também são incorporados na resposta devolvida pela função `get_sod_data_hex`.

```
def get_sod_data_hex(self):
    return self.ef_sod.encode()
```

A função `get_available_data_groups` surge como uma necessidade na implementação do processo de transação existente entre um dispositivo mDL e um leitor mDL. Efetivamente, o primeiro passo do mDL envolve a pré-seleção dos dados que podem ser partilhados. Desta forma, a função anterior é necessária para se averiguar quais os DG's que se encontram no mDL, para que, posteriormente, se possa definir as respetivas permissões.

```
def get_available_data_groups(self):
        return self.ef_com.get_data()['tag_list']
```

Por fim, tem-se a função `get_data_hex` que permite que os dados do mDL, codificados ASN1, sejam acedidos. Na prática a função percorre as tags dos grupos de dados recebidas como argumento e verifica, consultando o EF.GroupAccess se o DG pode ser acedido. Se tal for verdade, então o conteúdo do DG, codificado em ASN1, é concatenado com o restante *output* e é retornado.

```
def get_data_hex(self, data_group_tags):
        hex_data = ''
        for tag in data_group_tags:
                num_dg = self.TAGS[tag]
                if self.ef_groupAccess.is_allowed(num_dg):
                        hex_data += self.get_dg(num_dg).encode()
        return hex_data
```

Adicionalmente, também é possível aceder aos dados de forma estruturada (sem ser codificados em hexadecimal), através da função `get_data`.

### 7.7.2 Exemplo de utilização

Um possível exemplo de utilização envolve a execução de uma transação entre um dispositivo mDL e um leitor mDL. Para tal, foram criadas duas classes: `mDL_transaction` e `mDL_simulator`. Por um lado, a primeira tem como objetivo utilizar a classe `mDL` e implementar os passos que possam existir numa dada transação. Por outro lado, a segunda tem como objetivo animar a transação entre os dois intervenientes.

A classe `mDL_transaction` possui a seguinte API:

- `__init__`: inicializa o mDL.
- `open`: anima o processo de abertura do mDL
- `preselect`: permite que o utilizador defina as permissões para os DG's existentes.
- `request_additional_data`: permite que seja requerido acesso a determinados DG's.
- `transfer_data`: devolve os dados do mDL requeridos.

A título exemplificativo, o `mDL_simulator` assume a seguinte transação:

```
(...) # Carregamos do ficheiro JSON com os dados
MDLT = mDL_transaction(DATA)
MDLT.open()
MDLT.request_additional_data(['61', '62'])
transfer_data = MDLT.transfer_data(['61', '62'])
transfer_sod_data = MDLT.transfer_sod()

dg1 = dg1.DG1(transfer_data[1])
dg10 = dg10.DG10(transfer_data[10])
dgs = {1: dg1, 10: dg10}
transfer_digests = calculate_digests(dgs, transfer_sod_data['digestAlgorithm'])

(...) # Cálculo do valor dos digests dos DG's que se pretende aceder

digest1 = transfer_sod_data['dataGroupHash'][0]['dataGroupHashValue']
digest10 = transfer_sod_data['dataGroupHash'][2]['dataGroupHashValue']

(...) # Concatenação dos valores dos digests recebidos do mDL

print('DIGESTS VALIDATION:',
        transfer_digests[1] == digest1 and\
        transfer_digests[10] == digest10)
print('SIGNATURE VALIDATION:',
    verify_signature(
        sod_digests,
        bytes(transfer_sod_data['certificate']),
        bytes(transfer_sod_data['signature']),
        transfer_sod_data['signatureAlgorithm']
    ))
```

De forma, a popular o mDL é usado o seguinte ficheiro JSON:

```
{
    "dg1": {
        "family_name": "Smithe Williams",
```

```
        "name": "Alexander George Thomas",
        "date_of_birth": "19700301",
        "date_of_issue": "20020915",
        "date_of_expiry": "20070930",
        "issuing_country": "JPN",
        "issuing_authority": "HOKKAIDO PREFECTURAL POLICE
           ASAHIKAWA AREA SAFETY PUBLIC",
        "license_number": "A290654395164273X",
        "number_of_entries": 2,
        "categories_of_vehicles": [
            "C1;20000315;20100314;S01;<=;38303030",
            "C1;20000315;20100314;S01;<=;38303030"
        ]
    },
    "dg6": {
        "biometric_templates": [
            {
                "version": 257,
                "bdb_owner": 257,
                "bdb_type": 8,
                "bdb": "picture.jpg"
            },
            {
                "version": 257,
                "bdb_owner": 257,
                "bdb_type": 8,
                "bdb": "picture.jpg"
            }
        ],
        "number_of_entries": 2
    },
    "dg10": {
        "version": "1",
        "last_update": "20130115000000",
        "expiration_date": "20130314235959",
        "next_update": "20130122000000",
        "management_info": "info"
    },
    "ef_groupAccess": {
        "6": "75"
    },
    "ef_com": {
        "version": "0100",
        "tag_list": ["61", "75", "62"]
    },
    "ef_sod": {
        "digestAlgorithm": "id-sha256",
```

```
        "signatureAlgorithm": "id-pk-RSA-PSS-SHA256",
        "certificate": "certificates/certificate.der",
        "signature": "certificates/key.der"
    }
}
```

Além de popular os diferentes constituintes do mDL, o JSON apenas permite o acesso ao DG6 através do campo `ef_groupAccess`.

De seguida, encontra-se um exemplo de simulação, onde é feito uma correlação entre a interação e o processo de transação descrito no ISO.

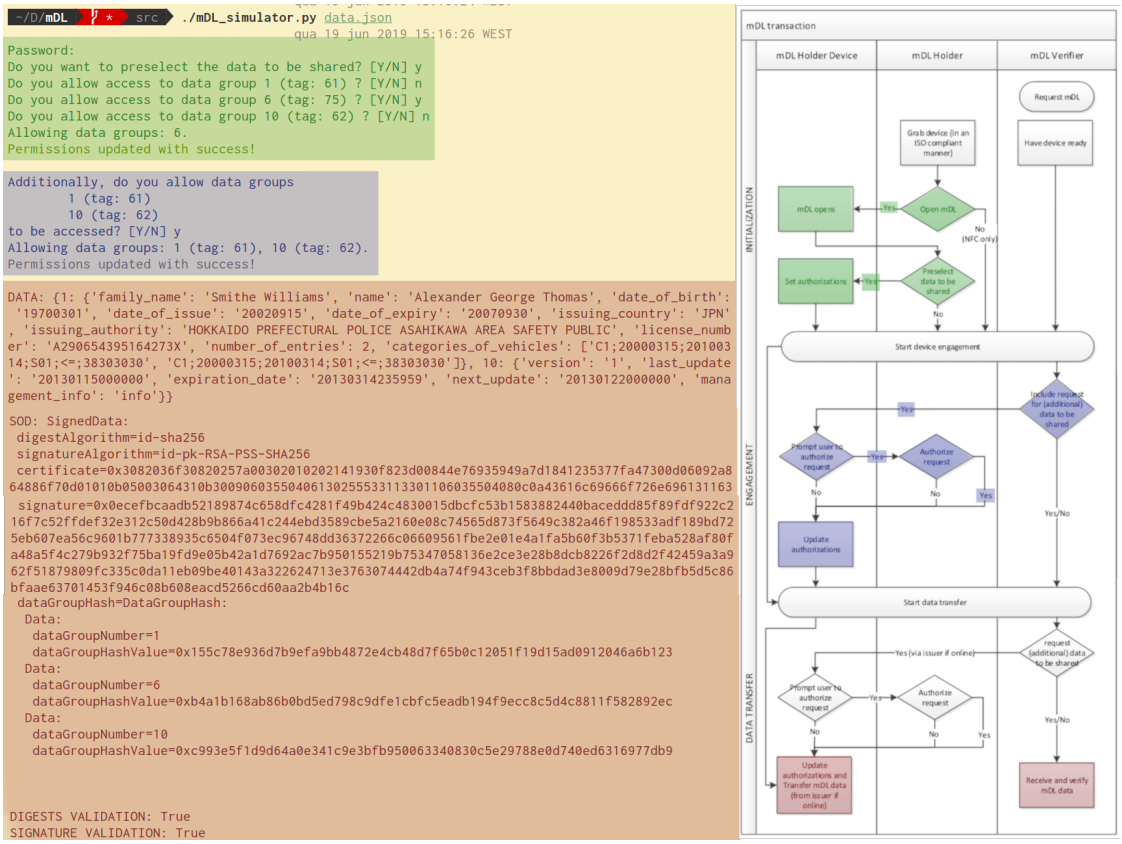



Figura 8: Exemplo de utilização do mDL.

Neste exemplo o utilizador aceita pre-selecionar os DG's que pretende permitir o acesso, sendo que apenas dá permissão ao DG6. De seguida, é requirido que seja dada permissão de acesso ao DG1 e DG10. O utilizador aceita e os dados são transferidos para o leitor do mDL, para que este possa prosseguir com a operação.

Importa realçar que o processo aqui descrito, que corresponde ao objectivo deste projeto, não envolve a componente de validação descrita no ISO do mDL. No entanto, esta foi adicionada de forma redimentar de forma a demonstrar o raciocínio necessário no momento de validção. Nesta pequena demonstração foram usados métodos e classes utilizados pelo mDL e, desta forma, teriam que ser reescritos pela entidade verificadora de forma a realizar a verificação de forma totalmente independente do mDL.

# 8 Conclusões

Este trabalho permitiu, através da análise do ISO de desmaterialização da carta de condução, perceber quais os principais constituintes da mesma, bem como os processos necessários para garantir a segurança dos dados, quer no seu armazenamento, quer na sua transmissão. Contudo, algumas das componentes do `mDL` não foram exploradas ao pormenor, nem implementadas, uma vez que eram opcionais ou envolviam a utilização de dispositivos físicos que não entravam no âmbito deste projeto. Para além disso, verificaram-se algumas dificuldades na compreensão de determinadas componentes do ISO, devido à sua especificidade e constante referência a *standards* externos.

De seguida, na fase de implementação, escolheu-se a linguagem `Python` devido à disponibilidade de várias bibliotecas criptográficas e à facilidade/rapidez de desenvolvimento associada. No entanto, encontraram-se dificuldades na codificação `ASN.1` exigida pelo ISO, uma vez que as bibliotecas já existentes não respeitavam todos os requisitos impostos. Desta forma, houve o trabalho adicional de desenvolver um *parser* genérico, capaz de converter as informações dos DG's para o formato hexadecimal desejado. Em termos de mecanismos de segurança, teve de se criar certificados, bem como gerar os *digests* de cada DG e a assinatura do conjunto dos mesmos. Para exemplificar a utilização da estrutura de dados desenvolvida (o `mDL`), animou-se uma transação de dados entre o titular e um verificador, tendo em conta o processo descrito no ISO.

Por fim, salienta-se que se deverá implementar, como trabalho futuro, a incorporação dos conjuntos de dados não considerados ou de *compact encoding* (para comunicação entre dispositivos com reduzidos recursos de armazenamento).

# Referências

[1] Information technology – personal identification – iso-compliant driving licence – part 1: Physical characteristics and basic data set.

[2] Information technology – personal identification – iso-compliant driving licence – part 2: Machine-readable technologies.

[3] Information technology – personal identification – iso-compliant driving licence – part 3: Access control, authentication and integrity validation.

[4] Personal identification – iso compliant driving licence – part 5: Mobile driving licence.