A dark blue vertical bar runs down the left side of the page. A blue arrow points to the right from this bar, containing the date.

13/04/2021

Projet carte bancaire

Several thin, curved lines in dark blue and light grey originate from the bottom left corner and sweep upwards and to the right.

Marc RENARD

Table des matières

Introduction.....	2
I) Matériel.....	2
II) APDU	4
III) PROGMEM et EEPROM	6
A. PROGMEM.....	6
B. EEPROM.....	7
IV) ACID.....	9
V) Sécurité, TEA	11
A. TEA.....	11
B. Stratégie	12
C. Vulnérabilités.....	15
VI) Quelques tests	16
A. Personnalisation	16
B. Crédit	17
C. Débit	19

Introduction

La carte à puce est la descendante directe des cartes à piste magnétique. Cette dernière fut déjà une révolution en soit, permettant l'utilisation d'un code confidentiel et évitant aux commerçants d'avoir à transmettre aux banques les relevés de transaction. Ces cartes bleues, c'est leur nom (et leur couleur) d'origine ont été utilisées à partir de 1971. Elles ne furent utilisées au début que pour les retraits dans les distributeurs automatiques, et c'est seulement à la fin des années 70 qu'elle fut utilisée pour des paiements électroniques. Mais malgré les prouesses de ces cartes à piste magnétique, les transactions restaient très lentes du fait des capacités du réseau de télécommunication à cette époque.

Au début de l'année 1974, Roland Moreno, un inventeur français découvre l'existence de mémoire qui conserve l'information enregistrée sans apport d'énergie. Le 25 mars 1974 il dépose le premier brevet d'une longue série pour sa puce électronique. Roland Moreno dira alors : « La carte à puce est ma première invention utile qui remplit une fonction souhaitable, sécuriser l'argent ».

Cette carte fut une révolution, car grâce à celle-ci, il n'était plus nécessaire de joindre la banque à chaque transaction et le temps de réalisation d'un paiement électronique fut ainsi drastiquement réduit. Son utilisation sera démocratisée au début des années 80 et se diversifiera outre les cartes bancaires, comme par exemple pour la carte vitale ou les cartes de téléphone.

1) Matériel

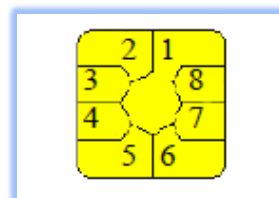
La carte en elle-même est le plus souvent en plastique, et ses dimensions sont fixées par la norme ISO 7816-1 :

- 85mm de longueur
- 54mm de largeur
- 0,76mm d'épaisseur

Cette norme a été créée pour que l'utilisation de carte à puce puisse être universelle et internationale.

L'ISO 7816-2 fixe la norme des contacts de la manière suivante :

- 1 et 2 : alimentation
- 3 : horloge
- 4 : remise à zéro
- 5 et 6 : optionnels
- 7 : entrée/sortie
- 8 : écriture dans l'EEPROM



L'ISO 7816-3 fixe les caractéristiques électriques, la réponse au reset (ATR Answer To Reset) et très important, le protocole de transmission TPDU (Transmission Protocol Data Unit).

Si T=0 le protocole est orienté octet, alors que si T=1 le protocole est orienté paquet. Ceci change complètement la manière dont la carte communique avec le lecteur.

Le protocole utilisé dans ce projet est le protocole orienté octet, ainsi les deux fonctions suivantes permettront la communication entre la carte et le lecteur :

- sendbyte0 permet d'envoyer un octet de la carte vers le lecteur
- recbyte0 permet à la carte de lire un octet envoyé par le lecteur

La norme ISO 7816 contient aujourd'hui 15 sous parties, mais nous ne les détaillerons pas toutes.

La carte à puce utilisée pour ce projet suit l'architecture Harvard, qui contrairement à l'architecture Von Neumann, possède deux mémoires de stockage séparées auxquelles le microprocesseur accède par des bus distincts. Ces deux mémoires sont la mémoire programme et la mémoire de données.

La mémoire programme est de type ROM(Read Only Memory). La ROM, aussi appelée mémoire morte, est un type de mémoire non volatile, c'est-à-dire une mémoire qui ne s'efface pas lorsque l'appareil qui la contient n'est plus alimenté en électricité. De plus, comme son nom l'indique, cette mémoire est en lecture seule, il est impossible de la modifier. Elle est donc définie et fixée lors de sa création et ne sera plus jamais modifiée par la suite.

L'autre mémoire de stockage de données est l'EEPROM. L'EEPROM se rapproche de la ROM à la différence près de la possibilité d'effacer le contenu de celle-ci lorsqu'elle est alimentée en électricité. Elle se comporte comme une ROM lors de sa perte d'alimentation : elle conserve en mémoire les données qu'elle contenait. Elle se distingue de la EPROM, qui elle pour être effacée, devait être sortie de l'appareil, et être exposée aux UV.

Malgré leur avantage de conserver l'information après coupure d'alimentation, ces deux mémoires ont un défaut : le temps nécessaire pour y accéder. Ainsi il existe un autre type de mémoire au sein de la carte : la RAM. La RAM aussi appelée mémoire vive, s'oppose à la mémoire morte. Si l'appareil qui la contient n'est plus alimenté en électricité, les données contenues dans la RAM sont perdues. On pourrait donc croire qu'elle n'apporte pas un grand intérêt comparé à la ROM ou l'EEPROM mais pourtant si, car la vitesse d'accès à une mémoire RAM est bien plus rapide que l'accès à une mémoire morte. C'est donc dans cette mémoire que l'on fait toute la partie calcul et que sont stockées les variables « temporaires » Cette mémoire est accessible en lecture et en écriture.

II) APDU

APDU signifie Application Protocol Data Unit, c'est un protocole qui définit la structure d'un message échangé entre une carte à puce et le lecteur dans lequel elle se trouve.

Un message APDU se compose d'au moins 5 octets qui sont les suivants :

CLA – INS – P1 – P2 – P3

CLA désigne la classe d'instruction, une sorte de catégorie d'instructions.

INS désigne l'instruction en elle-même.

P1 et P2 sont des paramètres d'instruction. Ils ne serviront pas dans ce projet.

Et enfin P3 qui peut avoir plusieurs rôles différents :

- P3 peut désigner le nombre d'octets à envoyer, dans ce cas P3 sera suivi de ce nombre d'octets dans la commande. C'est une commande d'envoi de données.
- Si P3 n'est suivi d'aucun octet, c'est une commande de requête de données et dans ce cas, P3 désigne le nombre d'octets attendus dans la réponse.
- Il existe une configuration qui combine les deux cas précédents (envoi de données, et récupérations de données. Dans ce cas la structure est la suivante :

CLA – INS – P1 – P2 – Lc – DATA₁ - ... - DATA_{Lc} - Le

Lc désigne le nombre d'octets à envoyer, et Le désigne le nombre d'octets attendus en retour.

Cette dernière configuration ne sera pas utilisée dans ce projet.

La norme APDU définit aussi une structure pour la réponse :

DATA₁ - ... - DATA_{Le} – SW1 – SW2

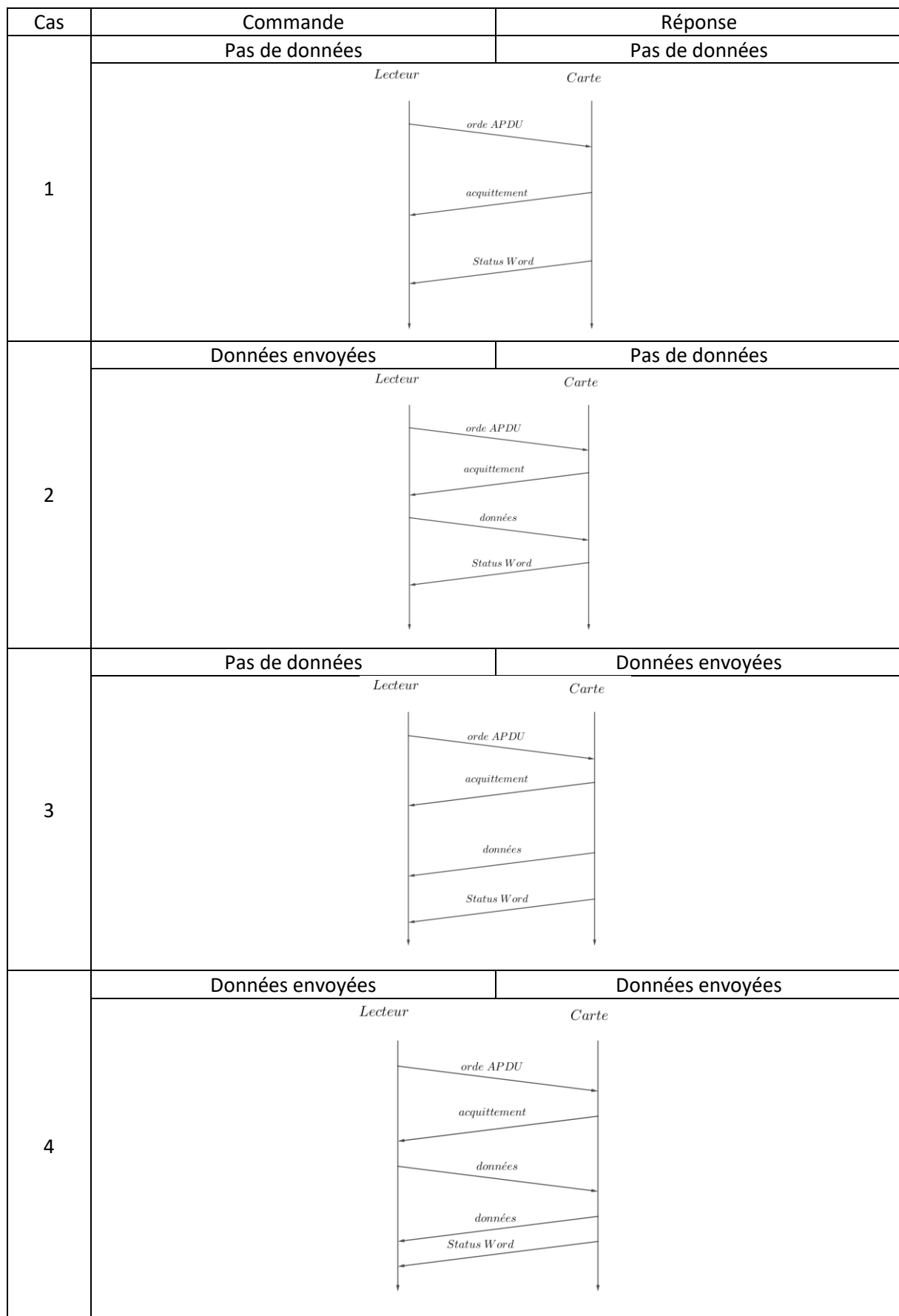
S'il n'y a pas de données attendues en retour, seuls SW1 et SW2 sont envoyés en réponse.

SW signifie Status Word, ils désignent des codes permettant de savoir si l'exécution s'est déroulée correctement.

Par exemple :

- 90 00 signifie que tout s'est déroulé correctement
- 6D xx signifie que le P3 est incorrect et qu'il doit être égal à xx.

Voici l'ordre d'exécution dans les différents cas :



III) PROGMEM et EEPROM

A. PROGMEM

PROGMEM est un attribut de la bibliothèque avr/pgmspace.h, bibliothèque qui contient les fonctions nécessaires à la gestion de la mémoire programme.

La mémoire programme étant en lecture seule, cette bibliothèque ne contient que des fonctions de lecture. Ceci implique que chaque objet stocké en mémoire programme doit être initialisé dès la déclaration, car ces objets sont constants et ne pourront plus être modifiés par la suite.

Voici un exemple de déclaration d'une variable en mémoire programme :

```
#include <avr/pgmspace.h>

#define size_atr 6
const char atr_str[size_atr] PROGMEM = "bourse";
```

Code 1 : Écriture d'une variable en mémoire programme

La fonction principale de la bibliothèque avr/pgmspace.h utilisée dans ce projet est la fonction qui permet de lire un octet :

uint8_t pgm_read_byte(adresse)

Cette fonction lit et renvoie l'octet qui se trouve à l'adresse indiquée.

Voici un exemple d'utilisation :

```
void atr()
{
    int i;
    sendbytet0(0x3b); // définition du protocole
    sendbytet0(size_atr); // nombre d'octets d'historique
    for (i=0; i<size_atr; i++) // Boucle d'envoi des octets d'historique
    {
        sendbytet0(pgm_read_byte(atr_str+i));
    }
    valide();
}
```

Code 2 : Lecture en progmem

On remarque que dans la boucle for, l'adresse est incrémentée à chaque passage, ce qui permet d'envoyer octet par octet le contenu de atr_str (ici cela envoie lettre par lettre le mot bourse déclaré précédemment).

Test d'interaction avec la carte :

```
1 Scat version 1.16
2 1 lecteur-s connecté-s
3 0 : SCM Microsystems Inc. SCR 355 [CCID Interface] 00 00
4 * reset
5 * reset
6 3.444 > 3b 0b 48 65 6c 6c 6f 20 73 63 61 72 64 ;.Hello scard
7 0.000 > 3b 0b 48 65 6c 6c 6f 20 73 63 61 72 64 ;.Hello scard
8 * 80 00 00 00 04
9 * 80 00 00 00 04
10 15.577 < 80 00 00 00 04 ♦....
11 \\ La commande envoyée fait appel à version qui a été modifiée pour lire dans
12 \\ la mémoire programme la chaîne de caractère : "1.00"
13 15.589 > 31 2e 30 30 90 00 1.00♦.
14 \\ le retour est bien 1.00 qui a été lu dans la mémoire programme.
15 exécution normale car 90 00|
```

Exécution 1 : Lecture en progmem

B. EEPROM

Tout comme pour la mémoire programme, il existe une bibliothèque qui permet de gérer la mémoire EEPROM : `avr/eeprom.h`

Voici les fonctions les plus utilisées de cette bibliothèque :

`void eeprom_write_byte(uint8_t* __p, uint8_t __value)`

- écrit à l'adresse p dans l'eeprom la valeur value de taille 1 octet

`void eeprom_write_block (const void* __src, void* __dst, size_t __n)`

- recopie un bloc de n octets depuis src vers dst

`uint8_t eeprom_read_byte(const uint8_t* __p)`

- lit et renvoie un octet à l'adresse p dans l'eeprom

`uint16_t eeprom_read_word(const uint16_t* __p)`

- lit et renvoie un mot de deux octet à l'adresse p dans l'eeprom
- retour en little endian

`void eeprom_read_block(void* __dst, const void* __src, size_t __n)`

- lit n octets à l'adresse __src dans l'eeprom et les écrit à l'adresse dst en dehors de l'eeprom

Exemple 1 : Code

```
#include <avr/eeprom.h>

uint8_t unEntier EEMEM;
uint16_t unMot EEMEM;

eeprom_write_byte(&unEntier,5); //passe l'entier à 5 dans l'eeprom
int a=eeprom_read_byte(&unEntier); //a prend la valeur de unEntier ie 5
uint16_t b=0x1234;
eeprom_write_block(&b,&unMot,2); //écrit b dans unMot dans l'eeprom
uint16_t c;
c=eeprom_read_word(&unMot); //c contient unMot en little endian
eeprom_read_block(&c,&unMot,2); //fait exactement la même chose
```

Code 3 : Lecture/écriture eeprom

Il est important de noter qu'il est nécessaire de déclarer une variable dans la RAM avant de l'écrire en EEPROM, car la fonction d'écriture requiert une adresse en entrée.

Exemple 2 : Test d'interaction avec la carte

```
//Déclaration de variables eeprom pour le solde
//initialisation du solde chiffré
uint32_t solde1 EEMEM=0x9c45df56;
uint32_t solde2 EEMEM=0x7194cb80;
```

```
313 void recupererSolde(uint32_t *clef1, uint16_t *destination){//cette fonction est implémentée à part car elle va servir dans lireSolde, dans credit et dans debit.
314     uint32_t s[2]={0,0};
315     for(int i=0;i<4;i++){ //lecture de solde1
316         s[i]=eeprom_read_byte((uint8_t*) &solde1 + 3 - i);
317         if(i!=3){
318             s[i]<<=8;
319         }
320     }
321     for(int i=0;i<4;i++){ //lecture de solde2
322         s[i]=eeprom_read_byte((uint8_t*) &solde2 + 3 - i);
323         if(i!=3){
324             s[i]<<=8;
325         }
326     }
327     uint32_t cleCode[2]={0,0};
328     tea_dechiffre(s,cleCode,clef1);
329     *((uint8_t*)(destination))=*((uint8_t*)(cleCode));
330     *((uint8_t*)(destination) + 1 )=*((uint8_t*)(cleCode + 1));
331 }
332
333
334 // lecture du solde
335 void lireSolde(uint32_t *k){
336     if (p3!=2)
337     {
338         sw1=0x6C; // taille incorrecte
339         sw2=2; // taille attendue
340         return;
341     }
342     sendbytet0(1ns);
343
344     uint16_t soldeLu=0;
345     recupererSolde(k,&soldeLu);
346
347     //Fin lecture, ici le solde est en clair
348     sendbytet0*((uint8_t*)&soldeLu + 1);
349     sendbytet0*((uint8_t*)&soldeLu);
350     taille = p3;
351     sw1=0x90;
352     sw2=0;
353 }
354 }

case 0x81:
    if(verrou==0){break;} //on empêche d'interagir avec la carte
    switch(1ns)
    {
    case 2:
        introOwner();
        break;
    case 3:
        showOwner();
        break;
    case 4:
        lireSolde(key);
        break;
```

Code 4 : Lecture/écriture eeprom

```
> 81 04 00 00 02  
< 00 64 90 00 : Normal processing.
```

Exécution 2 : Réponse lireSolde

La commande correspond à lire solde, et celle-ci fait appel à récupérer solde qui va récupérer le solde chiffré (qui est en EEPROM) à l'aide des deux boucles for.

IV) ACID

ACID signifie Atomicité, Cohérence, Isolation et Durabilité. Ce sont des propriétés qui garantissent qu'une transaction informatique soit exécutée de façon fiable.

L'atomicité assure que la transaction soit faite en totalité ou pas du tout. Elle empêche qu'une transaction soit faite partiellement. Si c'est le cas et que la transaction n'est pas terminée complètement, alors toute trace en est effacée, et tout retourne dans l'état d'avant le début de la transaction.

Cette propriété est nécessaire pour contrer d'éventuels problèmes tels que la panne d'alimentation, ou l'arrachage de la carte du lecteur.

La cohérence assure que la transaction fait passer le système d'un état valide à un autre état valide.

Ceci est nécessaire par exemple pour éviter de retirer plus d'argent que ce qui est disponible.

L'isolation est le principe suivant : toute transaction doit s'exécuter de manière indépendante des éventuelles autres transactions. Si deux transactions s'exécutent en même temps, cela doit donner le même résultat que si elles s'exécutent en chaîne.

La durabilité assure que lorsqu'une transaction a été confirmée, elle demeure enregistrée même à la suite d'une panne.

Pour assurer ces propriétés, nous utilisons deux fonctions : engage et valide.

Engage prends $3k+1$ arguments, ils vont par groupe de trois, et un 0 est nécessaire à la fin pour signaler la fin des arguments.

Les groupes de trois arguments sont composés de :

- une taille de données
- une adresse source
- une adresse de destination dans l'EEPROM.

La fonction engage va alors écrire dans une structure définie dans l'EEPROM, ces différentes informations. On remarquera que l'adresse source n'est pas mémorisée dans la structure, elle sert juste lors de la copie des données vers la structure dans l'EEPROM.

La fonction valider va alors vérifier l'état de cette structure, et si elle contient des données (état non vide) elle va se charger de les écrire au bon endroit dans l'EEPROM, en utilisant la taille, l'adresse de destination et les données à écrire qui se trouvent dans le buffer.

Voici la structure et un exemple d'engagement /validation :

Structure :

```
// nombre maximal d'opérations par transaction
#define max_ope 3
// taille maximale totale des données échangées lors d'une transaction
#define max_data 64
// définition de l'état du buffer -- plein est une valeur aléatoire
typedef enum(vide=0, plein=0x1c) state_t;
// la variable buffer de transaction mémorisée en eeprom
struct
{
    state_t state; // état
    uint8_t nb_ope; // nombre d'opération dans la transaction
    uint8_t tt[max_ope]; // table des tailles des transferts
    uint8_t *p_dst[max_ope]; // table des adresses de destination des transferts
    uint8_t buffer[max_data]; // données à transférer
}
ee_trans EEMEM={vide}; // l'état doit être initialisé à "vide"
```

Code 5 : Structure utilisée par engage

Engagement/validation :

```
engage(p3,data,proprietaire,1,&p3,&sizeProp,0);
```

```
valide();
```

Analyse des paramètres :

- premier groupe de 3 arguments :
 - o copie de p3 octets
 - o qui se trouvent à l'adresse data (data étant un tableau, c'est bien un pointeur)
 - o cette copie va se faire dans propriétaire (qui lui aussi est un tableau donc un pointeur)
- second groupe de 3 arguments :
 - o copie de 1 octets
 - o on veut copier la valeur de p3, donc on donne à la fonction l'adresse de p3
 - o cette copie se fera à l'adresse de sizeProp
- le 0 pour signaler la fin des arguments

À l'intérieur de la fonction atr, valide est appelée. Ainsi, si une transaction a été coupée par une panne quelconque avant qu'elle soit terminée, le reset au redémarrage lancera la fonction valide et terminera les transactions inachevées.

V) Sécurité, TEA

A. TEA

Le TEA (Tiny Encryption Algorithm) est un algorithme de chiffrement par bloc. Son principe de fonctionnement est basé sur les réseaux de Feistel, présenté pour la première fois en 1994 au salon Fast Software Encryption par David Wheeler et Roger Needham, ses créateurs, deux informaticiens de Cambridge.

Un réseau de Feistel utilise des principes comme les permutations, les substitutions, les échanges de blocs et utilise une clé à chacune des sous étapes (appelées tour) du chiffrement. Le TEA réalise 32 tours lors du chiffrement/déchiffrement.

Le TEA sert à chiffrer des mots de 64 bits, séparés en deux mots de 32 bits, et utilise pour cela une clé de 128 bits.

Le TEA a un avantage : sa simplicité d'implémentation. L'algorithme ne fait que quelques lignes, et n'utilise que des sommes, des décalages et des XOR. C'est donc un avantage considérable pour une utilisation en électronique embarquée, domaine dans lequel la mémoire est un facteur limitant.

Cependant, cette simplicité lui confère aussi des inconvénients. L'un d'eux est le fait que chaque clé est équivalente à trois autres (4 clefs différentes, mais qui donnent le même chiffrement), ce qui divise par 4 le nombre de clefs « réellement » différentes et abaisse la taille « réelle » des clefs à 126 bits. Le TEA est aussi vulnérable aux attaques par clefs apparentées.

C'est pour palier à ces faiblesses que le XTEA (eXtend TEA) a été créé (par les mêmes informaticiens) et présenté en 1997. Celui-ci fonctionne toujours sur des blocs de 64 bits, avec une clef de 128 bits et sur le principe des réseaux de Feistel. Le XTEA résout le problème attaques par clefs apparentées et celui des clefs équivalentes, et donc deux clefs différentes, donneront un chiffré différent sur la base d'un même mot clair.

Une variante du XTEA est le Block TEA, celui-ci permet de chiffrer des informations de taille quelconques en les chiffrant par bloc. Mais celui-ci est aussi vulnérable à certaines attaques.

Enfin, le successeur du Block TEA est le XXTEA, créé encore une fois par les mêmes informaticiens de Cambridge en 1998.

Le XXTEA travaille sur des blocs de taille supérieur à 64 bits et multiples de 32 bits. De plus, il repose sur un réseau de Feistel non symétrique : contrairement à ses prédécesseurs qui prenaient en entrée un clair réparti en deux mots de 32 bits, le XXTEA sépare le clair en deux mots de taille différente. Ceci augmente la sécurité du chiffrement, mais nécessite un nombre de tours plus conséquents qui dépend de la taille des blocs.

Malheureusement, malgré les différentes améliorations successives, le XXTEA s'avère vulnérable aux attaques à chiffré choisi comme il l'a été montré en 2010 dans la publication de E.Yarrikov.

B. Stratégie

Pour ne garder que des soldes chiffrés en EEPROM nous allons utiliser l'algorithme TEA.

Le solde ne fait que 16 bits, et TEA prends en paramètre deux entiers de 32 bits.

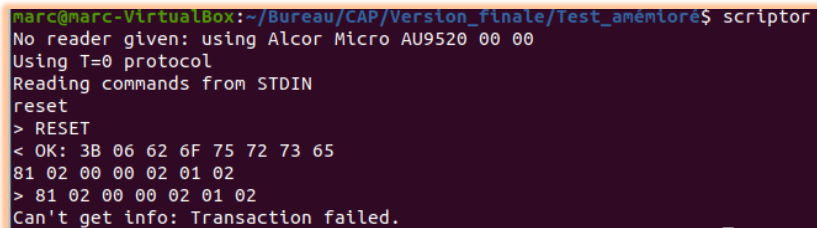
Ce que nous proposons de faire est de concaténer le solde initial SI avec lui-même pour avoir un entier de 32 bits SISI et ensuite de chiffrer avec une clé prédéfinie (qui n'apparaîtra jamais en EEPROM ni PROGMEM) la paire (SISI,SISI) pour l'écrire en EEPROM.

La boucle infinie qui récupère les instructions dans le main ne permet initialement que trois commandes (classe 80):

- Afficher la version
- Introduire une clef
- Tester la validité d'une clef

Toute tentative d'entrer une commande autre, avant d'avoir entré et testé une clef valide échouera.

Exemple :



```
marc@marc-VirtualBox:~/Bureau/CAP/Version_finale/Test_amélioré$ scriptor
No reader given: using Alcor Micro AU9520 00 00
Using T=0 protocol
Reading commands from STDIN
reset
> RESET
< OK: 3B 06 62 6F 75 72 73 65
81 02 00 00 02 01 02
> 81 02 00 00 02 01 02
Can't get info: Transaction failed.
```

Exécution 3 : Tentative d'accès à commande verrouillé

Dans cet exemple, on essaye d'entrer une commande de la classe 81 juste après le RESET, et donc sans entrer de clef. La commande est rejetée et l'exécution s'arrête.

Une fois la clé entrée, pour tester sa validité, on va déchiffrer le solde et vérifier que le déchiffré est de la forme (SISI,SISI) (en testant l'égalité des deux), si ce n'est pas le cas la clé est fausse, on décrémente le nombre d'essais restants et on l'écrit dans l'EEPROM par engage suivi de valide. Si la clef est valide alors on déverrouille l'accès à toute la boucle (les instructions dont la classe est autre que 80). De plus si la clé est valide, on remet le compteur d'essais restants à 3.

Code et tests compteur :

Le code suivant se trouve dans la fonction testClef, et représente le test effectué.

```
490     int rep=1;
491     int cmptTmp=eeprom_read_byte(&compteurEssai); //r
492     //On va vérifier si le résultat du déchiffrement
493     if(code[0]!=code[1]){ //Si le code est erroné, o
494         rep=0;
495         cmptTmp--;
496     }else{ //ici on a l'égalité entre code[0] et cod
497         dans cette partie du test)
498         if((code[0]>>16)!=(code[0]&0x0000ffff)){
499             rep=0;
500             cmptTmp--;
501         }else{//Si le code est valide, on repasse
502             cmptTmp=3;
503         }
504     }
505     engage(1,&cmptTmp,&compteurEssai,0);
506     valide();
```

Code 6 : Test de validité de la clef

```
marc@marc-VirtualBox:~/Bureau/CAP/Version_finale/Test_amélioré$ scriptor
No reader given: using Alcor Micro AU9520 00 00
Using T=0 protocol
Reading commands from STDIN
reset
> RESET
< OK: 3B 06 62 6F 75 72 73 65
80 01 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00
> 80 01 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00
< 90 00 : Normal processing.
80 02 00 00 02
> 80 02 00 00 02
< 00 02 90 00 : Normal processing.
80 01 00 00 10 11 11 11 11 11 11 11 11 11 11 11 11 11
> 80 01 00 00 10 11 11 11 11 11 11 11 11 11 11 11 11 11
< 90 00 : Normal processing.
80 02 00 00 02
> 80 02 00 00 02
< 00 01 90 00 : Normal processing.
80 01 00 00 10 54 BA 93 54 D5 67 6C E6 4C A7 CF 93 83 71 B9 30
> 80 01 00 00 10 54 BA 93 54 D5 67 6C E6 4C A7 CF 93 83 71 B9 30
< 90 00 : Normal processing.
80 02 00 00 02
> 80 02 00 00 02
< 01 03 90 00 : Normal processing.
```

Exécution 4 : Tests compteur et validité de clef

On entre la clef composée uniquement de 0 (80 01 00 00 10 00 00 ...) puis on la teste (80 02 00 00 02). Le résultat est 00 02 90 00. Le premier octet (00) signifie que la clef est incorrecte. Le deuxième octet (02) signifie qu'il reste 2 tentatives pour entrer une clef valide. 90 00 signifie que l'exécution s'est correctement déroulée.

Même procédure avec la clef composée uniquement de 1. On obtient 00 01 90 00 au test. Cette clef est donc fausse, et il ne reste plus qu'une seule tentative pour entrer une clef valide.

On entre enfin une clef valide (54 BA 93 54 D5 67 6C E6 4C A7 CF 93 83 71 B9 30) et on obtient au résultat du test : 01 03 90 00. La clef est donc valide (01) et on remarque que le nombre de tentatives restantes est bien repassé à 3 (03).

Si on essaye maintenant une commande de classe 81 (lire solde par exemple) :

```
> 80 02 00 00 02
< 01 03 90 00 : Normal processing.
81 04 00 00 02
> 81 04 00 00 02
< 00 64 90 00 : Normal processing.
```

On remarque que nous avons désormais accès à ces instructions qui nous étaient verrouillées au départ contrairement à : Exécution 3 : Tentative d'accès à commande verrouillé .

Par la suite, pour récupérer le solde, on déchiffrera le solde en EEPROM (en faisant attention au little endian) ce qui nous donnera une parie (SISI,SISI), on n'utilisera qu'un seul des deux. Par construction, SISI est la concaténation du solde avec lui-même, ainsi SISI>>16 nous donne le solde en clair SI.

```
313 void recupererSolde(uint32_t *clef1, uint16_t *destination){//cette
314     uint32_t s[2]={0,0};
315     for(int i=0;i<4;i++){ //lecture de solde1
316         s[0]+=eeprom_read_byte((uint8_t*) &solde1 + 3 - i);
317         if(i!=3){
318             s[0]<<=8;
319         }
320     }
321     for(int i=0;i<4;i++){ //lecture de solde2
322         s[1]+=eeprom_read_byte((uint8_t*) &solde2 + 3 - i);
323         if(i!=3){
324             s[1]<<=8;
325         }
326     }
327     uint32_t cleCode[2]={0,0};
328     tea_dechiffre(s,cleCode,clef1);
329     *((uint8_t*)(destination))=*((uint8_t*)cleCode));
330     *((uint8_t*)(destination) + 1)=*((uint8_t*)cleCode + 1));
331 }
```

Code 7 : Récupération de solde en clair

Il est alors possible de faire les calculs et les tests sur ce solde. Puis une fois les calculs terminés, on procédera de la même façon pour chiffrer le nouveau solde NS soit chiffrer (NSNS,NSNS) puis écrire le chiffré correspondant en EEPROM grâce à engage et valide. De cette manière, à la prochaine utilisation de la carte le test pour vérifier la validité de la clef est le même que le test initial, puisque le nouveau solde chiffré a été construit de la même manière que le solde chiffré initial.

```

385     uint16_t nouveauSolde=soldeActuel+soldeACrediter;
386     uint32_t soldeCodeInter=((uint32_t)nouveauSolde)<<16)+((uint32_t)nouveauSolde);
387     uint32_t soldeCode[2]={soldeCodeInter,soldeCodeInter};
388     uint32_t nouveauSoldeChiffre[2]={0,0};
389     tea_chiffre(soldeCode,nouveauSoldeChiffre,key1);
390     //engagement de l'opération d'écriture du nouveau solde
391     engage(4,(uint8_t*)nouveauSoldeChiffre,&solde1,4,(uint8_t*)nouveauSoldeChiffre + 4,&solde2,0);
392     taille=p3;           // mémorisation de la taille des données lues
393     sw1=0x90;
394     //validation de la transaction
395     valide();

```

Code 8 : Codage et chiffrement du solde (ici dans la fonction créditer)

Ce procédé permet de ne pas stocker la clé dans la carte, et exploite une structure dans les soldes codés (intermédiaires) avant chiffrement, qui ne se retrouve pas dans le solde chiffré.

C. Vulnérabilités

Lors d'une attaque en boîte noire (sans accès au code source) seuls 3 tentatives sont permises. Les clefs sont de taille 128 bits, il y a donc 2^{128} clefs au total. Cependant, comme il l'est expliqué dans ci-dessus, pour chaque clef, 4 clefs sont équivalentes. Il n'y a donc réellement que 2^{126} clefs distinctes (on peut facilement tester si deux clefs sont distinctes en les utilisant pour chiffrer un même message clair et en comparant les chiffrés). La probabilité de tomber sur une clef valide au hasard parmi les trois essais est donc de $\frac{3}{2^{126}} \approx 3.10^{-38}$. La probabilité de trouver une clef valide au hasard est donc très faible.

Cependant, lors d'une attaque en boîte blanche (accès au code), on peut tout simplement retirer la partie compteur qui verrouille la carte après trois essais infructueux et ensuite essayer des clefs par force brute.

VI) Quelques tests

A. Personnalisation

```
241 //déclaration du propriétaire dans l'EEPROM
242 char proprietaire[20] EEMEM;
243 uint8_t sizeProp EEMEM;
244
245
246 void introOwner()
247 {
248     int i;
249     // vérification de la taille
250     if (p3>20)
251     {
252         sw1=0x6c;        // P3 incorrect
253         sw2=20;          // sw2 contient l'information de la taille correcte
254         return;
255     }
256     sendbytet0(1);        // acquittement
257     for(i=0;i<p3;i++)      // boucle d'envoi du message
258     {
259         data[i]=recbytet0();
260     }
261     //écriture en eeprom du propriétaire
262     //eeprom_write_block(data,proprietaire,p3);
263     //eeprom_write_byte(&sizeProp,p3);
264     engage(p3,data,proprietaire,1,&p3,&sizeProp,0);
265     taille=p3;            // mémorisation de la taille des données lues
266     sw1=0x90;
267     valide();
268 }
269 }
```

Code 9 : Introduction du nom du propriétaire

```
272 // lecture perso
273 void showOwner(){
274     uint8_t size;
275     size=eeprom_read_byte(&sizeProp);
276     if (size==0){
277         sw1=0x61;
278         sw2=0;
279         return;
280     }
281     if (p3!=size)
282     {
283         if(p3>MAXI){
284             sw1=0x6c;        // taille incorrecte
285             sw2=MAXI;        // taille attendue
286             return;
287         }
288         sw1=0x6c;        // taille incorrecte
289         sw2=size;        // taille attendue
290         return;
291     }
292     sendbytet0(1);
293     char lecture[p3];
294     eeprom_read_block(lecture,proprietaire,p3);
295     for(int i=0;i<p3;i++){
296         sendbytet0(lecture[i]);
297     }
298     taille = p3;
299     sw1=0x90;
300 }
```

Code 10 : Récupération et affichage du propriétaire

```

583             case 0x81:
584                 if(verrou==0){break;}
585                 switch(ins)
586                 {
587                     case 2:
588                         introOwner();
589                         break;
590                     case 3:
591                         showOwner();
592                         break;

```

Code 11 : Interaction avec le nom du propriétaire dans le main

```

81 02 00 00 03 AB C1 23
> 81 02 00 00 03 AB C1 23
< 90 00 : Normal processing.
81 03 00 00 03
> 81 03 00 00 03
< AB C1 23 90 00 : Normal processing.

```

Exécution 5 : Test entrée et affichage du propriétaire

B. Crédit

```

356 void crediter(uint32_t *key1){ //testée, se comporte correctement
357     if(p3!=2){
358         sw1=0x6c; //taille incorrecte
359         sw2=2;
360         return;
361     }
362     sendbytet0(ins);
363     uint8_t data[2];
364     for(int i=0;i<p3;i++) // boucle d'envoi du message
365     {
366         //récupération des octets du solde à créditer
367         data[i]=recbytet0();
368     }
369
370
371 //##### Récupération du solde chiffré
372     uint16_t soldeActuel;
373     recupererSolde(key1,&soldeActuel);
374
375     uint16_t soldeACrediter=0;!=(uint16_t)(data[1])+((uint16_t)(data[0])<<8);
376     *((uint8_t*)&soldeACrediter)=data[1];
377     *((uint8_t*)&soldeACrediter +1)=data[0];
378     if(soldeACrediter>(0xffff-soldeActuel)){
379         //le solde max va être dépassé
380         //code erreur: out of boundary
381         sw1=0x91;
382         sw2=0xBE;
383         return;
384     }
385     uint16_t nouveauSolde=soldeActuel+soldeACrediter;
386     uint32_t soldeCodeInter=((uint32_t)nouveauSolde)<<16)+((uint32_t)nouveauSolde);
387     uint32_t soldeCode[2]={soldeCodeInter,soldeCodeInter};
388     uint32_t nouveauSoldeChiffre[2]={0,0};
389     tea_chiffre(soldeCode,nouveauSoldeChiffre,key1);
390     //engagement de l'opération d'écriture du nouveau solde
391     engage(4,(uint8_t*)&nouveauSoldeChiffre,&solde1,4,(uint8_t*)&nouveauSoldeChiffre + 4,&solde2,0);
392     taille=p3; // mémorisation de la taille des données lues
393     sw1=0x90;
394     //validation de la transaction
395     valide();
396
397 }

```

Code 12 : Fonction crédit

```

583         case 0x81:
584             if(verrou==0){break;} /
585             switch(ins)
586             {
587                 case 2:
588                     introOwner();
589                     break;
590                 case 3:
591                     showOwner();
592                     break;
593                 case 4:
594                     lireSolde(key);
595                     break;
596                 case 5:
597                     crediter(key);
598                     break;
599                 case 6:
600                     debiter(key);
601                     break;

```

Code 13 : Instructions débit/crédit

```

> 81 04 00 00 02
< 00 64 90 00 : Normal processing.
81 05 00 00 02 ff 01
> 81 05 00 00 02 ff 01
< 90 00 : Normal processing.
81 04 00 00 02
> 81 04 00 00 02
< FF 65 90 00 : Normal processing.
81 05 00 00 02 f0 00
> 81 05 00 00 02 f0 00
< 91 BE : Error not defined by ISO 7816

```

Exécution 6 : Tests crédits et affichage

Description des commandes :

- 81 04 00 00 02 → affichage du solde : 00 64 initialement
- 81 05 00 00 02 ff 01 → crédit de ff 01
- 81 04 00 00 02 → affichage du nouveau solde ff 65
- 81 05 00 00 02 f0 00 → erreur (résultat hors des bornes). Il n'est pas possible de créditer cette somme, le résultat ne pourrait être stocké dans un uint16_t.

C. Débit

```
398 // débit
399 void debiter(uint32_t* clef1){
400     if(p3!=2){
401         sw1=0x6C; //taille incorrecte
402         sw2=2;
403         return;
404     }
405     sendbytet0(ins);
406     for(int i=0;i<p3;i++){ // boucle d'envoi du message
407         //récupération des octets du solde à créditer
408         data[i]=recbytet0();
409     }
410     //##### Récupération du solde chiffré
411     uint16_t soldeActuel;
412     recupererSolde(clef1,&soldeActuel);
413
414
415
416
417     uint16_t soldeADebiter=0;
418     *((uint8_t*)&soldeADebiter)=data[1];
419     *((uint8_t*)&soldeADebiter +1)=data[0];
420     if(soldeADebiter>soldeActuel){
421         //le solde est insuffisant
422         sw1=0x91;
423         sw2=0xBE;
424         return;
425     }
426     uint16_t nouveauSolde=soldeActuel-soldeADebiter;
427     uint32_t soldeCodeInter=((uint32_t)nouveauSolde)<<16)+((uint32_t)nouveauSolde);
428     uint32_t soldeCode[2]={soldeCodeInter,soldeCodeInter};
429     uint32_t nouveauSoldeChiffre[2]={0,0};
430     tea_chiffre(soldeCode,nouveauSoldeChiffre,clef1);
431     //engagement de l'opération d'écriture du nouveau solde
432     engage(4,(uint8_t*)&nouveauSoldeChiffre,&solde1,4,(uint8_t*)&nouveauSoldeChiffre + 4,&solde2,0);
433     taille=p3; // mémorisation de la taille des données lues
434     sw1=0x90;
435     //validation de la transaction
436     valide();
437 }
```

Code 14 : Fonction débit

```
> 81 04 00 00 02
< FF 65 90 00 : Normal processing.
81 06 00 00 02 0F 60
> 81 06 00 00 02 0F 60
< 90 00 : Normal processing.
81 04 00 00 02
> 81 04 00 00 02
< F0 05 90 00 : Normal processing.
81 06 00 00 02 f1 00
> 81 06 00 00 02 f1 00
< 91 BE : Error not defined by ISO 7816
```

Exécution 7 : Tests débit et affichage

Description des commandes :

- 81 04 00 00 02 → affichage du solde : ff 65
- 81 06 00 00 02 0f 06 → débit de 0f 06
- 81 04 00 00 02 → affichage du nouveau solde f0 05
- 81 06 00 00 02 f1 00 → erreur (résultat hors des bornes). Il n'est pas possible de débiter cette somme car le solde est insuffisant.