# RAPPORT CARTE CODE PIN/CODE PUK

## I – Introduction

Le projet de la carte code PIN/code PUK a pour objectif de simuler une carte SIM ainsi que toutes ses fonctionnalités, i.e le fait de sauvegarder un code PIN , le fait de changer ce dernier , le fait d'introduire le code PUK dans la carte si jamais on se trompe de code pin au bout d'un certains nombres de tentative ect... Pour cela, nous nous conformons sur le même modèle que le rapport sur le porte monnaie électronique avec la présence des cinq même variables globales (uint8_t cla, ins, p1, p2, p3), des variables EEPROM qui permettent de sauvegarder le code PIN, le code PUK et le nombre restant de tentative de l'utilisateur. La seul chose qui diffèrent est qu'on ajoute un type structuré énumérée(type énum state_t )qui nous renseignera sur l'état de la carte (si elle est vierge, verrouillé, déverrouillé ou bloqué).

Nous retrouvons également la même fonction main et ses composantes.Nous avons fait en sorte que la carte soit robuste a plusieurs quelques attaques, notamment celle de l'attaque temporelle et l'étude de la consommation de la carte (qu'on expliquera en quoi elle consiste ci-dessous ).

## II– Les commandes de la carte

Le nombres des commande sont en nombre de quatre :

### a) Vérification du code pin entré par l'utilisateur

Grâce à la fonction ' verify_pin() ', nous vérifions si le code pin entré par l'utilisateur est bien le bon. Pour cela nous commençons par regarder si l'état actuel de la carte est verrouillé et que le troisième paramètre de la commande correspondant à la taille des données entrante est valide (bien que la taille d'un code pin est de 4 octets , ici on l'a fixé à 8 car on se conforme à la norme ISO),si ce n'est pas le cas on envoie un message d'erreur sinon on envoie un acquittement puis on récupère le code de l'utilisateur via un recbytetv0 (comme dans le rapport précèdent ) et celui présent dans l'EEPROM (le code exact ) via un 'eeprom_read_block'. Puis une fois effectué ,on les compare grâce à la fonction 'compare' :elle prend en entrée 2 tableaux d'octets de taille 8 puis compare case par case les octets,dès que deux cases diffèrent elle renvoie 1 sinon 0.

```
int compare_fst(uint8_t test[8], uint8_t code[8])
{
        int i;
        for(i=0;i<8;i++)
        {
                if(test[i]!=code[i]) return 1;
         }
        return 0;
}
```

une fois les avoir comparé ,si l'utilisateur n'a pas rentré le bon code , on abaisse le nombre de tentative ( que je rappelle est une variable stockée dans l'EEPROM) puis on redemande à l'utilisateur de rentrer un nouveau code pin. Au bout de trois tentative si l'utilisateur ne parvient a rentré le bon code pin on met l'état à bloqué. L'indication du nombre de tentative restante à l'utilisateur est réalisé grâce au STATUS WORD sw2 en additionnant l'erreur au nombre restant de tentative.Si l'utilisateur réussi a mettre le bon code, on remet le nombre de tentative à la valeur 3 puis on met l'état à déverrouillé.

b) Changer le code pin actuel en un autre code pin

Si on veut maintenant changer notre code pin actuel, on utilisera la méthode 'change pin' qui vérifie d'abord si l'état de la carte déverrouillé (il faut en premier lieu déverrouillé la carte avant de pouvoir changer le code pin) et vérifie la taille des données ( ici 16 octets pour l'ancien + le nouveau code pin ). Ensuite, elle envoie un acquittement et récupère les huit premiers octets dans une variable et les huit derniers octets dans une autre. Elle compare si la première variable corresponds bien aux octets du code pin stocké dans la mémoire EEPROM , si c'est le cas, elle change le code pin actuel par le code pin de la deuxième variable sinon on effectue la même procédure que la fonction précédente en abaissant le nombre de tentative.

c) Débloquer la carte

Lorsqu'aux bout de trois tentatives l'utilisateur ne parvient pas à déverrouillé sa carte , la carte se bloque automatiquement et pour continuer à utiliser sa carte SIM , il doit exister un moyen pour la débloquer. Cette commande est faite pour répondre à ce besoin : elle invoque la méthode 'deblocage' . L'utilisateur doit entrer le code puk et ensuite le code pin en paramètres de la commande . La méthode va faire les vérifications habituelles ensuite récupérer le code puk de l'utilisateur , la comparer avec celle stocké dans la mémoire EEPROM, si elles sont identiques alors on récupère le nouveau code pin de l'utilisateur ,on l'a stocke dans la mémoire EEPROM puis on met l'état de la carte à verrouillé.

d) Introduction du code puk

Lorsqu'on reçoit la carte toute neuve , elle est dans un état vierge. Avant de pouvoir l'utiliser , on doit au préalable rentrer le code puk pour l'activer, initialiser le code pin à 0 ainsi mettre la variable EEPROM du nombre d'essai à 3.L'utilisateur pourra ensuite changer de code pin si il veut via la fonction 'change pin'.La dernière commande a pour rôle d'effectuer cette tâche en invoquant la fonction ' intro_puk()'.

Remarque : la fonction state_t get_state() qui renvoie  l'état de la carte au moment de l'arrachement (on est obligé de l'implémenter car la variable 'état' qui est du type 'state_t' est placé dans la RAM qui est une mémoire volatil) permet de synchroniser l'état de la carte sur la dernière connexion de la carte avant arrachement ( si elle a jamais été utilisés, la variable du nombre d'essai aura la valeur 0xff , c'est la valeur d'une variable EEPROM lorsqu'elle a jamais été initialisé, on renverra donc l'état vierge , si au moment de l'arrachement de la carte, le nombre d'essai était zéro alors lors du rebranchement de la carte ,on initialisera l'état de la carte à bloqué ect ...). Elle est placé au tout début de la fonction main afin que ce soit la toute première qui soit faite lors du branchement de la carte.

## III– Amélioration du code contre des éventuelle attaques

### a) Attaque temporelle

Nous avons donc maintenant toute les fonctionnalités et les outils pour jouir de notre carte SIM virtuelle. La seul chose qui nous reste à vérifier est de savoir si on peut l'utiliser en toute sécurité et on y regardant de plus près notre code, tout n'est pas si rose.

Particulièrement , en analysant la fonction 'compare' , on remarque lorsqu'on compare deux codes pin ou deux codes puk , on ne les testent pas de manière directement mais plutôt on les comparent octets par octets.En cas d'égalité, la carte passe à l'octet suivant Mais à partir du moment l'une des comparaison nous indiquent que les octets diffèrent, c'est là qu'on renvoie la valeur retour, i.e que les deux codes ne correspondent pas.

Donc le temps de réponse de la fonction ne sera pas la même pour deux codes qui diffèrent dès le premier octet par rapport à deux codes qui diffèrent seulement au huitième octet.

Une tierce personne qui aurait la malice de tester plusieurs combinaisons possibles tout en observant le temps de réponse de la fonction , augmente la probabilité d'avoir la bonne combinaison , surtout du code pin qui est ,en faite, fait de quatre octets (les quatre autres derniers étant que des zéros). Même si il faut relativiser , vu que le nombre de tentative est infiniment négligeable devant le nombre de combinaisons qui peut y avoir pour le code pin , le code puk peut être entrer autant de fois qu'on veut. Donc l'attaquant a tout intérêt de faire sa cryptanalyse sur le code puk en premier avant de s'attaquer au code pin.

Pour se prémunir de ce genre d'attaque, attaque dite temporelle, nous ré implémentons la fonction 'compare' de sorte qu'elle fonctionne en temps constant pour n' importe quelles codes en entrées. Pour cela nous comparant toujours octet par octet sauf nous envoyons la valeur retour une fois après avoir effectuer toute les comparaison de la manière suivante :

```
int compare_cst(uint8_t test[8], uint8_t code[8]) {
	int i;
	uint8_t code_bon=0;
	for(i=0;i<8;i++){
		code_bon|=test[i]^code[i];
	}
	 return code_bon;
}
```

la comparaison consiste a faire l'addition modulo 2 de chaque octet à comparer ( si ça vaut 0 , ce sont les même sinon si c'est 1 , c'est que il diffèrent ) puis de faire un « ou inclusive » entre tout ces additions là. Si tous ces octets se valent alors le résultats est 0 sinon si ,l'un d'entre eux diffèrent , automatiquement le résultat est de 1.

### b) Attaque sur la consommation

Cette attaque consiste à étudier la consommation électrique de la carte pour lui soutirer de l'information. Pour cela, on utilise le faite que cette consommation est fonction du nombre de mémoire qui change d'état. L'ensemble de départ de cette fonction est un octet ( 8 bits ) et le résultat associé est la mesure de la tension qui traverse une résistance du dispositif. Donc à un certain octet, on lui associe la consommation de la machine lors du traitement de cette octet. La consommation va dépendre du poids de Hamming de l'octet (en faite du nombre de transition de 0 à 1 ou de 1 à 0 )

Après plusieurs analyse algébrique, nous obtenons un modèle de consommation qui est le poids de Hamming des données traités par la cartes plus un erreur qui dépend de l'octet plus une constante k. Dans ce modèle, on a une inconnu k qu'on cherche à retrouver et cette valeur de k ,on la veut tel qu'elle minimise l'erreur. Quelle est la valeur de k pour laquelle le modèle se rapproche le plus de la mesure physique réalisé. Ce qui revient à maximiser la transformé de fourier de notre modèle de fonction.

Pour résumé,on a une consommation qu'on mesure à l'oscilloscope qui dépend de la valeur qu'on a affecté au code PUK (la valeur des donnés a traité ) . On calcule la transformé de fourier de toute les valeurs possibles du code PUK et le maximum de cette transformé , nous donnera la valeur de clé la plus probable.

Pour contrer cette attaque, il suffit de chiffrer nos données à traiter par une fonction a sens unique , i.e une fonction injective difficile à trouver un antécédent. Ceci nous permet, lors des comparaisons des données, de non plus comparer les valeurs de ces données mais leurs chiffrés et pour un potentiel attaquant , de ne pas avoir accès au valeur du code PUK.
La fonction a sens unique utilisés ici est la fonction qui à k associe le chiffrement d'un message quelconque (qu'on choisie au préalable) a la clé k.. Dans notre exemple utilisés est le chiffrement TEA.

Exemple : pour l'introduction d'un code PUK dans la mémoire EEPROM

```
for(i=0;i<8;i++) data[i]=recbytet0();


uint32_t tea_puk[2]={1,2};
uint32_t crypto[2];
uint32_t
k[4]={65536*data[0]+data[1],65536*data[2]+data[3],65536*data[4]+data[5],65536*data[6]+data[7
]};
tea_chiffre(tea_puk, crypto, k);
eeprom_write_block(crypto,ee_puk,8);
```

La fonction 'tea_chiffre' prend entrée le message qu'on va chiffrer , un tableau de 32 bits de taille 2 (la variable tea_puk[2] qu'on a affecté a une valeur prise de manière hasardeuse) , le chiffré en lui même (tableau de 32 bits de taille 2) et la valeur de la clé (tableau de 32 bits de taille 4 ) .Donc le code PUK qu'on récupérera sera décomposé 4 parties de 2 octets afin de pouvoir le stocker dans le tableau puis on conservera le chiffré dans la mémoire EEPROM.

Pour comparer deux codes PUK, on va récupérer les données de l'utilisateur , chiffré le message ( qui sera le même pour tout les chiffrement effectué ) avec comme clé la valeur des données puis comparé le chiffré avec celui présent dans l'EEPROM.

```
 uint32_t k[4]={65536*puk[0]+puk[1],65536*puk[2]+puk[3],65536*puk[4]+puk[5],
65536*puk[6]+puk[7]};

tea_chiffre(tea_puk, crypto, k);

uint8_t cryptoN[8]={crypto[0]>>24,(crypto[0]&0b11111111000000000000000)>>16
,(crypto[0]&0b1111111100000000)>>8,crypto[0]&0b11111111,crypto[1]>>24,
(crypto[1]&0b11111111000000000000000)>>16,
(crypto[1]&0b1111111100000000)>>8,crypto[1]&0b11111111};


uint8_t dataN[8]={data[0]>>24,(data[0]&0b11111111000000000000000)>>16,
(data[0]&0b1111111100000000)>>8,data[0]&0b11111111,data[1]>>24,
(data[1]&0b11111111000000000000000)>>16,(data[1]&0b1111111100000000)>>8
data[1]&0b11111111} ;

if(compare(cryptoN,dataN)!=0){sw1=0x98; sw2=4; return;};
```

## IV – Stratégies de test

Pour le test , on a commencer par tester si la fonction 'void intro_puk() ' fonctionné. Pour cela on a essayé d'introduire le code '12345678' via le script comme tel :

```
# perso puk
a0 40 00 00 08 "12345678"
```

ceci nous renvoie le résultat suivant :

```
* a0 40 00 00 08 "12345678"
0.000 < a0 40 00 00 08 31 32 33 34 35 36 37 38          @...12345678
0.084 > 90 00
exécution normale
```

ce qui signifie que l'opération s'est bien déroulé. On a ensuite tenter de bloquer la carte , en entrant a chaque fois un faux code pin (on le fait quatre fois ) de la manière suivante :

```
# blocage code pin
a0 20 00 00 08 "11111111"
a0 20 00 00 08 "11111111"
a0 20 00 00 08 "11111111"
a0 20 00 00 08 "00000000"
```

nous obtenons le résultat suivant :

```
 * a0 20 00 00 08 "11111111"
0.084 < a0 20 00 00 08 31 31 31 31 31 31 31 31          ...11111111
0.108 > 98 42   Il nous reste 2 tentatives restantes                Berreur
* a0 20 00 00 08 "11111111"
0.108 < a0 20 00 00 08 31 31 31 31 31 31 31 31          ...11111111
0.132 > 98 41 Il nous reste plus qu'une seul                        Aerreur
* a0 20 00 00 08 "11111111"
```

0.132 < a0 20 00 00 08 31 31 31 31 31 31 31 31     ...11111111
0.156 > 98 40 Plus de tentative        @erreur
* a0 20 00 00 08 "00000000"
0.156 < a0 20 00 00 08 30 30 30 30 30 30 30 30     ...00000000
0.168 > 6d 00 erreur qui nous annonce que la carte est bloqué

Nous avons ensuite essayé de débloquer la carte via ce script :
#déblocage
a0 2c 00 00 10 "1234567811111111"

le résultat est le suivant:
* a0 2c 00 00 10 "1234567811111111"
0.168 < a0 2c 00 00 10 31 32 33 34 35 36 37 38 31 31 31   ,...12345678111
0.168 < 31 31 31 31 31        11111
0.236 > 90 00          .
exécution normale

Donc on a bien réussi a débloquer la carte et mettre un nouveau code pin (11111111)
Ici on tente de déverrouillé la carte :
#déverrouiller
a0 20 00 00 08 "11111111"
on a alors :
* a0 20 00 00 08 "11111111"
0.236 < a0 20 00 00 08 31 31 31 31 31 31 31 31     ...11111111
0.260 > 90 00          .
exécution normale

puis enfin on change de code pin et puis on essaye de déverrouiller avec le nouveau code pin via :
#changer pin
a0 24 00 00 10 "1111111122222222"
a0 20 00 00 08 "22222222"

et nous obtenons le résultat suivant :
* a0 24 00 00 10 "1111111122222222"
0.260 < a0 24 00 00 10 31 31 31 31 31 31 31 31 32 32 32   $...11111111222
0.260 < 32 32 32 32 32        22222
0.320 > 90 00          .
exécution normale
* a0 20 00 00 08 "22222222"
0.320 < a0 20 00 00 08 32 32 32 32 32 32 32 32     ...22222222
0.336 > 90 00          .
exécution normale

Tout semble marcher

Auteurs du projet : MOINEAU Mathieu et BODIAN Ibrahima

N.B : j'ai oublier de mentionner les noms dans le premier rapport , il s'agit bien d'un projet en communs de Mathieu et de moi-même