# UN SYSTÈME RFID/NFC POUR NOTRE GNU/LINUX

*par Denis Bodor*

La technologie NFC se popularise sur les smartphones et les tablettes. Très proche du RFID dont elle tire une partie de ses spécifications, la technologie NFC permet pour le grand public toute une collection d'applications rendant la fameuse « *user experience* » plus intéressante et agréable. Techniquement parlant, RFID et NFC sont deux domaines très riches qu'il est agréable d'explorer pour, ensuite, l'adapter à ses besoins spécifiques. Chose que nous allons découvrir sans plus attendre...

## 1 RFID ou NFC ?

On voit tantôt utiliser « RFID » en lieu et place de « NFC » et inversement dans quelques descriptions sommaires et un peu trop brouillonnes. Les deux technologies sont très proches et similaires sur bien des points, ce qui ne manque pas de créer une confusion chez les personnes les moins pointilleuses. Commençons par le début : RFID (*Radio Frequency IDentification*) est un ensemble de normes et standards définissant une technologie permettant de mémoriser des données sur un support et de les faire transiter à distance. Les *tags* ou étiquettes RFID peuvent prendre la forme d'autocollants, de cartes, de badges ou encore de capsules sous-cutanées. Le but de la technologie RFID est de permettre une identification des objets, des personnes ou des animaux. Sommairement, un tag ou marqueur RFID est composé d'une antenne associée à une puce électronique. L'antenne joue deux rôles dans le cas des tags passifs : assurer la communication sans fil et capter l'énergie fournie par le dispositif de lecture/écriture. Le tag passif ne dispose donc pas de source d'énergie propre, contrairement aux tags actifs ou semi-actifs (dit BAP pour *Battery-Assisted Passive*).

Le RFID utilise plusieurs bandes de fréquences en fonction de l'application et de la précision souhaitée :

- LF (*Low Frequency*) : 125 ou 134 Khz,
- HF (*High Frequency*) : 13,56 Mhz,
- UHF (*Ultra High Frequency*) : 865 ou 915 Mhz en fonction de la réglementation locale (865 MHz à 868 MHz dans la CE).

NFC pour *Near Field Communication* est une norme définissant le même type de fonctionnalités et de technologies. Il ne s'agit pas de la même chose, mais certaines des spécifications sont communes.

*Petit échantillon de différentes formes que peuvent prendre les tags RFID : stickers, porte-clé, carte...*

Dans le cas du NFC, il s'agit par exemple uniquement d'utilisations de tags passifs HF. LF et UHF ne sont pas couvert par la norme NFC et tous les lecteurs NFC fonctionnent « sur » 13,56 Mhz. D'autre part, certaines fonctionnalités sont ajoutées par la norme NFC mais ne sont pas présentes dans RFID. La communication *peer-to-peer* et l'émulation de tags n'existent pas dans RFID mais permettent respectivement le partage de contenus via Android Beam entre deux smartphones compatibles NFC et le futur paiement par smartphone NFC.

RFID et NFC ne sont donc pas «la même chose», même si certains tags, comme les Mifare Classic par exemple, peuvent être lus et écrits avec un périphérique NFC car ils répondent, à peu près, aux normes définies par l'un et l'autre ensemble de normes.

Un autre élément caractérisant le NFC est le format des données stockées dans la mémoire du tag. En considérant les tags RFID Mifare Classic 1k (S50) ou 4k (S70) par exemple, l'espace de stockage est organisé, comme nous le verrons par la suite, en secteurs de 4 ou 16 blocs de 16 octets. Certains blocs dans chaque secteur (ou dans certains secteurs) ont une utilité particulière dans la norme. Ainsi, par exemple, le dernier bloc (3 ou 15) de chaque secteur est le *sector trailer* qui contient des informations à propos de l'accès au secteur. En plus (ou au-dessus) de ces spécifications, s'ajoute un format de messages spécifique au NFC : le NDEF pour *NFC Data Exchange Format*. Il s'agit d'un format binaire de messages permettant d'encapsuler des données pour une application. On peut faire le parallèle avec les technologies XML par exemple, permettant de structurer une information.

La différenciation entre RFID et NFC tient donc également dans le format des données stockées sur un tag. Techniquement parlant, un tag Mifare Classic 4k vierge n'est pas un tag NFC. S'il est initialisé ou « formaté » de manière spécifique en utilisant des données NDEF, il devient un tag NFC, même


*L'ACR122U, sans doute le lecteur RFID/NFC USB le plus économique et le plus populaire... mais certainement pas le plus fiable.*

vierge. Plus précisément, comme l'indique la note d'application de NXP sur les opérations NFC avec les tags Mifare Classic, un secteur **PEUT** contenir des données NDEF, il est alors appelé un secteur NFC. Ne vous étonnez donc pas de pouvoir lire des tags NFC avec un lecteur Mifare et des tag Mifare avec un lecteur NFC... Mais ceci à ses limites et les choses sont un peu plus compliquées qu'il n'y paraît.

La problématique de la compatibilité NFC/Mifare Classic s'est popularisée récemment avec l'apparition de nouveaux modèles de smartphone. Ceux-ci utilisant des puces autres que ceux de chez NXP (la PN544 en particulier) comme la Broadcom BCM20793 ne peuvent physiquement plus lire les tags Mifare Classic mais pourtant sont compatibles NFC. Ceci est difficile à expliquer à moins de passer par une lecture complète des spécifications, mais nous pouvons le résumer ainsi : un tag NFC tel que décrit dans les spécifications du NFC Forum est un tag répondant aux standards ISO/IEC 14443A ou ISO/IEC 14443B. A et B sont deux déclinaisons du standard mais le problème n'est pas là. ISO/IEC 14443 est un ensemble découpé en quatre parties :

- ISO/IEC 14443-1 : Caractéristiques physiques,
- ISO/IEC 14443-2 : Interface d'alimentation et de gestion de signaux par fréquence radio,
- ISO/IEC 14443-3 : Initialisation et anti-collision,
- ISO/IEC 14443-4 : protocole de transmission.

Un tag Mifare Classic utilise en partie des technologies propriétaires de NXP. Il respecte ainsi une partie des spécifications mais pas ISO/IEC 14443-4. De ce fait, comme le protocole de transmission est différent, seuls les contrôleurs NXP sont en mesure de lire et d'écrire les tags Mifare Classic mais non les contrôleurs Broadcom. Pour ajouter à la confusion, les spécifications du NFC Forum précisent également des types (1-4) :

- Type 1 : présence d'un UID, verrouillable en lecture seule,

- Type 2 : UID, verrouillable, anti-collision,
- Type 3 : pas d'UID, verrouillable, anti-collision,
- Type 4 : UID, verrouillable, anti-collision, contenu actif possible (le contenu du tag peut être modifié par le périphérique de lecture/écriture mais également par le tag lui-même).

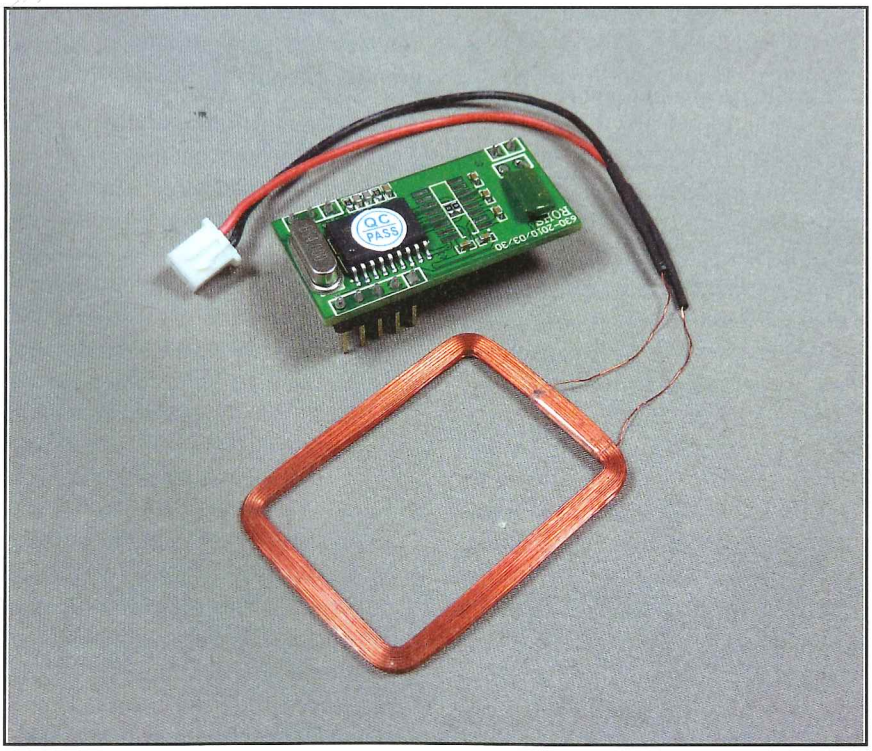
Et le plus perturbant est le fait suivant : les très courants Mifare Classic de NXP ne sont donc pas pleinement compatibles avec les spécifications du NFC Forum mais d'autres tags NXP le sont :

- NXP Mifare Ultralight & Ultralight C : NFC Forum Type 2,
- NXP Mifare DESFire EV1 : NFC Forum Type 4 Tag v2.0.

Et pour finir de semer le trouble, dans certaines documentations NXP (http://www.nxp.com/documents/other/R_10014.pdf), on parle de Mifare Classic comme d'un « *NFC Type MIFARE Classic Tag* », comme s'il s'agissait finalement d'un type nouveau mais uniquement compatible NXP...

Ce qu'il faut retenir de tout cela est finalement simple : Mifare Classic est presque un tag NFC, il peut contenir des données NDEF et s'utilisera comme un tag NFC avec des périphériques utilisant une puce NXP. Un vrai tag NFC lui, fonctionnera de la même manière, mais comme il respect l'intégralité des spécifications, il fonctionnera avec tous les périphériques, même ceux d'NXP. Pour information, les smartphones suivants sont incapables de lire/écrire un tag Mifare Classic car utilisant une puce Broadcomm :

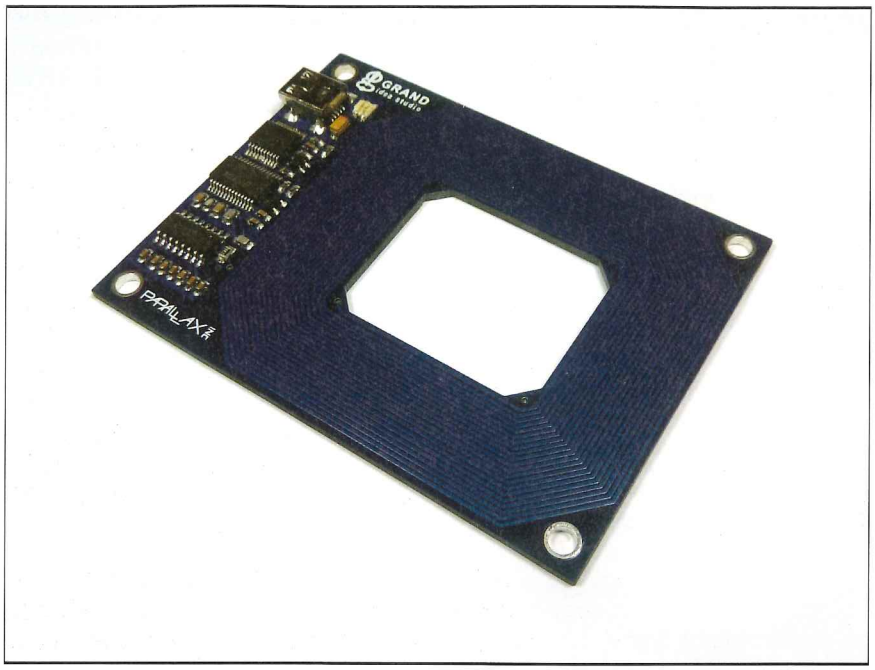
- Google Nexus 4 ;
- Google Nexus 5 ;
- Google Nexus 7 (2013) ;
- Google Nexus 10 ;
- Samsung Galaxy S4 ;
- Samsung Galaxy Ace 3 ;
- Samsung Galaxy Express 2 ;
- Samsung Galaxy Mega ;
- Samsung Galaxy Note 3 ;
- LG G2 ;
- LG Optimus L7 II.




*Exemple de module permettant la lecture de tags RFID 125 Khz tel qu'on en trouve un peu partout.*

Faisons un petit détour par le RFID 125 Khz & 134,2 KHz tout en rappelant que cette technologie n'est pas dans les normes NFC. Ces tags (EM4x05, EM4x50, EM4x02) sont généralement utilisés pour la détection de présence et la simple identification (non l'authentification). Chaque tag est identifié par un ID unique et ne peut être reprogrammé. Un certain nombre de systèmes de contrôle d'accès (ouverture de porte) utilise ce type de tags dont la sécurité est très limitée, sinon inexistante. En effet, plusieurs développeurs ont fait la démonstration qu'il était relativement simple d'émuler un tag à l'aide d'un microcontrôleur comme un Atmel AVR Atmega8 ou un ATtiny85 par exemple. On notera que dans la gamme de fréquences des 134 Khz (standard ISO 11784 & 11785) nous trouvons d'autres types de tags RFID destinés à l'identification des animaux. J-M Friedt, dans un excellent article paru dans une autre de nos publication (GLMF HS 39, « Analyse des étiquettes d'identification par radiofréquence »), a présenté la façon de lire les informations contenues dans ce type de tags, implantés dans deux lévriers italiens (Adélie et Cézane), avec un Atmel U2270B, le logiciel Audacity et GNU/Octave. Ce type de tags RFID est encore largement utilisé dans des domaines où des standards plus sûrs devraient être utilisés (restauration rapide par exemple).

En parlant de sécurité, remarquons que la plupart des systèmes ont d'ores et déjà montrés leurs limites tant au niveau technique qu'au niveau des protocoles de gestion utilisés. La sécurité des Mifare Classic par exemple, a fait l'objet de travaux de recherche en sécurité, d'analyses, de *reverse*, de conférences... qui ont conduit à la fois à la divulgation de certains problèmes de sécurité et à la diffusion d'outils de tests mettant ceux-ci en évidence. MFOC pour *Mifare*




*Le module Parallax RFID USB intègre antenne et circuits logiques en un seul PCB. Il se présente pour un PC comme un port série envoyant des données en 2400 8N1*

*Classic Offline Cracker* et MFCUK pour *MiFare Classic Universal toolKit* utilisent des faiblesse connues différentes, largement documentées et la facilité d'utilisation de ces *proof on concept* est édifiante. MFOC, par exemple, est en mesure de découvrir l'ensemble des clés si une seule d'entre elle est connue, et ce en un temps record.

Mais plus grave encore, c'est l'utilisation qui est faite de ces technologies qui reste le maillon faible (comme toujours me direz-vous) car la plupart du temps, les mécanismes de sécurité les plus simples ne sont même pas utilisés. Pire encore, parfois c'est le choix du type de carte lui-même qui ne prend en compte absolument aucune disposition dans ce sens. C'est ainsi, avec étonnement, que nous avons pu découvrir que des cartes Mifare Ultralight étaient utilisées dans certains hôtels, offrant donc la possibilité à une personne malveillante de cloner une carte existante et d'accéder à une chambre (ou pire encore, de comprendre la structure des données contenues dans la carte et d'accéder à n'importe quelle chambre).

## 2 Quel matériel pour lire et écrire des tags RFID/NFC ?

Il existe un grand nombre de solutions permettant de lire et d'écrire les tags RFID. En fonction du type de tag visé et des fonctionnalités recherchées, le budget peut aller de quelques euros à plusieurs centaines. Un module 125 Khz constitué d'un circuit comprenant un simple microcontrôleur et d'une bobine se trouve facilement (chez SeeedStudio par exemple) pour quelques 10 euros, mais vous pouvez également opter pour la solution générique comme la carte Proxmark3 à 400 euros, capable de lire, écrire, cloner, analyser un nombre incroyable de tags (125 Khz, 134 Khz, 13,56 Mhz) et permettant d'explorer le domaine dans son ensemble voire de développer vos propres solutions. Entre ces deux extrêmes se place toute une collection de modules, circuits, cartes et périphériques, parmi lesquels, par exemple :

- Le NFC/RFID breakout Module d'ElecFreacks construit autour du PN532 de NXP,
- Le très courant et économique ACS ACR122U interfacé en USB,
- Le SCM Microsystems SCL3711 également en USB, plus petit, mais surtout plus « stable » que l'ACR122U,
- Le Parallax RFID Card Reader USB ou série destinée à lire des tags de la famille EM4100,
- Le module YHY502CG de chez Ehuoyan offrant une liaison série pour la lecture/écriture de Mifare Classic accompagné d'une antenne PCB,
- Le shield Arduino RFID/NFC PN532 d'AdaFruit,
- Un smartphone supportant NFC et n'étant pas un de ceux spécifiés précédemment (du moins si vous souhaitez utiliser des tags Mifare Classic).

L'ensemble des périphériques, qu'on qualifiera de « lecteurs » même si la plupart peuvent également écrire sur/dans les tags, peut être réparti en trois grandes catégories :

- Les lecteurs intégrés aux périphériques mobiles comme les smartphones. On dénombre actuellement plus d'une centaine de modèles de smartphones (Android ou non) supportant NFC et presque tout autant sont annoncés par les constructeurs. Cette fonctionnalité, initialement intégrée aux périphériques haut de gamme, apparaît maintenant dans des appareils plus accessibles (voir d'entrée de gamme comme le Samsung Galaxy Ace 2). A noter également, le périphérique Nexus Q de Google, une station multimédia connectée qui, 6 mois après son apparition, fut abandonnée faute de popularité. Le support NFC de ce périphérique lui permet de réagir à la présence de tags mais aussi et surtout de communiquer avec un smartphone Android afin de


*La « clé USB » SCM SCL3711, similaire à l'ACR122U, à peine plus cher mais beaucoup plus fiable*

« beamer» de la musique, des URL, etc. Le «beam» peut également se faire entre deux smartphones Android disposant de cette fonctionnalité et revient à faire communiquer deux smartphones de manière un peu obscène en les faisant se frotter... Aujourd'hui, le futur du NFC sur smartphone semble être plus en lien avec les systèmes de paiement (*Google Wallet*) qu'avec les échanges de données entre périphériques.

- Les lecteurs ou clés USB. Ceux-ci sont généralement pris en charge par un *middleware* (PC/SC sous GNU/Linux par exemple) ou par un accès direct au matériel via une bibliothèque (libnfc). PC/SC pour *Personal Computer/Smart Card* est un ensemble logiciel permettant d'unifier et simplifier l'accès aux smartcards. L'implémentation en logiciel libre pour GNU/Linux permet par exemple de prendre en charge les lecteurs de smartcards au sens large du terme (Javacard, SIM, CB, etc) mais également certains lecteurs RFID.

- Les modules électroniques. Interfacés en RS232 (TTL ou non), SPI, USB ou encore i2c, ces modules se composent généralement d'une partie construite autour d'un microcontrôleur associé à un contrôleur RFID et une antenne. De nombreux modèles existent et chacun utilise un protocole de communication propriétaire détaillé dans la documentation qui l'accompagne. On trouve ces modules sous la forme de kits vendus sur de nombreuses boutiques en ligne à destination des amateurs éclairés et autres bidouilleurs. Ce type de solutions, en dehors de l'aspect économique, présente de nombreux avantages. En effet, ils permettent d'approcher au plus près la structure interne des tags, permettent une intégration dans un projet en cours (domotique par exemple) et surtout offrent une polyvalence inégalée. On pourra ainsi expérimenter via un PC et un code Python pour ensuite implémenter sa solution sur un microcontrôleur en C par exemple, chose bien plus délicate/lourde avec un lecteur USB (mais pas impossible).

*Les cartes RFID peuvent être des supports combinant plusieurs technologies. Ici des cartes DESFire avec une « puce JCOP » et une piste magnétique.*

En fonction du matériel utilisé, on dispose alors d'une interface de plus ou moins haut niveau. Les smartphones Android intègrent cela au cœur de l'API Android et bon nombre d'applications existent, permettant de manipuler des tags NFC ou Mifare. Les périphériques USB s'utilisent généralement au travers d'un middleware ou de bibliothèques disponibles pour plusieurs langages (C, C++, Python, etc). Là encore, un certain nombre d'outils existent, permettant de faciliter les manipulations. Enfin, les modules, eux, sont complètement dépendant de leur conception, pouvant tout aussi bien fournir une interface complète ou un jeu de commandes basiques, simple traduction à la volée des commandes prisent en charge par le contrôleur (APDU).

Au fil des articles qui suivent, nous verrons différentes manières de mettre en œuvre les technologies RFID et dans une grande partie celles liée aux tags Mifare Classic et au NFC. Il est impossible d'être réellement exhaustif dans ce domaine tant les solutions techniques sont nombreuses et nous nous concentrerons donc sur les plus utilisées. ■

# UTILISATION ET MANIPULATION DES TAGS MIFARE CLASSIC

*par Denis Bodor*

La technologie de « carte à puce » sans contact Mifare n'est pas récente, mais c'est très certainement la plus répandue dans le monde avec des milliards de tags servant dans toutes sortes de domaines, du contrôle d'accès au porte-monnaie électronique en passant par les cartes de membre et de fidélité. Un membre de cette famille de solutions Mifare est majoritairement utilisé : Mifare Classic, bien qu'il ne soit plus recommandé de l'utiliser pour des nouveaux projets.

La Mifare Classic porte bien son nom car si vous tombez sur une carte parfaitement inconnue que l'on vous confie à l'occasion d'un événement, d'une inscription à un programme de fidélité ou même permettant de stocker une valeur financière comme pour une laverie automatique ou une cafétéria, il y a de fortes chances qu'il s'agisse d'une Mifare Classic. Voici une bonne raison de nous pencher sur cette technologie et d'en comprendre les fondements.

## 1 La famille Mifare

La technologie Mifare ou MIFARE pour *MIkron FARE-collection System* a initialement été développée par la société Mikron (1994) puis, en 1998, a été acquise par Philips maintenant devenue NXP. Cette technologie a fait l'objet de cessions de licences à plusieurs constructeurs dont Infineon, Hitachi/Renesas, Gemalto ou encore Oberthur.

Le terme Mifare est aujourd'hui un simple nom commercial puisqu'il désigne toute une famille de cartes RFID, tantôt radicalement différentes.

Nous avons d'une part les Mifare Classic qui sont principalement, vues de l'extérieur, des cartes mémoires disposant d'un espace de stockage et de différents mécanismes de contrôle d'accès aux données. On distingue, dans cette sous-famille, les Mifare Classic 1k (S50) et 4k (S70) auxquelles s'ajoutent les Mifare Classic Mini (320 octets) et les Mifare Ultralight de 64 octets. Ces dernières utilisent une organisation mémoire assez similaire, tout en utilisant certaines spécificités. L'idée est de disposer de puces à bas coût permettant une utilisation « jetable », comme pour des billets de spectacles ou de manifestations, des titres de transport ou encore pour la consommation de crédits non renouvelables. Notez qu'une déclinaison plus sécurisée existe, la Mifare Ultralight C, intégrant un support triple DES destiné à limiter le clonage des cartes.

A cette gamme classique s'ajoute une toute autre technologie constituée de tags à microprocesseur ou plutôt à microcontrôleur. Mifare ProX et Mifare SmartMX n'ont pas grand-chose à voir avec les Mifare Classic et doivent être envisagés comme des systèmes autonomes (exactement comme une carte SIM ou une carte bancaire). Elles sont similaires dans leurs usages aux Javacards, permettant de faire fonctionner un système d'exploitation JCOP (*Java Card OpenPlatform*). La carte Française Sesam-Vitale utilise, par exemple, la technologie NXP SmartMX.

Une version spéciale de SmartMX est à la base d'une autre carte Mifare très utilisée : la Mifare DESFire. Celle-ci est composée d'un cœur SmartMX faisant fonctionner le système d'exploitation DESFire et se décline en une version 4k et 8k. Le modèle le plus courant, DESFire EV1, introduit en 2008, remplace le vieillissant modèle DESFire initial (MF3ICD40) arrêté fin 2011. Une Mifare DESFire EV2 a été annoncée fin

2013, intégrant un certain nombre de nouvelles fonctionnalités. La DESFire EV1 est, semble-t-il, considérée comme la carte la plus sûre pour les nouvelles implémentations.

Notez également qu'un remplacement pour la Mifare Classic existe, la Mifare Plus, destinée à fournir une certaine compatibilité tout en permettant la migration vers une technologie plus sûre. Il est cependant peu probable que le parc entier de Mifare Classic ne soit rapidement remplacé par des Mifare Plus car, en dehors du coût brut du remplacement des cartes, la migration impose un changement de lecteurs ou du moins leurs mises à jour. Ainsi, bien qu'il existe des failles de sécurité importantes avec les actuelles Mifare Classic S50 et S70, il y a fort à parier qu'on les retrouvera toujours un peu partout un sacré petit bout de temps...

## 2 Organisation mémoire

Une carte Mifare Classic S50 (1Ko) ou S70 (4Ko) se comporte comme une carte mémoire. L'espace de stockage est organisé en secteurs, eux-mêmes divisés en blocs de 16 octets. Pour une Mifare S50, nous avons 16 secteurs de 4 blocs (16*4*16=1024), pour la S70 nous avons 32 secteurs de 4 blocs, plus 8 de 16 blocs (32*4*16+8*16*16=4096). Les secteurs sont numérotés de 0 à 15 (ou 0 à 40 pour les S70). Les blocs dans les secteurs sont parfois numérotés de 0 à 3 (ou 0 à 15 dans le cas des 8 derniers secteurs des S70) mais le plus souvent on utilise le numéro de bloc par rapport à l'ensemble de la mémoire, le premier bloc du second secteur étant alors le bloc 4, etc.

Dans chaque secteur, le dernier bloc est appelé *sector trailer* et contient des informations concernant la gestion d'accès aux données des trois autres blocs du secteur qui sont appelés blocs de données (*data blocks*). De plus, le premier bloc du premier secteur d'un tag est le *manufacturer block*. Ce bloc contient des informations sur le tag, son modèle et son constructeur et est en lecture seule. On parle généralement du « bloc 0 » pour le désigner dans les documentations, sans préciser le secteur qui est implicitement le premier. C'est ici qu'est inscrit, par exemple, le numéro de série sur 32 bits de chaque tag. En fonction du modèle exact de tag, les sept premiers octets du bloc 0 n'ont pas la même signification.

Ce point de détail est rarement spécifié dans les documentations et tutoriels génériques que l'on trouve sur le net, mais un coup d'œil aux *datasheets* NXP (MF1S50YYX.pdf) apporte une vision explicite. Une puce MF1S503yX, par exemple, voit les octets 0-6 constituer un UID (identifiant unique) alors que la puce MF1S503yX, elle, utilise les octets 0-3 comme NUID (identifiant non unique), le reste des données du bloc constituant les données fabricant. Dans le cas d'un NUID de 4 octets, l'octet 4 est un LRC (*Longitudinal Redundancy Check*) des 4 octets du NUID. De manière générale, et ce indépendamment du modèle exact de puce S50, une seule chose est à retenir : NE BASEZ JAMAIS LA SÉCURITÉ DE VOTRE SYSTÈME SUR LE CONTENU DU BLOC 0 ! En effet, même si la quasi-totalité des cartes Mifare Classic, NXP ou non, ont un bloc 0 en lecture seule, ceci n'est pas une vérité universelle. Un marché parallèle existe, proposant des cartes chinoise compatibles, offrant des fonctions de lecture/écriture sur l'ensemble des données. On appelle cela des *Mifare with changeable UID* et ces cartes ne sont pas vendues sous le manteau mais librement disponibles dans des boutiques en ligne ou sur eBay. Il est donc parfaitement possible de cloner intégralement une Mifare Classic, bloc 0 inclus, sans avoir recours à un émulateur de carte ou à un quelconque autre montage électronique complexe et coûteux, il suffit d'acheter une carte adaptée.

En dehors des spécificités du bloc 0, tous les secteurs sont organisés de la même manière avec 3 blocs de 16 octets pour les données et le *sector trailer* comprenant une clé A (0-5), un ensemble de bits décrivant les conditions d'accès aux blocs du secteur (6-9) et une clé B (10-15). Les blocs peuvent contenir des données brutes ou être considérés comme des *value blocks*. Le fait qu'un bloc soit un *value block* est déterminé par les conditions d'accès spécifiées pour ce bloc dans le *sector trailer*. Dès lors qu'un bloc est dans ce « mode », les données qu'il contient sont/doivent être formatées/organisées spécifiquement :

- octets 0-3 : valeur signée sur 4 octets (32 bits),
- octets 4-7 : la même valeur inversée,
- octets 8-11 : copie de la valeur,
- octet 12 : octet pouvant servir à spécifier un adresse (de bloc) pour la destination,
- octet 13 : le même octet inversé,
- octet 14 : copie de l'octet,
- octet 15 : copie de l'octet inversé.

Comme vous pouvez le constater, les redondances utilisées permettent la détection d'erreurs. Les opérations d'incrémentation et de décrémentation de la valeur ne peuvent être effectuées que si les données du bloc sont cohérentes. Petite précision sur les quatre derniers octets, désignés comme « adr » ou « adresse » dans la documentation NXP. Ne cherchez pas de relation spécifique avec un mode d'adressage quelconque, il ne s'agit que d'un espace de stockage pour une valeur non signée sur un octet. La notion d'adresse utilisée est en relation avec le mode de fonctionnement des opérations d'incrémentation et de décrémentation. En effet, si le bloc est un *value block*, du fait de la configuration des bonnes conditions d'accès, une incrémentation/décrémentation n'est pas une opération atomique au regard du stockage des données. L'opération a pour effet de prendre la valeur du bloc et d'en faire une copie incrémentée/décrémentée dans un registre interne. A ce stade, il n'y a pas encore d'écriture. Ce n'est qu'ensuite que l'on peut enregistrer la nouvelle valeur avec une opération de transfère (copie de la

valeur du registre vers un *value block*). Mais ce transfère peut être tout aussi bien fait vers le bloc d'origine que vers un autre *value block*. Ceci est désigné comme un *backup management* dans la documentation. Et c'est précisément là que l'octet d'adresse peut être utile afin de stocker le numéro du bloc de destination de la copie registre/bloc. Ceci n'est pas une procédure interne de la carte mais, pourrait-on dire, la mise à disposition d'un espace de stockage permettant de faciliter la gestion de sauvegarde de *value blocks*.

## 3 Mécanisme de sécurité

Le mécanisme de sécurité intégré aux Mifare Classic utilise un système d'authentification par clé. Il existe, pour chaque secteur, deux clés qui peuvent être utilisées : A et B. Le *sector trailer*, le quatrième bloc de chaque secteur pour une Mifare Classic 1k, contient ces deux clés ainsi que trois octets qui définissent leurs utilisations, ce sont les *access bits*. Ceux-ci ont une signification différente lorsqu'ils concernent un bloc de données ou le *sector trailer* lui-même.

C'est un peu touffu mais nous allons essayer de rendre cela plus intelligible que les datasheets NXP, en particulier en présentant cela sous un angle adapté pour un développeur (vous savez, avec le premier bit qui est le bit 0 et non le bit 1, le MSB à gauche, etc). Avant toute chose, il faut savoir que l'accès aux données se fera après authentification et ce, par bloc. Le lecteur ou PCD (pour *Proximity Coupling Device*), piloté par un programme, s'authentifie avec la clé A ou la clé B pour un bloc donné. Après cette étape, les conditions d'accès déterminées par les *access bits* définissent ce qu'il est possible de faire : lire, écrire, ou incrémenter/décrémenter une valeur sur 32 bits sur le bloc qui contient des données adaptées pour être considéré comme un *value block*.

Pour chaque bloc de données, trois bits déterminent ces conditions d'accès aux données (la notation utilisée correspond ici à celle utilisée, par exemple, par la libfreefare, avec C1 à droite et C3 à gauche, par opposition à la notation dans les datasheets qui est, selon moi, très perturbante et source d'erreurs graves) :

- **000** : lecture, écriture, incrémentation et décrémentation après authentification avec les clés A ou B. Ceci est appelé « configuration de transport » et il s'agit de la configuration par défaut des cartes sortant d'usine.
- **001** : lecture grâce aux clés A ou B, écriture par clé B, pas d'incrémentation, pas de décrémentation.
- **010** : lecture avec A ou B et c'est tout, pas d'écriture, pas d'incrémentation/décrémentation.
- **011** : lecture avec A ou B, écriture et incrémentation avec B, décrémentation avec A ou B. Ceci est typiquement une configuration pour un *value block* avec des crédits à consommer via la clé A ou B, et rechargeable via la clé B.
- **100** : lecture par clé A ou B, pas d'écriture ou d'incrémentation et décrémentation avec A ou B. Là encore, il s'agit d'une configuration pour une consommation de crédits ou de points mais sans possibilité de rechargement.
- **101** : lecture via clé B et rien d'autre. Ce type de configuration rend un bloc secret sauf si on dispose de la la clé B.
- **110** : lecture et écriture via clé B.
- **111** : Absolument rien, quelque soit la clé utilisée. Pas de lecture, ni d'écriture ou d'incrémentation/décrémentation. J'avoue ne pas bien comprendre l'utilité ce cette configuration sauf pour « désactiver » des blocs (sauf à repasser pas la configuration des *access bits*).

Notez que, s'il est possible de lire la clé B dans le *sector trailer* correspondant au bloc auquel nous voulons accéder, celle-ci ne pourra pas être utilisée malgré une configuration adéquate des *access bits*. En d'autres termes, si on peut lire la clé B grâce à la clé A alors, la clé B ne peut être utilisée. Ceci cependant ne semble être valable que pour les véritables Mifare Classic de chez NXP. Nos essais ont montré que la clé B, même si elle est lisible dans le *sector trailer*, PEUT être utilisée avec des tags compatibles. En particulier les puces FM11RF08 de Fudan Microelectronics.

Les *access bits* peuvent également être configurés pour le quatrième bloc d'un secteur, c'est à dire le *sector trailer*. Dans ce cas, leur signification est alors différente car ceci concerne les conditions de lecture et d'écriture des clés elles-mêmes ainsi que des *access bits*. Quelque soit la configuration, la clé A n'est jamais lisible :

- **000** : clé A inscriptible avec la clé A, bits lisibles par A, clé B lisible et inscriptible par A.
- **001** : A inscriptible par B, bits lisibles par A ou B, B inscriptible par B.
- **010** : bits par A, clé B lisible par A.
- **011** : bis lisibles par A ou B.

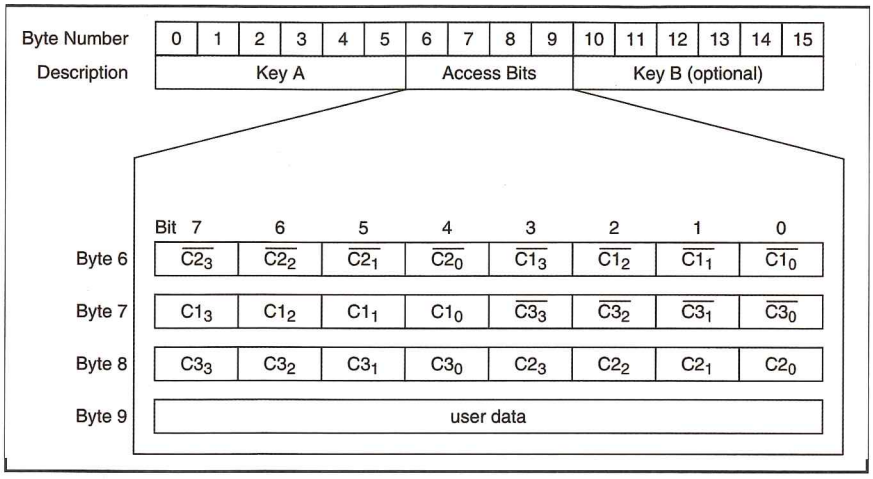
| C1XY | C2XY | C3XY | Read | Write | Increment | decr, transfer, restore |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | KEYA\|B | KEYA\|B | KEYA\|B | KEYA\|B |
| 0 | 1 | 0 | KEYA\|B | Never | Never | Never |
| 1 | 0 | 0 | KEYA\|B | KEYB | Never | Never |
| 1 | 1 | 0 | KEYA\|B | KEYB | KEYB | KEYA\|B |
| 0 | 0 | 1 | KEYA\|B | Never | Never | KEYA\|B |
| 0 | 1 | 1 | KEYB | KEYB | Never | Never |
| 1 | 0 | 1 | KEYB | Never | Never | Never |
| 1 | 1 | 1 | Never | Never | Never | Never |

*Tableau récapitulatif de la signification des access bits pour les blocs de données*

- **100** : tout est accessible via la clé A (sauf la lecture de la clé A). Ceci est la configuration de transport.
- **101** : bits lisibles par A ou B et inscriptibles par B.
- **110** : A inscriptible par B, bits lisibles par A ou B et inscriptibles par B, B inscriptible par B.
- **111** : seuls les *access bits* sont accessibles en lecture uniquement par A ou B.

Vous remarquerez, en dehors du nombre hallucinant de combinaisons possibles, qu'on peut ainsi, en configurant judicieusement les clés est les conditions d'accès aux données et au *sector trailer*, structurer n'importe quelle politique de sécurité. Autre point important, il est parfaitement possible de verrouiller un tag, le rendant complètement inutilisable et impossible à réinitialiser. Il suffit pour cela, par exemple, d'opter pour des *access bits*/ **111** pour le *sector trailer* et la même chose pour les blocs de données. A ce moment, avec ou sans clé, la configuration ne peut plus être changée et les données ne sont plus lisibles. Techniquement parlant, votre tag peut être simplement mis à la poubelle (ou plongé dans l'acétone pour accéder à l'antenne et à la puce).

Vous trouverez en illustration dans cet article, les tableaux précisant l'utilisation des *access bits* tels que présentés dans la documentation de Fudan pour la puce FM11RF08 des tags compatibles Mifare Classic. Vous noterez l'organisation peu pratique des colonnes des trois bits avec C1 à gauche et C3 à droite, ce qui laisse entendre, si on n'est pas très attentif, que **001** est la configuration de transport pour le *sector trailer* alors que, dans votre code, ce sera **100**.




*Organisation des access bits présents dans le dernier bloc de chaque secteur (sector trailer)*

Et pour ajouter à la confusion, les *access bits*, tels que physiquement présents dans le *sector trailer* sont, eux aussi, organisés de manière un peu particulière. En effet, ceux-ci sont enregistrés ainsi (la syntaxe *Cx_y* désigne le bit *x-1* pour le bloc *y* du secteur, la barre de fraction indique une inversion de bit) :

- octet 6 : /C2_3, /C2_2, /C2_1, /C2_0 puis /C1_3, /C1_2, /C1_1, /C1_0
- octet 7 : C1_3, C1_2, C1_1, C1_0 puis /C3_3, /C3_2, /C3_1, /C3_0
- octet 8 : C3_3, C3_2, C3_1, C3_0 puis C2_3, C2_2, C2_1, C2_0

Les séries de bits sont dupliquées et inversées pour assurer un contrôle d'intégrité mais on remarque que l'ordre dans lequel apparaissent les bits a été choisi dans le sens « le *x*-ième bit pour chaque bloc » et non « tous les bits pour un bloc *y* ».

Cette diversité de sources potentielles d'erreurs, à la fois dans la documentation et dans les choix techniques, ne sont très certainement pas sans rapport avec le laxisme de certains développeurs et sociétés utilisants les tags Mifare Classic pour leurs projets. À la lecture des *datasheets*, le premier réflexe est presque de se dire que, pour éviter de faire une bêtise, le plus simple est encore de conserver la configuration de transport nous laissant une porte de sortie pour corriger un éventuel problème. Imaginez un parc de machines ou d'automates et une population de quelques milliers d'utilisateurs et donc de tags, puis imaginez les conséquences d'un simple petit bug dans le code pouvant conduire à rendre tous ces tags inutilisables. En collectant des tags usagés, perdus ou abandonnés, on se rend rapidement compte que les mécanismes de sécurité ne sont pas systématiquement mis en place, y compris pour des applications monétaires (laverie, points de fidélité convertibles en remise, etc) ou des contrôles d'accès ou d'identité (carte de membre). C'est un syndrome courant et l'on pourra faire le pendant avec ce qui se passe depuis quelques mois vis-à-vis de la surveillance globale. Une solution existe, gratuite et open source,

| | | | KEYA | KEYA | Access Con | Access Con | KEYB | KEYB |
|---|---|---|---|---|---|---|---|---|
| C1X3 | C2X3 | C3X3 | read | Write | Read | Write | read | Write |
| 0 | 0 | 0 | never | KEYA\|B | KEYA\|B | Never | KEYA\|B | KEYA\|B |
| 0 | 1 | 0 | never | Never | KEYA\|B | Never | KEYA\|B | Never |
| 1 | 0 | 0 | never | KEYB | KEYA\|B | Never | never | KEYB |
| 1 | 1 | 0 | never | Never | KEYA\|B | Never | never | Never |
| 0 | 0 | 1 | Never | KEYA\|B | KEYA\|B | KEYA\|B | KEYA\|B | KEYA\|B |
| 0 | 1 | 1 | Never | KEYB | KEYA\|B | KEYB | never | KEYB |
| 1 | 0 | 1 | Never | Never | KEYA\|B | KEYB | never | Never |
| 1 | 1 | 1 | Never | Never | KEYA\|B | Never | never | Never |

*Récapitulatif des access bits pour les trailer blocks*

c'est PGP/GnuPG, mais elle est clairement trop complexe et difficile à appréhender pour la plupart des utilisateurs qui préfèrent donc jeter l'éponge et qui, finalement, continuent à envoyer des messages et des documents non chiffrés. Il en va de même ici, à la différence que, si un développeur décide de se passer de la sécurité la plus basique dans son utilisation des tags Mifare Classic, l'utilisateur final et le client, eux, n'en sauront rien et ne pourront évaluer les risques qu'ils prennent. L'acquisition d'un lecteur RFID/NFC bas de gamme, dans le seul but d'en savoir un peu plus, justifie la dépense et la plupart du temps, un smartphone fera l'affaire.

Ajoutez à cela que, comme la sécurité des Mifare Classic a déjà fait l'objet de démonstrations concernant ses limites, certains pourraient se dire que si un pirate veut vraiment accéder aux données il le peut, alors à quoi bon. Bien entendu, ce n'est pas là une raison justifiant un laxisme. Si le cahier des charges impose l'utilisation de Mifare Classic, le développeur ou la SSII doit mettre en œuvre tout ce qui peut l'être pour assurer un minimum de sécurité, tout en insistant sur l'intérêt d'opter pour des tags plus sûrs (comme les DESFire EV1). Ce n'est pas parce qu'un voleur peut tout simplement casser une vitre pour entrer qu'il faut laisser les portes et les fenêtres ouvertes en permanence...

# 4 Un peu de pratique, que diable !

Après cette longue et pénible introduction très théorique, le mieux est encore de mettre la main sur un lecteur et une poignée de tags pour étudier tout cela sur le terrain. Le périphérique que nous allons utiliser ici est un lecteur, appelé dans les documentations un PCD (pour *Proximity Coupling Device*), de marque ACS et plus précisément l'ACR122 (ou ACR122U). Ce type de matériel se trouve un peu partout et surtout sur les sites d'enchères en ligne pour moins de 20 euros. Il ne fait pas parti des périphériques les plus stables et de meilleur qualité (plusieurs commentaires sur les forums de développeur conseil de l'éviter) mais il a l'avantage d'être tout simplement le moins cher et le plus facile à trouver. Il est souvent vendu sous l'appellation ACR122 (ou Tikitag/Touchatag) mais également simplement sous la désignation générique « *NFC reader* » mais on le reconnaîtra facilement à son aspect typique : boîtier blanc et énorme logo NFC entouré d'un épais cercle segmenté gris. Un meilleur choix est le SCM SCL3711 se présentant comme une simple clé USB noire mais aux performances bien supérieures (entre 30€ et 50€). L'ACR122 peut parfois cesser brusquement de fonctionner ou répondre sans raison, chose qui n'est jamais arrivée avec le SCL3711 durant nos essais.

Le support du matériel qui nous intéresse ici est, bien entendu, celui de GNU/Linux, le système maintenant dominant dans le monde de l'embarqué. Ces lecteurs USB peuvent être pris en charge de deux façons : via le middleware PCSC-Lite ou directement via la LibNFC. PCSC-Lite est un système composé d'un démon, d'un ensemble de pilotes et d'un framework permettant d'accéder aux périphériques USB CCID et fournissant une API unifiée. Il ne s'agit pas à proprement parler d'un framework pour le RFID/NFC. De ce fait, même s'il est possible d'utiliser PCSC-Lite pour accéder à un ACR122, mieux vaudra accéder directement au périphérique via la LibUSB (même si vous devez éventuellement faire des petits changements dans votre configuration udev pour gérer correctement les permissions).

La connexion d'un ACR122 provoque naturellement sa détection en tant que périphérique USB. On retrouve alors le message suivant dans les journaux :

```
usb 2-1.2: new full-speed USB device number 9 using ehci-pci
usb 2-1.2: New USB device found, idVendor=072f, idProduct=2200
usb 2-1.2: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 2-1.2: Product: ACR122U PICC Interface
usb 2-1.2: Manufacturer: ACS
```

Pour le SCL3711, nous obtenons ceci :

```
usb 2-1.2.2: new full-speed USB device number 22 using ehci_hcd
usb 2-1.2.2: New USB device found, idVendor=04e6, idProduct=5591
usb 2-1.2.2: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 2-1.2.2: Product: SCL3711-NFC&RW
usb 2-1.2.2: Manufacturer: SCM Micro
```

Notez que l'installation du paquet **libnfc5** (et **libnfc-dev** + **libnfc-bin**) sur une distribution Debian, provoquera l'installation d'un fichier **/etc/udev/rules.d/pn53x.rules** configurant udev de manière à autoriser un accès aux utilisateurs du groupe **plugdev**. Vous ne devriez donc pas avoir de soucis quant aux permissions pour utiliser ces périphériques sans **sudo**.

## 4.1 Quelques outils pratiques

La LibNFC fait partie du projet *NFC Tools* qui fournit, principalement à titre de démonstration technologique, un certain nombre d'utilitaires permettant d'accéder à différents tags (Mifare Classic, tags compatibles NFC Forum, Mifare DESFire, etc). Vous pouvez utiliser ces outils pour vous familiariser avec la technologie et faire vos premiers essais. Ainsi, par exemple, l'outil **nfc-list** vous permet d'obtenir des informations sur le lecteur (PCD) et sur le tag présent (PICC) :

```
% nfc-list
nfc-list uses libnfc 1.7.1
NFC device: ACS / ACR122U PICC Interface opened
1 ISO14443A passive target(s) found:
ISO/IEC 14443A (106 kbps) target:
    ATQA (SENS_RES): 00  04
       UID (NFCID1): 8a  f1  20  11
      SAK (SEL_RES): 08
```

À notre disposition se trouve également **nfc-mfclassic** offrant la possibilité de lire et écrire les Mifare Classic 1k et 4k :

```
% nfc-mfclassic r a dumptest.mfd
NFC reader: ACS / ACR122U PICC Interface opened
Found MIFARE Classic card:
ISO/IEC 14443A (106 kbps) target:
    ATQA (SENS_RES): 00  04
       UID (NFCID1): 8a  f1  20  11
      SAK (SEL_RES): 08
Guessing size: seems to be a 1024-byte card
Reading out 64 blocks |....................|
Done, 64 of 64 blocks read.
Writing data to file: dumptest.mfd ...Done.

% hd -v dumptest.mfd | head
00000000  8a f1 20 11 4a 08 04 00  62 63 64 65 66 67 68 69  |.. .J...bcdefghi|
00000010  fd ff ff ff 02 00 00 00  fd ff ff ff 00 ff 00 ff  |................|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  ff ff ff ff ff ff dd 27  82 69 00 00 00 00 00 00  |.......'.i......|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000070  ff ff ff ff ff ff ff 07  80 69 00 00 00 00 00 00  |.........i......|
00000080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

L'intérêt de ces outils pour une utilisation courante est relativement limité. Le plus utile reste, sans le moindre doute, la lecture des sources même s'il existe des différences entre l'implémentation de ces outils et des facilités actuellement disponibles. Les NFC Tools n'utilisent que la LibNFC alors qu'une bibliothèque spécifique aux Mifare Classic existe désormais.

## 4.2 Premier programme

Il est temps de faire un peu de code. Nous utiliserons ici la LibNFC en version 1.7.1 ainsi que la libfreefare 0.4.0. Cette dernière comprend du code initialement intégré dans la LibNFC et prenant en charge spécifiquement les tags Mifare Classic, Ultralight et DESFire. Bon nombre de documentations en ligne, détaillant la manière d'accéder aux tags Mifare, ne sont plus à jour du fait du retrait de certaines fonctions de la LibNFC. D'autres encore font usage de fonctions définies dans un fichier **mifare.c** qu'il est nécessaire d'intégrer à ses sources. Depuis, tout ceci a été transféré dans la jeune, mais déjà riche, bibliothèque libfreefare offrant un ensemble de fonctions facilitant grandement les manipulations de secteurs, de blocs et surtout des *sector trailers*.

Avant toute chose, intégrons les *headers* adéquats :

```
#include <stdlib.h>
#include <string.h>

#include <nfc/nfc.h>
#include <freefare.h>
```

Notre premier source se veut très simple. Nous nous contenterons d'initialiser les bibliothèques, trouver un lecteur (PCD), les tags, et de lister tout cela à l'utilisateur. Un certain nombre de variables de type particulier sont nécessaires :

```
int main(int argc, const char *argv[])
{
  nfc_context *context;
  nfc_device *pnd;
  MifareTag *tags = NULL;
  int i;
```

Nous avons ici respectivement de quoi stocker le contexte LibNFC découlant de l'initialisation, le périphérique de lecture/écriture et un pointeur vers une structure décrivant des tags. Nous pouvons maintenant mettre en oeuvre la chaîne d'accès en commençant par l'initialisation du contexte :

```
nfc_init(&context);
if (context == NULL) {
  printf("Unable to init libnfc\n");
  exit(EXIT_FAILURE);
}
```

Rien de bien particulier ici, ce type de chose est assez similaire à ce que l'on peut trouver avec d'autres bibliothèques, comme la LibUSB par exemple. Une fois le contexte obtenu, nous nous penchons sur le périphérique :

```
pnd = nfc_open(context, NULL);
if (pnd == NULL) {
  printf("ERROR: %s\n",
    "Unable to open NFC device.");
  exit(EXIT_FAILURE);
}
printf("NFC reader: %s opened\n", nfc_device_get_name(pnd));
```

Cette ouverture sélectionne automatiquement :

- le périphérique désigné par la variable d'environnement **LIBNFC_DEFAUT_DEVICE**,
- le premier périphérique du fichier **/etc/nfc/libnfc.conf**,
- le premier périphérique désigné par le premier fichier dans **/etc/nfc/devices.d/**,
- le premier périphérique détecté (si ceci n'est pas interdit dans **/etc/nfc/libnfc.conf**).

Bien souvent, ces fichiers de configuration n'existent simplement pas et c'est donc le premier lecteur trouvé qui est ouvert. Si l'opération réussie, **pnd** pointe sur ce périphérique et nous pouvons immédiatement afficher quelques informations, comme son nom.

Le reste, qui initialement faisait également usage de la LibNFC, dépend maintenant de la libfreefare, ce qui évite un grand nombre de configurations intermédiaires. Tout ce que nous avons à faire est d'obtenir la liste des tags présents :

```
tags = freefare_get_tags(pnd);
if (!tags) {
        printf("no tag found\n");
} else {
```

Notez que **tags** est un tableau car il est physiquement possible d'accéder à plusieurs tags en même temps sans que les communications n'interfèrent les unes avec les autres. Si vous posez plusieurs tags sur le lecteur, chacun d'entre eux peut être accédé et utilisé.

Maintenant que nous avons obtenu une liste de tags, il ne nous reste plus qu'à la parcourir en affichant quelques

informations de base puis à terminer proprement le programme en fermant le périphérique et en libérant les ressources :

```
  for (i = 0; tags[i]; i++) {
     switch(freefare_get_tag_type(tags[i])) {
        case CLASSIC_1K:
           printf("%u : Mifare 1k (S50) : %s\n",
               i, freefare_get_tag_uid(tags[i]));
           break;
        case CLASSIC_4K:
           printf("%u : Mifare 4k (S70) : %s\n",
               i, freefare_get_tag_uid(tags[i]));
           break;
        default:
           printf("%u : other ISO14443A tag : %s\n",
               i, freefare_get_tag_uid(tags[i]));
        }
     }
     freefare_free_tags(tags);
  }

  nfc_close(pnd);
  nfc_exit(context);
  exit(EXIT_SUCCESS);
}
```

La libfreefare fournit une fonction permettant de connaître le type de tag désigné : **ULTRALIGHT**, **ULTRALIGHT_C**, **CLASSIC_1K**, **CLASSIC_4K**, ou **DESFIRE**. Pour l'heure, les Mifare Mini et Mifare Plus ne sont pas supportés. Nous affichons donc le type mais également l'UID/NUID de chaque tag. Celui-ci, bien que présent dans le *manufacturer bloc* des tags, n'est pas obtenu en lisant le bloc mais transmit par le lecteur. Ceci explique pourquoi nous n'avons pas eu besoin de nous authentifier.

La compilation de ce premier code se fera via **gcc code1.c -o code1 -lnfc -lusb -lfreefare** ou en utilisant **pkg-config** (uniquement disponible pour LibNFC). Son exécution nous affichera, comme attendu :

```
NFC reader: ACS / ACR122U PICC Interface opened
0 : Mifare 1k (S50) : 8af12011
```

## 4.3 Lecture de blocs

Obtenir l'accès au périphérique, aux tags et obtenir des UID n'est pas suffisant et surtout, ne saurait faire office de quelconque mécanisme de sécurité. Pour manipuler un tag Mifare Classic, nous devons accéder aux données qu'il contient en utilisant le mécanisme d'authentification. Nous l'avons dit plus haut, deux clés peuvent être définies et utilisées et les conditions d'accès déterminent ce que nous pouvons faire après nous être authentifié avec l'une d'elle. Il existe un certain nombre de valeur de clés utilisées par défaut, en fonction du type d'utilisation du tag. Ainsi, en sortie d'usine, les tags ont comme clé par défaut **FFFFFFFFFFFF** pour A comme pour B. Cependant, un tag Mifare Classic initialisé avec des données NDEF (formaté) utilisera par défaut une clé A **A0A1A2A3A4A5**. D'autres valeurs par défaut du même type existent et on commencera donc généralement par intégrer à son code un tableau de clés **MifareClassicKey** (libfreefare) :

```
MifareClassicKey keys[] = {
  { 0xd3,0xf7,0xd3,0xf7,0xd3,0xf7 },
  { 0xa0,0xa1,0xa2,0xa3,0xa4,0xa5 },
  { 0xb0,0xb1,0xb2,0xb3,0xb4,0xb5 },
  { 0xff,0xff,0xff,0xff,0xff,0xff },
  { 0x4d,0x3a,0x99,0xc3,0x51,0xdd },
  { 0x1a,0x98,0x2c,0x7e,0x45,0x9a },
  { 0xaa,0xbb,0xcc,0xdd,0xee,0xff },
  { 0x00,0x00,0x00,0x00,0x00,0x00 }
};
```

Pour, par exemple, lire le contenu d'un tag ou plus exactement d'un bloc donné d'un secteur du tag, on utilise grossièrement la procédure suivante :

```
initialisation LibNFC
  ouverture PCD
    découverte tags
      connexion au tag
        authentification via-à-vis d'un bloc
          lecture des données
      déconnexion du tag
    libération de la liste de tags
  fermeture PCD
clôture LibNFC
```

Dans le but d'explorer le contenu d'un tag, il existe plusieurs techniques pour mettre cela en œuvre, avec plusieurs variations. Deux d'entres elles se distinguent :

- Une première tentative d'authentification avec A et B puis lecture des *access bits* afin de mapper le tag. On construit alors une carte du tag sachant avec quelle clé on peut accéder à quel bloc.
- Une lecture en une fois avec, pour chaque secteur, une tentative d'authentification avec la clé A puis B et pour chaque bloc de ce secteur, une tentative d'accès. Ici, on ne regarde pas les *access bits* et on se contente de voir l'accès accepté ou refusé.

Ceci relève de la construction du code en lui-même mais met en œuvre les même fonctions. Par soucis de lisibilité et d'économie des précieuses pages du magazine, nous opterons ici pour la seconde méthode, un peu plus « brutale ».

Vous l'avez compris, il s'agira surtout de faire des boucles. Sur la base du type de carte obtenu par **freefare_get_tag_type()**, nous pouvons en déduire le nombre de secteurs présents et initier la première boucle. Immédiatement, nous utilisons le tableau de clés pour une seconde boucle et nous testons l'authentification :

```
for(i=0; i<nbrsect; i++) {
  for(j=0; j < sizeof(keys)/sizeof(keys[0]); j++) {
    if((mifare_classic_connect(tags[0]) == OPERATION_OK) &&
        (mifare_classic_authenticate(
                                    tags[0],
                                    mifare_classic_sector_last_block(i),
                                    keys[j],
                                    MFC_KEY_A) == OPERATION_OK)) {
```

Dans une seule et même condition **if** nous nous connectons au tag et procédons à l'authentification en spécifiant le

*sector trailer*. Remarquez les fonctions de confort mises à disposition par la libfreefare comme ici **mifare_classic_sector_last_block()** nous évitant de tenir à jour des compteurs nous-même. Cette fonction retourne simplement le numéro absolu du dernier bloc du secteur spécifié. L'authentification auprès d'un bloc se fait en utilisant **mifare_classic_authenticate()** prenant en argument le tag, le numéro du bloc concerné, la clé à utiliser et le type de clé A (**MFC_KEY_A**) ou B (**MFC_KEY_B**). Notez qu'une authentification réussie ne signifie pas pour autant qu'on puisse lire ou écrire dans un bloc ou un *sector trailer* mais simplement que la clé est valide pour ce secteur.

Il ne nous reste plus, ensuite, qu'à tenter de lire les quatre blocs du secteur avec une nouvelle boucle :

```
      printf("sector %02d auth with A[%d]\n", i, j);
      for(k=mifare_classic_sector_first_block(i);
          k<=mifare_classic_sector_last_block(i); k++) {
        if(mifare_classic_read(tags[0], k, &data) == OPERATION_OK) {
          print_hex(data,16);
        } else {
          printf("read error\n");
        }
      }
```

Là encore les fonctions de confort comme **mifare_classic_sector_first_block()** et **mifare_classic_sector_last_block()** sont bien pratiques. La lecture à proprement parler est effectuée avec **mifare_classic_read()** qui prend en argument le tag, le numéro de bloc et un pointeur vers un **MifareClassicBlock**. Nous pourrons ensuite simplement afficher les valeurs hexadécimales des 16 octets ainsi lus ou, si les conditions d'accès ne nous permettent pas l'opération, signaler une erreur.

Nous terminons ensuite là nos différentes boucles :

```
      mifare_classic_disconnect(tags[0]);
      break;
    }
    mifare_classic_disconnect(tags[0]);
  }
  printf("\n");
}
```

La fonction **print_hex()** est simpliste et se résume au code suivant :

```
static void print_hex(MifareClassicBlock blkData,
                      const size_t szBytes) {
  size_t  szPos;

  for (szPos = 0; szPos < szBytes; szPos++) {
    printf("%02x ", (uint8_t)blkData[szPos]);
  }
  printf("\n");
}
```

Le code se compilera comme précédemment et son exécution nous donne quelque chose comme :

```
% make && ./mynfc3
make: Nothing to be done for 'all'.
NFC reader: ACS / ACR122U PICC Interface opened
0 : Mifare 1k (S50) : 8af12011
Found Mifare Classic 1k
sector 00 auth with A[3]
8a f1 20 11 4a 08 04 00 62 63 64 65 66 67 68 69
fd ff ff ff 02 00 00 00 fd ff ff ff 00 ff 00 ff
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 dd 27 82 69 ff ff ff ff ff ff

sector 01 auth with A[3]
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[...]
sector 15 auth with A[3]
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 ff 07 80 69 ff ff ff ff ff ff
```

Remarquez que le *sector trailer* affiché montre une clé A de **000000000000** alors que nous nous sommes pourtant authentifié avec **FFFFFFFFFFFF**. La clé A ne peut jamais être lue et est systématiquement remplacée par des **00**. Toujours sur les *sector trailers* apparaissent les *access bits*, **DD2782** pour le secteur 0 et **FF0780** pour les autres. Cette dernière valeur est celle par défaut et correspond à la configuration de transport. Notez que celle-ci laisse la clé B lisible suite à l'authentification avec la clé A. Ainsi, avec une Mifare Classic NXP, si nous nous authentifions avec la bonne clé B (**MFC_KEY_B**), nous ne pouvons pas lire les données :

```
0 : Mifare 1k (S50) : 2ce073ec
Found Mifare Classic 1k
sector 00 auth with A[3]
read error
read error
read error
read error

sector 01 auth with A[3]
read error
[...]
```

Le problème, ou plutôt ce mécanisme de sécurité, n'existe pas avec une Mifare Classic compatible de chez Fudan, par exemple, qui ne voit aucun soucis dans l'utilisation d'une clé B parfaitement lisible par la clé A.

## 4.4 Écriture et configuration des access bits

Une fois que l'on s'est amusé à développer un code souple permettant de lire le contenu des Mifare Classic afin d'acquérir une certaine compétence, l'étape suivante, en toute logique, est l'écriture et la configuration des conditions d'accès. Il s'agit là d'un processus qui peut, potentiellement, être risqué pour vos tags. Comme nous l'avons évoqué en début d'article, une mauvaise configuration des *access bits* peut rendre un tag partiellement ou totalement inutilisable, même en possédant les clés A et B. Préparez-vous donc à gâcher

des tags car les erreurs sont une excellente méthode pédagogique. En termes de coût, on trouvera des lots de 10, 25, 100 ou 200 cartes très facilement sur le web et sur eBay par exemple à moins de 1€/pièce. Bien entendu, il ne s'agit généralement pas de tags NXP mais de clones chinois.

L'écriture d'un bloc est relativement aisée. Après authentification via la clé A, ou en fonction des conditions d'accès déjà configurées, ceci revient à simplement utiliser la fonction **mifare_classic_write()** en spécifiant en argument le tag, le numéro de bloc concerné et un **MifareClassicBlock** (et non un pointeur sur un **MifareClassicBlock** comme pour la lecture). Vous pouvez composer votre **MifareClassicBlock** ou votre tableau de **MifareClassicBlock** pour une écriture en boucle, comme vous l'entendez, avec les données qui vous conviennent. Il en va de même avec le *sector trailer*.

La libfreefare met à disposition cependant là aussi un certain nombre de fonctions permettant de vous faciliter la vie. Le *sector trailer* peut être composé via la fonction **mifare_classic_trailer_block()**. Par exemple :

```
mifare_classic_trailer_block (
    &my_trailer_block, // pointeur sur le bloc à composer
    my_key_A,          // MifareClassicKey pour la clé A
    C_000,             // access bits pour le 1er bloc
    C_011,             // access bits pour le 2eme bloc
    C_000,             // access bits pour le 3eme bloc
    C_100,             // access bits pour sector trailer
    0x69,              // l'octet d'usage général
    my_key_B);         // MifareClassicKey pour la clé B
```

Notez, c'est important, que les macros mises à disposition font référence aux *access bit* sous la forme **C3C2C1** et non **C1C2C3** comme dans les *datasheets* des tags Mifare Classic. Ainsi, **C_011** défini ici pour le second bloc du secteur, configure un bloc en *value block* avec lecture A ou B, écriture avec B, incrémentation avec B, et décrémentation avec A ou B, et non un bloc standard avec lecture/écriture avec B uniquement. Faites excessivement attention à cela, en particulier pour les *access bits* du *sector trailer*, car sur les 8 combinaisons possibles, 5 bloquent définitivement l'écriture des *access bits* !

L'écriture de ce bloc ainsi « forgé » se fera via :

```
mifare_classic_write (
    tags[0],
    mifare_classic_sector_last_block(0),
    my_trailer_block);
```

Nous avons ici configuré un *sector trailer* pour que le second bloc du secteur soit un *value block*. Il faut donc formater les données de ce bloc de manière à ce qu'elles soient compatibles (redondance des données, inversion, octet d'adresse, etc). Là encore, libfreefare vient à la rescousse et nous n'avons pas besoin de nous torturer l'esprit pour cette tâche :

```
mifare_classic_init_value(
   tags[0],
   mifare_classic_sector_first_block(0)+1,
   0x42,    // valeur
   0);      // adr
```

Nous initialisons ici le second bloc du premier secteur avec la valeur **0x42** et spécifions l'adresse (pour l'éventuel backup) à **0x00**. La fonction produira et enregistrera automatiquement un bloc **42000000BDFFFFFF4200000000FF00FF**, ou en d'autre terme :

```
42000000 : valeur
BDFFFFFF : valeur inversée
42000000 : valeur répétée
00 : adresse
FF : adresse inversée
00 : adresse répétée
FF : adresse inversée répétée
```

Remarquez l'endianness de la valeur 32 bits ainsi enregistrée. Bien entendu, libfreefare met également à disposition une fonction permettant de simplifier la lecture des *value blocks* : **mifare_classic_read_value()** qui prend en argument le tag, le bloc, un pointeur sur un **int32_t** pour la valeur et un pointeur sur un **MifareClassicBlockNumber** pour l'adresse.

A partir du moment où les *access bits* sont correctement configurés et le contenu du bloc en rapport avec le comportement attendu, il devient possible d'utiliser les fonctions **mifare_classic_increment()** et **mifare_classic_decrement()**, prenant en argument le tag visé, le numéro du bloc et la quantité à incrémenter/décrémenter. Ceci n'applique pas directement l'opération mais place le résultat dans un registre interne du tag qu'on peut ensuite enregistrer avec **mifare_classic_transfer()** en spécifiant le bloc de destination (qui peut être le même bloc ou celui désigné par l'octet d'adresse du bloc initial). Il est également possible d'annuler l'opération avec **mifare_classic_restore()**.

## 5 Failles et limitations

Deux énormes problèmes mettent en péril la sécurité des Mifare Classic qui, je le rappelle, ne devraient plus être utilisées pour des nouveaux projets. Le premier problème concerne la sécurité elle-même, l'algorithme utilisé (CRYPTO-1), et le générateur de nombre pseudo-aléatoire (LFSR) implémenté dans les tags. De nombreuses attaques sont ainsi possibles et la plus connue est sans doute la « *nested authentication attack* », utilisée par l'outil MFOC. Ce dernier est en mesure, si un seul secteur du tag est accessible par une clé connue, de retrouver l'ensemble des clés (A et B).

En guise de démonstration, nous avons utilisé un tag Mifare Classic 1k (non NXP) et avons spécifié une clé A à **426942694269** pour le secteur 0 et laissé les autres secteurs accessibles avec les clés A et B par défaut (**FFFFFFFFFFFF**).

Nous avons ensuite utilisé MFOC en version 0.10.6 pour tester la sécurité de notre tag :

```
% ./mfoc -P 50 -T 30 -O mycard.mfd
[...]
Try to authenticate to all sectors with default keys...
[Key: ffffffffffff] -> [\xxxxxxxxxxxxxx]
[Key: a0a1a2a3a4a5] -> [\xxxxxxxxxxxxxx]
[Key: d3f7d3f7d3f7] -> [\xxxxxxxxxxxxxx]
[Key: 000000000000] -> [\xxxxxxxxxxxxxx]
[Key: b0b1b2b3b4b5] -> [\xxxxxxxxxxxxxx]
[Key: 4d3a99c351dd] -> [\xxxxxxxxxxxxxx]
[Key: 1a982c7e459a] -> [\xxxxxxxxxxxxxx]
[Key: aabbccddeeff] -> [\xxxxxxxxxxxxxx]
[Key: 714c5c886e97] -> [\xxxxxxxxxxxxxx]
[Key: 587ee5f9350f] -> [\xxxxxxxxxxxxxx]
[Key: a0478cc39091] -> [\xxxxxxxxxxxxxx]
[Key: 533cb6c723f6] -> [\xxxxxxxxxxxxxx]
[Key: 8fd0a4f256e9] -> [\xxxxxxxxxxxxxx]

Sector 00 -  UNKNOWN_KEY [A]  Sector 00 -  FOUND_KEY   [B]
Sector 01 -  FOUND_KEY   [A]  Sector 01 -  FOUND_KEY   [B]
Sector 02 -  FOUND_KEY   [A]  Sector 02 -  FOUND_KEY   [B]
Sector 03 -  FOUND_KEY   [A]  Sector 03 -  FOUND_KEY   [B]
Sector 04 -  FOUND_KEY   [A]  Sector 04 -  FOUND_KEY   [B]
Sector 05 -  FOUND_KEY   [A]  Sector 05 -  FOUND_KEY   [B]
Sector 06 -  FOUND_KEY   [A]  Sector 06 -  FOUND_KEY   [B]
Sector 07 -  FOUND_KEY   [A]  Sector 07 -  FOUND_KEY   [B]
Sector 08 -  FOUND_KEY   [A]  Sector 08 -  FOUND_KEY   [B]
Sector 09 -  FOUND_KEY   [A]  Sector 09 -  FOUND_KEY   [B]
Sector 10 -  FOUND_KEY   [A]  Sector 10 -  FOUND_KEY   [B]
Sector 11 -  FOUND_KEY   [A]  Sector 11 -  FOUND_KEY   [B]
Sector 12 -  FOUND_KEY   [A]  Sector 12 -  FOUND_KEY   [B]
Sector 13 -  FOUND_KEY   [A]  Sector 13 -  FOUND_KEY   [B]
Sector 14 -  FOUND_KEY   [A]  Sector 14 -  FOUND_KEY   [B]
Sector 15 -  FOUND_KEY   [A]  Sector 15 -  FOUND_KEY   [B]
[...]
Using sector 00 as an exploit sector
Sector: 0, type A, probe 0, distance 15147 .....
Sector: 0, type A, probe 1, distance 15299 .....
Sector: 0, type A, probe 2, distance 15259 .....
Found Key: A [426942694269]
Auth with all sectors succeeded, dumping keys to a file!
```

L'opération a durée environ 75 secondes, l'utilitaire nous affiche fièrement **Found Key: A [426942694269]** avant de lire le tag et enregistrer les données, clés incluses, dans le fichier **mycard.mfd**. MFOC intègre, comme vous le voyez, un certain nombre de clés par défaut qui permettent éventuellement de trouver un secteur sur lequel on peut s'authentifier. Dès lors que cette opération a réussi, l'attaque est portée et conduit généralement rapidement à un résultat satisfaisant.

Un autre type d'attaque peut être porté contre un tag, la « *Darkside attack* » visant une faiblesse dans l'implémentation du PRNG. Là encore un outil existe, c'est MFCUK. Il est en mesure, avec certains tags, de récupérer une des clés avec un délai entre quelques secondes et 30 minutes. Une fois la clé récupérée, il suffit d'utiliser MFOC pour obtenir toutes les autres. Notez cependant que les tags récents utilisent une implémentation corrigée du PRNG et ne sont plus sensibles à cette attaque.

Les données ainsi récupérées peuvent, soit être analysées pour déterminer le type, le formatage et la nature des données stockées ou, tout simplement, permettre la création d'un clone. « Mais le *manufacturer block* est en lecture-seule », direz-vous. Pas tout à fait ou, du moins, pas pour tous les tags. La fabrication de tags est un processus industriel et il arrive qu'un certain nombre d'entre eux passent au travers des contrôles qualité tout en ayant un bloc 0 inscriptible. Pendant un temps, le défi consistait à trouver ce type de tag et les utiliser pour produire des clones de tag 100% identiques à l'original. Depuis, les choses ont un peu changé et certains fabricants chinois produisent tout simplement des tags spéciaux disposant d'un bloc 0 réinscriptible sous certaines conditions. Il est important de faire la distinction entre un tag entièrement inscriptible difficile à trouver et un de ces tags spéciaux disponibles en ligne pour quelques 5€ ou 10€, libellés «*UID Changeable* » et vendus comme » *a perfect solution for a lost irreplaceable Mifare Cards ID* » tout en précisant « *Please kindly use it in the legal ways* »...

Un outil livré parmi les codes exemples des NFC Tools, **nfc-mfsetuid**, est conçu pour ces tags chinois spéciaux et permet, par un jeu de commandes particulier, de débloquer le bloc 0 et d'y inscrire une valeur de son choix. Certaines outils de clonage utilisent même directement cette technique pour procéder à une copie complète d'un tag.

Le duo outils de récupération des données/clés + tags chinois spéciaux transforme littéralement les Mifare Classic en de vulgaires mémoires sans aucune sécurité, exactement comme des EEPROMs ou des pistes magnétiques. Elles sont pourtant encore très largement utilisées pour des applications sensibles comme le contrôle d'accès, les titres de transport ou le stockage de valeurs monétaires. L'arrivé des Mifare Plus, compatibles Mifare Classic, permet de changer cet état de fait, mais au prix d'un renouvellement complet des tags que le prestataire devra alors justifier, annonçant ainsi à ses clients que la sécurité de son produit était jusqu'alors très limité. Toutes les sociétés ne sont pas prêtes à appliquer ce genre de procédures, en dehors de quelques rares exceptions.

Le résultat est édifiant, on trouve des tags Mifare Classic partout et le parc est souvent renouvelé régulièrement sans aucune considération pour les problèmes de sécurité. Rappelons que l'on parle ici de technologie sans contact, ce qui implique des risques importants de vol d'informations dans les lieux publiques. Par défaut, la lecture d'un tag nécessite une proximité de l'ordre de 5 ou 10 cm ce qui est déjà important (transport en commun bondé) mais il existe des façons d'étendre cela à 20, 30 voir 70 cm en utilisant des antennes particulières ou des systèmes d'amplification. Il est donc très fortement recommandé de placer ce type de tags à l'abri des regards ou des oreilles indiscrètes en les plaçant dans des étuis ou pochettes de protection (principe de la cage de Faraday). Il est finalement amusant de constater que le tapage fait autour des systèmes de paiement sans contact qui provoque la prolifération de ce type de produit est une aubaine pour tous les utilisateurs de technologies Mifare Classic qui méritent ce type de protection depuis des années... ■

# POUR EXPLORER, UTILISONS PYTHON !

*par Denis Bodor*

Les plateformes embarquées sont de plus en plus puissantes et ressemblent, au fil des jours, de plus en plus à des nano-ordinateurs comme c'est le cas par exemple pour la très populaire Raspberry Pi. En conséquence, l'accessibilité de ce type de cartes s'en trouve augmentée, au point que les utilisateurs ne sachant pas développer dans un langage de bas niveau comme le C peuvent, eux aussi, en bénéficier, par exemple au travers du langage Python. Or, il existe un ensemble d'outils et de bibliothèques Python permettant d'explorer les technologie RFID : c'est RFIDIOt !

C'est Adam Laurie qui a développé RFIDIOt (*RFID I/O tools*) et choisi le nom du projet sur la base de son expérience, comme il le dit lui-même : « J'aime les calembours. De plus, j'en suis venu à ce projet avec le point de vue d'un idiot : je n'y connaissais rien aux tags RFID, et encore moins concernant Python. Ainsi, je me sentais totalement idiot quand j'ai commencé ». Derrière cette modeste remarque se cache non seulement un expert en sécurité reconnu mais également un projet très riche en fonctionnalités.

RFIDIOt n'est généralement pas intégré dans le système de gestion de paquets des distributions GNU/Linux. Il est donc nécessaire de récupérer directement les sources sur le serveur GitHub et de procéder à l'installation :

```
% git clone https://github.com/AdamLaurie/RFIDIOt.git
Cloning into 'RFIDIOt'...
remote: Reusing existing pack: 356, done.
remote: Total 356 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (356/356), 673.44 KiB | 398 KiB/s, done.
Resolving deltas: 100% (187/187), done.

% cd RFIDIOt/

% python setup.py build
running build
running build_py
creating build
creating build/lib.linux-i686-2.7
creating build/lib.linux-i686-2.7/rfidiot
[...]
changing mode of build/scripts-2.7/unique.py
  from 644 to 755
changing mode of build/scripts-2.7/writelfx.py
  from 644 to 755
changing mode of build/scripts-2.7/writemifare1k.py
  from 644 to 755
```

```
% % sudo python setup.py install
running install
running build
running build_py
[...]
changing mode of /usr/local/bin/eeprom.py to 755
running install_egg_info
Removing /usr/local/lib/python2.7/
   dist-packages/rfidiot-1.0.egg-info
Writing /usr/local/lib/python2.7/
   dist-packages/rfidiot-1.0.egg-info
changing mode of /usr/local/bin/eeprom.py to 755
running install_egg_info
Removing /usr/local/lib/python2.7/
  dist-packages/rfidiot-1.0.egg-info
Writing /usr/local/lib/python2.7/
  dist-packages/rfidiot-1.0.egg-info
```

RFIFIOt est en mesure d'utiliser plusieurs *backend* du lecteur série à celui accessible par PCSC-Lite en passant par la LibNFC. Cette configuration se fait en ligne de commande en rédigeant un petit fichier **RFIDIOtconfig.opts** contenant les options à passer en argument. Notez que dans les sources de RFIDIOt est livré un tel fichier, mais un petit problème existe :

```
#-s 9600 -l /dev/ttyUSB1
# uncomment the above line to enable global overrides
# options should be all on the first line, as if
# added on the command line
# this file will be looked for in the following places:
#   $(RFIDIOtconfig_opts) - note that this environment
# variable should be set to the full path and filename
#   ./RFIDIOtconfig.opts
#   /etc/RFIDIOtconfig.opts
```

Le contenu décrit l'utilisation du fichier et les différents emplacements où il est automatiquement recherché. Cependant, bien que le texte soit placé en commentaire, le fichier ne peut contenir que les options à passer en argument. On aurait, à tord, envie de simplement ajouter une ligne **-R READER_LIBNFC** par exemple. Les différents scripts Python de démonstration utilisent, en effet, le nombre d'arguments pour vérification via :

```
args= rfidiot.args
help= rfidiot.help
[...]
if help or len(args) > 6:
[...]
```

Or, en laissant le texte en commentaire, celui-ci est interprété comme autant d'arguments qui ne sont pas pris en charge par RFIDIOt et sont donc potentiellement destinés au script. Comme ceci découle sur quelques 60 arguments non traités, la condition **if** échoue et on se retrouve, quoi qu'on fasse, à chaque exécution, avec un résumé de la syntaxe de la commande (**readmifaresimple.py** en l'occurrence). RFIDIOt est perfectible et aurait bien besoin d'un développeur Python senior.

## 1 Prise en main de RFIDIOt

Comme à son habitude, le langage Python s'avère idéal pour toutes les phases de prototypage en offrant une simplicité extrême de prise en main. Autre avantage non négligeable, l'étendue des périphériques supportés s'en trouve augmentée du fait de la compatibilité avec différentes solutions de prise en charge. Il est même possible d'utiliser une application Android au travers du réseau soit via **netcat** soit directement avec RFIDIOt en appliquant un patch distribué avec les sources (non testé).

Si vous avez déjà lu l'article consacré aux tags Mifare Classic et à l'utilisation de la LibNFC, le code va vous sembler d'une simplicité déconcertante. Exemple :

```
#!/usr/bin/python

import sys
import os
import rfidiot

args = rfidiot.args
help = rfidiot.help

try:
        card = rfidiot.card
except:
        print "Couldn't open reader!"
        os._exit(True)

card.select()

if not card.select():
        card.waitfortag('waiting for Mifare TAG...')

print 'Card ID: ' + card.uid

block = 0

if not card.login(block,'AA','FFFFFFFFFFFF'):
        print 'Read error: %s %s' % \
          (card.errorcode, \
           card.ISO7816ErrorCodes[card.errorcode])
        os._exit(True)

while block <= 3:
        if card.readMIFAREblock(block):
                print card.MIFAREdata
        block += 1
```

Ce qui aura pour résultat :

```
Card ID: 2AB84D55
2AB84D558A08040062636465666768690
00000000000000000000000000000000
00000000000000000000000000000000
000000000000FF078069FFFFFFFFFFFF
```

Nous venons de lire très simplement le premier secteur d'un tag et d'en afficher le contenu à l'écran. La méthode **rfidiot.card** nous fourni un accès au lecteur et nous nous connectons au tag avec la méthode **select** de l'objet obtenu. Il nous suffit ensuite de nous authentifier sur le secteur de notre choix avec **login** et de boucler avec **readMIFAREblock** pour obtenir le contenu de chacun des quatre blocs.

Comme la gestion de chaîne de caractères est un vrai bonheur avec Python (par comparaison au C), il est aisé de manipuler ces informations avec des choses comme **card.MIFAREdata[0:12]** ou encore **card.MIFAREdata[20:]**. On aura tôt fait, par exemple, d'ajouter un peu de couleur via des caractères d'échappement, afin de rendre la sortie plus facile à interpréter, voire de faire intervenir quelques bibliothèques graphiques ou *toolkits* quelconque pour créer une interface graphique.

## 2 Un peu plus loin ? Ou pas

RFIDIOt, via la LibNFC, s'en sort relativement bien lorsqu'il s'agit de lire des tags Mifare Classic bien que nous ayons rencontré quelques problèmes de déconnexion, alors qu'avec la LibNFC, via un code en C, les même tags répondaient systématiquement. Étrangement, le plus souvent ces problèmes se posaient avec le périphérique SCL3711 et non l'ACR122U. L'écriture d'un bloc ou l'utilisation des codes exemples comme **readmifaresimple.py** (qui est moins simple que **readmifare1k.py** ?!) afin d'écrire (effacer) des blocs d'un tag, s'est avérée tout aussi problématique.

Pourtant, RFIDIOt est riche en termes de fonctionnalités, peut être trop justement. La diversité des options qui s'offrent à nous semble avoir un coût en termes de stabilité du code et de qualité de la documentation, quasi inexistante. Sans vouloir critiquer un travail qui est remarquable par ailleurs, un seul coup d'œil au dépôt GitHub donne une vague idée du désordre qui règne dans les sources et du caractère bêta de l'ensemble.

D'autres solutions existent comme Pynfc de Mike Auty ou encore nfcpy de Stephen Tiedemann mais ne sont malheureusement pas aussi stables ou riches que la LibNFC. La conclusion vient donc d'elle-même et peut se résumer ainsi : si vous comptez lire des tags, RFIDIOt est un bon choix, tout comme si vous comptez participer au développement du projet, mais pour des utilisations avancées, vous n'aurez pas d'autres choix que de vous pencher sur le C et la LibNFC directement. ■

# MIFARE DESFIRE : UN NIVEAU DE SÉCURITÉ ADAPTÉ

*par Denis Bodor*

Comme toutes technologies, celle des cartes à puce sans contact connaît ses hauts et ses bas, ses successions de générations et ses évolutions notables. C'est ainsi qu'aujourd'hui, la très populaire et courante Mifare Classic, laisse sa place à une technologie bien plus sûre et avancée, basée sur une architecture radicalement différente, très proche de ce que l'on trouve avec les JavaCards. Cette technologie s'appelle Mifare DesFire et en particulier DESFire EV1.

Avant toute chose, précisons que le matériel utilisé pour la lecture et l'écriture des tags DESFire est le même que pour les Mifare Classic. Il ne vous sera donc, en principe, pas nécessaire de devoir acquérir un matériel différent, plus coûteux. Il en va de même pour la partie logicielle et en particulier les bibliothèques LibNFC et libfreefare, prenant parfaitement en charge cette nouvelle génération (2008 pour la Mifare DESFire EV1). Là encore, malheureusement diront certains, le langage de prédilection reste le C étant donné l'état actuel d'autres implémentations, telles que les solutions en Python, Perl, etc. Il semblerait cependant que la communauté des développeurs Java, quant à elle, bénéficie de beaucoup plus d'options, ce qui est sans doute non sans relation avec les technologies JavaCard qui restent assez proches (il existe d'ailleurs des tags/cartes à double interface puce/contactless). Nous nous en tiendrons cependant ici au C, bien plus en rapport avec l'objet de cette série d'articles (et les préférences de l'auteur).

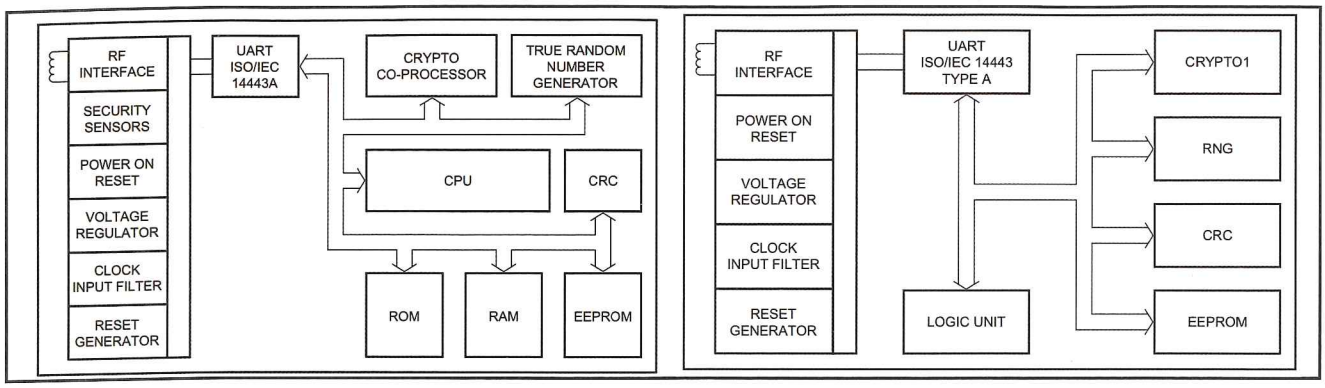
## 1 DESFire

Pour appréhender au mieux la technologie DESFire, il est préférable d'oublier la majeure partie de ce que l'on sait sur les Mifare Classic. En effet, l'architecture de DESFire est totalement différente. Alors qu'un tag Classic peut être vu comme une EEPROM évoluée, organisée en blocs et secteurs, un tag DESFire lui est bien plus proche d'un ordinateur sur une puce. DESFire est en réalité une version particulière de la plateforme SmartMX de NXP préprogrammée avec le système d'exploitation DESFire.

DESFire est donc beaucoup plus évolué que Mifare Classic et la notion de bloc est totalement écartée au bénéfice d'une organisation mémoire (2Ko, 4Ko ou 8Ko selon les modèles) se rapprochant de ce qu'on trouve avec les JavaCard ou encore, tout simplement, avec les systèmes de fichiers utilisés sur nos ordinateurs. Les essais faits lors de la rédaction de cet article ont principalement porté sur des tags DESFire EV1 (MF3ICD81) de 8Ko, laissant une grande liberté d'exploration et disposant d'un maximum d'espace de stockage. Les tags DESFire EV1 (EVolution-1) sont compatibles DESFire (2002) et apportent en plus le support du chiffrement AES 128 bits, les UIDs aléatoires et un niveau d'assurance EAL4+ (*Common Criteria* ISO 15408). De plus, contrairement aux Mifare Classic, les DESFire offrent une réelle compatibilité NFC Forum (ISO/IEC 14443A Type 4) et fonctionneront donc avec n'importe quel périphérique NFC et en particulier avec les nouveaux smartphones n'utilisant pas de contrôleurs de marque NXP (Google Nexus 4, 5, 7 (2013), 10, etc).

Le nom « DESFire » est décrit dans la documentation NXP comme susceptible d'évoquer à la fois un niveau important de sécurité (« *DES* » pour le moteur 3DES/AES intégré) et de puissance (« *Fire* ») avec des performances respectables en terme de rapidité (848 kbit/s) et de fonctionnalités. Il est vrai que le choix de ce nom est relativement pertinent car, en effet, il évoque « quelque chose de sérieux ».

Enfin, précisons qu'en novembre 2013, NXP a annoncé DESFire EV2 qui




*Comparaison des architectures Mifare DESFire et Classic, avec d'un côté un véritable système complet (CPU/RAM/ROM/périphérique) et de l'autre quelque chose qui se rapproche davantage d'une EEPROM intelligente.*

sera compatible EV1 et offrira en plus des mécanismes de sécurité avancés comme une architecture de virtualisation de carte, la mise à disposition d'espace pour des tiers (sans divulgation de la clé maîtresse/principale) ou encore une protection contre les attaques par relais (MITM).

Contrairement aux Mifare Classic, les DESFire ne disposent pas d'une mémoire organisée mais d'un espace disponible dépendant du modèle de tag. Pour que le tag soit utilisable il doit donc être formaté et structuré, exactement comme un support de stockage courant. Ainsi, la mémoire peut être divisée en applications identifiées par un AID (*Application IDentifier*) de 3 octets. Un tag peut supporter jusqu'à 28 applications qui peuvent être vues comme des répertoires. Chaque application peut contenir jusqu'à 16 ou 32 fichiers (selon le modèle de tag) de tailles arbitrairement choisies. Ces fichiers peuvent être de cinq types :

- Fichier de données standard,
- Fichier de sauvegarde,
- Fichier « valeur » (similaire aux valus *blocks* des Mifare Classic) avec sauvegarde,
- Fichier contenant des enregistrements de façon linéaire, avec sauvegarde,
- Fichier d'enregistrements cycliques (le plus récent écrase le plus ancien), avec sauvegarde.

Comme vous pouvez le constater, si vous avez lu l'article sur les Mifare Classic, nous avons ici affaire à une technologie bien plus avancée et plus riche. La sécurité, quant à elle, est également très différente, à commencer par le chiffrement des communications entre le tag (PICC) et le lecteur (PCD) qui peut être, au choix :

- En clair (mode de compatibilité),
- En clair avec somme de contrôle cryptographique (MAC/CMAC),
- Chiffrées DES/3DES/AES.

La gestion des clés est totalement différente et n'est pas liée à la structure matérielle rigide de la mémoire. Un tag DESFire dispose d'une clé principale (*Master PICC key*) permettant la gestion du tag mais se sont les applications qui « portent » les clés d'accès pouvant aller jusqu'à 14 avec, là aussi, une clé maîtresse (0) par application. C'est ensuite au niveau « fichier » que des droits d'accès permettent de définir ce qu'il est possible de faire avec chacune des 16 options : 14 clés + accès libre + accès interdit. Ces droits d'accès (*access rights* ou AR) décrivent s'il est possible de lire, écrire, lire/écrire ou changer les droits après s'être authentifié auprès de l'application avec une clé donnée. Il est donc envisageable de mettre facilement en place une véritable gestion de permissions avancées. Sachez qu'il est, de plus, possible d'utiliser un système de dérivation de clés ainsi que de définir des versions pour les clés et donc, de mettre en œuvre une politique de sécurité dynamique permettant de facilement piloter un parc important d'utilisateurs avec un minimum de contraintes techniques.

## 2 Dans le vif du sujet : du code !

Comme toujours, la solution favorite dans le magazine pour appréhender et comprendre une technologie est de passer par la case « pratique et code ». Nous allons donc maintenant faire le tour des manipulations importantes en rapport avec les tags DESFire EV1. Notre démarche sera la suivante : obtenir des informations sur un tag présenté au lecteur (PCD), formater le tag en effaçant totalement son contenu, créer une application de démonstration utilisant un ensemble de clés dont une clé AES ainsi qu'un fichier lisible uniquement grâce à cette clé, et enfin, lire le tag ainsi initialisé pour afficher le contenu du fichier via la clé AES.

Un certain nombre de points ne sera pas couvert par cette démarche comme par exemple, la diversification des clés via l'UID aléatoire, la gestion de versions de clés ou encore le changement de la clé maîtresse du tag qui, par défaut, est une clé DES égale à `0x0000000000000000`.

## 2.1 Sécurité

Cet article choisi de couvrir les tags Mifare DESFire EV1 (puces MF3ICD81/MF3ICD41/MF3ICD21) pour plusieurs raisons parmi lesquelles le fait que la sécurité des DESFire (puce MF3ICD40) a été mise à mal par des étudiants de l'université de Bochum en 2011. Le groupe de recherche a, bien entendu, signalé le problème à NXP qui, en septembre 2011, a pris les mesures qui s'imposaient pour informer ses clients. L'attaque est relativement complexe et difficile à reproduire. Elle se base sur une analyse pointue de la consommation de courant et du rayonnement électromagnétique qui permet alors de déduire le comportement interne du tag et dont découle la récupération de la clé utilisée (attaque par canal auxiliaire de type DAP pour *Differential Power Analysis*). Comme le précise à la fois le groupe de recherche indépendant et NXP, ceci demande non seulement des connaissances techniques importantes mais également du matériel adapté ainsi qu'un environnement d'expérimentation parfaitement contrôlé. Il n'est donc pas possible de porter cette attaque « sur le terrain » mais nécessitera le vol d'un tag. NXP précise dans son communiqué que ceci doit limiter les conséquences pour les utilisateurs finaux car, de plus, il n'y a généralement pas de données bancaires enregistrées sur des tags DESFire MF3ICD40. On se permettra cependant de mitiger cette affirmation puisqu'il est aisé de constater que des tags moins sécurisés (Mifare S50 ou Ultralight) sont, dans les faits, utilisés pour des applications monétisées. Tout dépend donc du sens et de la portée que l'on donne aux termes « données bancaire » (*banking data*).

Quoi qu'il en soit, cette attaque existe et est reproductible, mettant à mal la sécurité des tags DESFire qui ne sont plus produits depuis la fin 2011, remplacés par les DESFire EV1 qui, pour l'instant, n'ont pas encore été attaqués avec succès. Soyez donc prudent quant à l'achat de tags DESFire, en particulier si vous comptez implémenter une politique de sécurité forte. Il semblerait que certains distributeurs (chinois ou hollandais) écoulent de vieux stocks de ce type de tags.

## 2.2 Obtenir des informations sur un tag

Nous ne reprendrons pas ici en détail les explications déjà données sur l'utilisation de la LibNFC et la libfreefare concernant l'accès au lecteur (PCD) et au tag (PICC). Dans les grandes lignes, la procédure est strictement identique :

- Initialiser la libNFC

```
nfc_context *context;
nfc_init (&context);
```

- Obtenir un accès au lecteur

```
nfc_device *pnd = NULL;
pnd = nfc_open(context, NULL);
```

- Obtenir la liste des tags présents

```
MifareTag *tags = NULL;
tags = freefare_get_tags(pnd);
```

- Tester si le premier tag est un DESFire

```
if(freefare_get_tag_type(tags[0]) != DESFIRE) {
  printf("tag 0 is not DESFIRE!\n");
  freefare_free_tags(tags);
  nfc_close(pnd);
  nfc_exit(context);
  exit(EXIT_FAILURE);
}
```

- Obtenir un accès au tag

```
mifare_desfire_connect(tags[0])
```

On prendra soin, bien entendu, de tester les valeurs de retour des ces différentes fonctions (**OPERATION_OK**) et de réagir en conséquence, soit en terminant le programme soit en imbriquant les conditions **if** de manière, le cas échéant, à fermer le périphérique et libérer les ressources proprement. Notez que nous décidons ici de nous intéresser uniquement au premier tag détecté afin de ne pas surcharger le code exemple de boucles fastidieuses.

Une fois l'accès au **tags[0]** obtenu suite à la vérification de son modèle, nous pouvons commencer à obtenir des informations génériques :

```
struct mifare_desfire_version_info info;
mifare_desfire_get_version(tags[0], &info);
```

Cette fonction, si elle retourne **OPERATION_OK**, peuple une structure **mifare_desfire_version_info** (**freefare.h**) regroupant toutes les informations génériques sur le tag lui-même :

```
struct mifare_desfire_version_info {
    struct {
        uint8_t vendor_id;
        uint8_t type;
        uint8_t subtype;
        uint8_t version_major;
        uint8_t version_minor;
        uint8_t storage_size;
        uint8_t protocol;
    } hardware;
    struct {
        uint8_t vendor_id;
        uint8_t type;
        uint8_t subtype;
        uint8_t version_major;
        uint8_t version_minor;
        uint8_t storage_size;
        uint8_t protocol;
    } software;
    uint8_t uid[7];
    uint8_t batch_number[5];
    uint8_t production_week;
    uint8_t production_year;
};
```

Il nous suffit alors d'utiliser cette structure pour afficher toutes les informations, ce qui nous donnera un résultat comme :

```
% ./desfire_info
NFC reader: ACS / ACR122U PICC Interface opened
UID:                    0x043d555afc2e80
Batch number:           0xba4415aba0
Production date:        week 5, 2013
Hardware Information:
    Vendor ID:          0x04
    Type:               0x01
    Subtype:            0x01
    Version:            1.0
    Storage size:       0x1a (=8192 bytes)
    Protocol:           0x05
Software Information:
    Vendor ID:          0x04
    Type:               0x01
    Subtype:            0x01
    Version:            1.4
    Storage size:       0x1a (=8192 bytes)
    Protocol:           0x05
```

Notez que ces propriétés sont bien plus détaillées que celles simplement fournies par un tag Mifare Classic et son bloc 0. Nous trouvons ici des informations aussi poussées que le numéro de lot ou encore la date de fabrication du tag. Nous obtenons également des données sur les caractéristiques matérielles et logicielles qui peuvent vous permettre de connaître le type exact de tag. Voici, à titre d'exemple, les données d'un tag acquises via un site d'enchère en ligne cumulant DESFire (MF3ICD40), une puce JCOP 2.4 et une piste magnétique haute-coercivité (HiCo) :

```
UID:                    0x042f5952c32080
Batch number:           0xba1519b280
Production date:        week 50, 2010
Hardware Information:
    Vendor ID:          0x04
    Type:               0x01
    Subtype:            0x01
    Version:            0.2
    Storage size:       0x18 (=4096 bytes)
    Protocol:           0x05
Software Information:
    Vendor ID:          0x04
    Type:               0x01
    Subtype:            0x01
    Version:            0.6
    Storage size:       0x18 (=4096 bytes)
    Protocol:           0x05
```

Remarquez la date de fabrication ainsi que les différentes versions affichées à la fois pour le matériel et le logiciel. La provenance de ce tag suscite des questions mais son utilisation pour un projet où la sécurité est critique, elle, est tout à fait claire : à éviter.

A ce stade, nous pouvons également obtenir des informations sur la clé maîtresse du tag (PICC) avec la fonction :

```
uint8_t settings;
uint8_t max_keys;
mifare_desfire_get_key_settings(tags[0], &settings, &max_keys);
```

suivie de l'affichage des bits intéressants de **settings**. Ce qui nous donnera alors :

```
Master Key settings (0x0f):
    0x08 configuration changeable;
    0x04 PICC Master Key not required for create / delete;
    0x02 Free directory list access without PICC Master Key;
    0x01 Allow changing the Master Key;
```

On prendra soin de vérifier la valeur retournée par la fonction et en particulier si elle est égale à **AUTHENTICATION_ERROR** car ceci signifierait alors que les paramètres de la clé maîtresse du tag sont verrouillés. Cette sortie, comme le reste du code, est inspirée des exemples livrés avec la lib-freefare et nécessite quelques explications importantes. Les paramètres de clés, qu'ils concernent la clé maîtresse PICC ou la/les clé(s) des applications, définissent plusieurs autorisations en rapport avec les clés et la lecture des informations :

- **0x08**, bit 3 à 1 : la configuration des clés peut changer et n'est donc pas gelée.
- **0x04**, bit 2 à 1 : la clé maîtresse n'est PAS nécessaire pour créer/effacer les applications (niveau PICC) ou les fichiers (niveau application),
- **0x02**, bit 1 à 1 : la clé maîtresse n'est PAS nécessaire pour lister les applications (niveau PICC) ou les fichiers (niveau application),
- **0x01**, bit 0 à 1 : la clé maîtresse peut être changée.

Notez que dans ce code, nous n'avons pas sélectionné d'applications car, en nous connectant au tag, l'application ayant pour AID **0x000000** est sélectionnée automatiquement. Elle correspond à l'application représentant le tag lui-même, l'application maîtresse en quelque sorte et est tantôt décrite dans la documentation comme « l'application au niveau carte » (*card level application*). Si vous sélectionnez une autre application et souhaitez revenir à cette dernière, il vous suffira d'utiliser **mifare_desfire_select_application(tags[0], NULL)**. Le second argument à **NULL** en guise de **MifareDESFireAID** joue ce rôle.

Enfin, dernière information qui peut être utile, nous pouvons également nous renseigner sur l'espace encore disponible sur le tag avec :

```
uint32_t size;
mifare_desfire_free_mem(tags[0], &size);
```

Notez qu'il s'agit là de l'espace encore disponible car non utilisé par les fichiers des applications. Ceux-ci sont créés avec une taille maximum fixe et cette valeur retournée par la fonction ne tient donc pas compte de l'espace effectivement

utilisé dans les fichiers. Dans le cas, par exemple, d'un tag DESFire EV1 de 8ko formaté avec des données NDEF (NFC Forum type 4), une application AID **0x000001** est créée dans laquelle se trouve deux fichiers (**0x01** et **0x02**) respectivement de 15 octets et de 7680 octets, même s'il ne contient rien. Avec un tel tag, cette fonction retournera un espace disponible de 0 octet car c'est au niveau NDEF que la gestion de ces données prend place. On peut considérer cela comme un système de fichiers structuré construit au dessus du système de stockage des tags DESFire EV1.

## 2.3 Créer une application et un fichier

Il est maintenant temps de faire réellement quelque chose avec ce tag DESFire et donc, d'y stocker des informations. Voici ce que nous allons faire : Nous créerons une application disposant de plusieurs clés et allons en personnaliser une, puis nous ajouterons un fichier en utilisant les mécanismes de sécurité disponibles.

Mais avant toute chose, nous allons nous assurer que le tag se prête bien à ces manipulations et pour cela, nous allons être relativement radicaux puisque nous allons tout simplement formater le tag en supprimant l'intégralité du contenu qu'il possède éventuellement. Nous ne reprendrons pas ici le code nécessaire à l'initialisation et la connexion au tag qui, là encore, est le premier tag détecté (**tags[0]**). Comme précédemment, à votre charge de tester les valeurs de retour des fonctions (**OPERATION_OK**) et de réagir en conséquence.

Pour procéder au formatage, nous devons nous authentifier auprès du tag avec la clé maîtresse PICC :

```
uint8_t picc_key_data_null[8] = {
  0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00 };

MifareDESFireKey piccnullkey =
  mifare_desfire_des_key_new(picc_key_data_null);

mifare_desfire_authenticate(tags[0], 0, piccnullkey);
```

Notez que nous devons créer la clé à partir d'une série de valeurs et la stocker dans une structure de type **MifareDESFireKey** grâce à la fonction **mifare_desfire_des_key_new()**. En fonction du type de clé, il faudra adapter la taille de la clé et la fonction utilisée :

- **mifare_desfire_des_key_new(uint8_t value[8])** : DES,
- **mifare_desfire_3des_key_new(uint8_t value[16])** : triple DES (3DES),
- **mifare_desfire_3k3des_key_new(uint8_t value[24])** : triple DES avec trois clés différentes (3K3DES),
- **mifare_desfire_aes_key_new(uint8_t value[16])** : AES.

Il peut paraître surprenant de procéder ainsi à la création d'une clé pour l'utiliser ensuite mais cette façon de procéder permet de n'avoir qu'un seul type (**MifareDESFireKey**) à gérer, quelque soit la nature de la clé.

Une fois authentifié de la sorte, il ne nous reste plus qu'à demander le formatage :

```
mifare_desfire_format_picc (tags[0]);
```

Là encore, **OPERATION_OK** nous permettra de nous assurer que l'opération a réussi et que notre tag est maintenant vierge de toutes informations. Nous pouvons alors débuter le processus de création. Rappelez-vous des paramètres de clés obtenus précédemment, **0x0F** nous indique que l'authentification avec la clé maîtresse PICC n'est pas nécessaire, ni pour lister les applications, ni pour créer/supprimer des applications.

Après formatage, la première chose à faire est exactement la même chose que pour la connexion à un nouveau tag : nous commençons donc par sélectionner l'application principale PICC :

```
mifare_desfire_select_application(tags[0],NULL);
```

Nous pouvons immédiatement enchaîner sur la création de notre application qui aura naturellement pour AID **0x000001**. Ceci fonctionne exactement de la même manière que ce que nous venons de voir avec les clés :

```
MifareDESFireAID myaid;
myaid = mifare_desfire_aid_new(0x000001);

mifare_desfire_create_application_aes(tags[0], myaid, 0x0f, 5);
```

Cette dernière ligne nécessite quelques explications. Pour créer une application, nous avons autant de fonctions qu'il existe de types de clés :

- **mifare_desfire_create_application()** : DES,
- **mifare_desfire_create_application_3k3des()** : triple DES à trois clés, notez que 3DES et 3K3DES représentent un seul et même type d'application, seule l'utilisation de la clé change (trois fois la même ou trois clés différentes)
- **mifare_desfire_create_application_aes()** : AES.

Rappelons que le type de clé est « porté » par l'application, il faudra donc, bien entendu, que la ou les clés utilisées correspondent. Les arguments passés sont : le tag, l'AID de l'application à créer, les paramètres de clés (similaires à ceux vu en début d'article, on retrouve ici notre **0x0F**) et enfin, le nombre de clés pour cette application. Nous avons arbitrairement choisi de définir 5 clés. Sur ce total, la première est la clé maîtresse pour l'application (clé 0) et les quatre autres sont dites « clés utilisateur ».

L'étape suivante consistera à changer l'une des clés (la 4), mais pour cela nous devons nous authentifier avec la clé maîtresse de l'application fraîchement créée. A ce stade, toutes les clés sont une suite de **0x00** par défaut. Nous commençons donc par créer cette clé AES ainsi que la nouvelle :

```
uint8_t key_data_null[16] = {
  0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00 };

MifareDESFireKey nullkey =
  mifare_desfire_aes_key_new(key_data_null);

uint8_t mykey123[16] = {
  0x00, 0x01, 0x02, 0x03,
  0x04, 0x05, 0x06, 0x07,
  0x08, 0x09, 0x0A, 0x0B,
  0x0C, 0x0D, 0x0E, 0x0F };

MifareDESFireKey mykey =
  mifare_desfire_aes_key_new(mykey123);
```

Nous nous authentifions après avoir sélectionné la bonne application :

```
mifare_desfire_select_application(tags[0],myaid);
mifare_desfire_authenticate(tags[0], 0, nullkey);
```

Et enfin, nous changeons la clé 4 de **nullkey** en **mykey** (oui, il faut fournir le numéro de clé concernée ET préciser celle actuellement en place) :

```
mifare_desfire_change_key(tags[0], 4, mykey, nullkey);
```

A présent, il ne nous reste plus qu'à créer un fichier. L'application étant déjà sélectionnée du fait du changement de clé, ceci se résume à :

```
mifare_desfire_create_std_data_file(
   tags[0],
   1,
   MDCM_ENCIPHERED,
   MDAR(MDAR_KEY4,MDAR_FREE,MDAR_DENY,MDAR_FREE),
   20);
```

Nous créons un fichier standard. Les arguments sont :

- **tags[0]** : le tag concerné,
- **1** : le numéro du fichier,
- **MDCM_ENCIPHERED** : le mode de communication qui peut être totalement chiffré comme ici, mais également simplement comporter un code d'authentification de message ou MAC pour *Message Authentication Code* (**MDCM_MACED**) ou encore autoriser une communication en clair (**MDCM_PLAIN**),
- **MDAR(MDAR_KEY4,MDAR_FREE,MDAR_DENY,MDAR_FREE)** : permet de définir les droits d'accès au fichier (voir ci-après),
- **20** : la taille du fichier en octets.

La partie importante est, bien entendu, l'ensemble des conditions d'accès au fichier, codées via un **uint16_t**. Fort heureusement, la libfreefare met à notre disposition (**freefare.h**) un ensemble de macros très pratique, permettant de choisir la clé qui est utilisée pour un droit spécifique :

```
#define MDAR_KEY0  0x0
#define MDAR_KEY1  0x1
#define MDAR_KEY2  0x2
#define MDAR_KEY3  0x3
#define MDAR_KEY4  0x4
#define MDAR_KEY5  0x5
#define MDAR_KEY6  0x6
#define MDAR_KEY7  0x7
#define MDAR_KEY8  0x8
#define MDAR_KEY9  0x9
#define MDAR_KEY10 0xa
#define MDAR_KEY11 0xb
#define MDAR_KEY12 0xc
#define MDAR_KEY13 0xd
#define MDAR_FREE  0xE
#define MDAR_DENY  0xF
```

Mais également une macro pour composer l'argument :

```
#define MDAR(read,write,read_write,change_access_rights) ( \
        (read << 12) | \
        (write << 8) | \
        (read_write << 4) | \
        (change_access_rights) \
        )
```

Avec **MDAR** il suffit de passer en argument « qui » peut :

- lire : ici la clé 4,
- écrire : tout le monde (libre d'accès),
- lire et écrire : personne (aucun droit),
- changer les droits d'accès (tout le monde).

Des macros existent également permettant de tester les droits sur une valeur **uint16_t** présente dans une structure de type **mifare_desfire_file_settings** obtenue via la fonction **mifare_desfire_get_file_settings()** prenant en argument le tag, le numéro du fichier et un pointeur vers la structure en question :

```
#define MDAR_READ(ar)       (((ar) >> 12) & 0x0f)
#define MDAR_WRITE(ar)      (((ar) >>  8) & 0x0f)
#define MDAR_READ_WRITE(ar) (((ar) >>  4) & 0x0f)
#define MDAR_CHANGE_AR(ar)  ((ar)         & 0x0f)
```

Pour terminer cette étape, il ne nous reste plus qu'à construire une petit chaîne de caractères de moins de 20 octets et de l'enregistrer dans le fichier :

```
const char *s= "Hello World";

if(mifare_desfire_write_data(tags[0],
1, 0, strlen(s), s) == strlen(s)) {
    printf("file write ok\n");
}
```

Les arguments donc dans l'ordre, le tag, l'offset (par rapport au début du fichier), la longueur de la chaîne à enregistrer et un pointeur sur cette dernière. Notez qu'à la différence de la plupart des autres fonctions, la valeur retournée est la taille des données enregistrées.

A présent, nous avons terminé l'opération et il ne nous reste plus qu'à nous déconnecter du tag avec **mifare_desfire_disconnect(tags[0])** sans oublier, ensuite, de libérer les ressources créées au passage avec **free(myaid)** pour l'application et **mifare_desfire_key_free()** pour les différentes clés, et terminer le programme proprement :

```
mifare_desfire_disconnect(tags[0]);
freefare_free_tags(tags);
nfc_close(pnd);
nfc_exit(context);
exit(EXIT_SUCCESS);
```

## 2.4 Lire un tag

Notre tag DESFire EV1 contient à présent une application AID **0x000001** possédant un fichier standard numéro 1 contenant la chaîne mythique **" Hello World "** lisible uniquement avec la clé AES numéro 4. Nous pouvons très simplement lire ce contenu après avoir, comme précédemment, tout initialisé et obtenu un accès au tag. Je vous ferai grâce ici des créations de clé et d'AID à partir de tableaux d'**uint8_t** et de valeur numérique puisqu'elles sont strictement identiques à ce que nous avons déjà vu. Nous commençons donc à la sélection de l'application :

```
mifare_desfire_select_application(tags[0], myaid);
```

pour enchaîner sur l'authentification :

```
mifare_desfire_authenticate_aes(tags[0], 4, mykey);
```

et enfin lire le contenu du fichier et l'afficher :

```
char buffer[20];

if(mifare_desfire_read_data(
    tags[0],    // tag
    1,          // num fichier
    0,          // offset
    20,         // taille
    buffer      // pointeur
    ) > 0) {
  printf("Data read : [%s]\n", buffer);
} else {
  printf("read error : %s\n", freefare_strerror(tags[0]));
}
```

Toutes tentatives de lecture avec des paramètres erronés échouera (mauvaise clé, authentification avec le mauvais numéro de clé, mauvais numéro de fichier, etc). Ici, nous avons présenté la méthode la plus simple et celle que l'on peut considérer comme la plus déterministe. Nous connaissions le numéro de l'application, les paramètres, le numéro du fichier, la quantité de données à lire, etc.

Il peut également être intéressant de parcourir le contenu d'un tag. Pour cela, on commencera par obtenir la liste des applications (ceci peut, selon la configuration, nécessiter une authentification avec la clé maîtresse du tag (PICC)) :

```
MifareDESFireAID *aids = NULL;
size_t aids_count;

mifare_desfire_get_application_ids(tags[0],
  &aids, &aids_count);
```

Sur la base de ces informations, nous pouvons construire une boucle **for(i=0; i<aids_count; i++) {}** et, par application, obtenir les paramètres pour les clés après sélection de chacune des AIDs :

```
mifare_desfire_select_application(tags[0],aids[i]);

uint8_t ksetting;
uint8_t maxkey;
mifare_desfire_get_key_settings(tags[0],
  &ksetting, &maxkey);
```

ou encore, tout simplement, enchaîner, sur le listing des fichiers par application :

```
uint8_t *files = NULL;
size_t files_count;

mifare_desfire_get_file_ids(tags[0], &files, &files_count);
```

Et dans une nouvelle boucle **for(j=0; j<files_count; j++) {}**, afficher les paramètres et le type de chaque fichier, obtenu via :

```
mifare_desfire_get_file_settings(
  tags[0], files[j], &fsettings);
```

Les macros suivantes, dont les noms sont assez explicites, permettent de déterminer le type de fichier :

- **MDFT_STANDARD_DATA_FILE**,
- **MDFT_BACKUP_DATA_FILE**,
- **MDFT_VALUE_FILE_WITH_BACKUP**,
- **MDFT_LINEAR_RECORD_FILE_WITH_BACKUP**,
- **MDFT_CYCLIC_RECORD_FILE_WITH_BACKUP**.

Quelque soit le type, une même structure (**freefare.h**) reposant sur des **union** nous permet d'obtenir les informations qui nous intéressent :

```
struct mifare_desfire_file_settings {
    uint8_t file_type;
    uint8_t communication_settings;
```

```
    uint16_t access_rights;
    union {
        struct {
            uint32_t file_size;
        } standard_file;
        struct {
            int32_t lower_limit;
            int32_t upper_limit;
            int32_t limited_credit_value;
            uint8_t limited_credit_enabled;
        } value_file;
        struct {
            uint32_t record_size;
            uint32_t max_number_of_records;
            uint32_t current_number_of_records;
        } linear_record_file;
    } settings;
};
```

Construire un outil d'exploration n'est donc pas très complexe si ce n'est par le fait de devoir gérer les différentes limitations imposées par la politique de sécurité du développeur du contenu du tag. Il faut, en effet, prendre en compte toutes les erreurs liées à ces mécanismes et réagir en conséquence de manière à présenter à l'utilisateur des données cohérentes. Bien entendu, dans la plupart des cas, la mise en œuvre de tags DESFire EV1 est quelque chose de relativement encadrée et on sera plus généralement dans la situation où on manipulera des tags avec un contenu connu. En dehors de quelques cas particuliers, le temps de développement d'un « explorateur » DESFire sympathique (avec GUI GTK+, QT ou EFL) sera sans doute mieux investi dans d'autres tâches.

## Conclusion

Nous arrêterons là cette mise en bouche des DESFire EV1 faute d'espace dans les pages du magazine. La richesse de cette technologie liée à la difficulté d'obtenir des informations précises sous forme de documentation constructeur (il est souvent nécessaire de reposer sur plusieurs documents de sources différentes), rend la mise en œuvre rapide un peu délicate. Nous espérons que les trois étapes décrites ici seront à même de mettre le pied à l'étrier du lecteur pour qu'il puisse poursuivre par lui-même ses expérimentations.

La libfreefare est d'une utilité évidente pour une telle tâche car, en plus de fournir une méthode d'accès très efficace pour les tags Mifare Classic et DESFire, elle est une source d'informations très riche et sûre. On saluera comme il se doit ici le travail des développeurs *Romain Tartiere* et *Romuald Conty*, à la fois tout à fait admirable, utile et pédagogique.

Le monde des tags RFID et NFC est un domaine à part entière qu'une petite série d'articles comme celle-ci ne saurait couvrir entièrement. Chaque technologie mériterait sans doute un hors-série complet s'il fallait les traiter de manière exhaustive.

Les technologies RFID et NFC sont de plus en plus présentes dans nos vies et cette « invasion » se fait de manière fort heureusement synchrone avec la facilité d'accès aux solutions permettant leurs explorations. Je pense non seulement à la disponibilité à faible coût du matériel mais également aux logiciels et bibliothèques qui, à présent, sont relativement matures.

La recherche de spécimens elle, est tout autant facilitée. Une fois ces premiers pas effectués dans le domaine, il suffira de sauter sur toutes les occasions qui se présentent grâce à quelques mots magiques « Vous connaissez notre programme de fidélité ? Vous voulez une carte ? », « Oui ! ». Carte de laverie, de collecte de points, d'accumulation de crédit pour la machine à café, d'accès à un bâtiment ou complexe sportif... sont autant de chances d'obtenir un tag d'une technologie encore non maîtrisée. Bien entendu, tantôt vous tomberez sur des systèmes quasi-préhistoriques (piste magnétique, code-barre, QRcode, etc) mais globalement l'ensemble évolue vers le RFID.

C'est aussi là l'occasion de vous assurer que vos données sont correctement protégées et, malheureusement souvent, de constater le laxisme de certaines sociétés de service qui s'abstiennent, tout bonnement, d'utiliser les systèmes de sécurité pourtant présents dans les tags, y compris pour des applications de monétique. Soyez responsable dans vos explorations et devant une énormité, essayez d'adopter une démarche pédagogique en fonction de la situation et des interlocuteurs. Souvent la société à l'origine du système sera assez peu réceptive concernant vos critiques et commentaires, mais le client lui, faisant usage du système et constatant que la sécurité promise (en particulier pour ses finances) n'est pas au rendez-vous, sera sans doute plus sensible. Bien entendu, le risque de passer pour le loup dans la bergerie et d'être accusé d'être « un méchant pirate » sera à prendre en compte dans ce genre de démarches. Les gens n'apprécient généralement pas que l'on remette en cause une sécurité qu'ils pensaient acquise et ont pour réflexe, malheureusement, d'exprimer leur contrariété sur celui qui lance l'alerte. C'est à vous d'agir en votre âme et conscience mais, dans tous les cas, les informations fournies ici n'ont en aucun cas pour objectif de vous inciter ou vous donner les moyens d'agir illégalement, mais sont purement pédagogiques. ■

# RFID : QUELQUES APPLICATIONS INTÉRESSANTES POUR ANDROID

*par Denis Bodor*

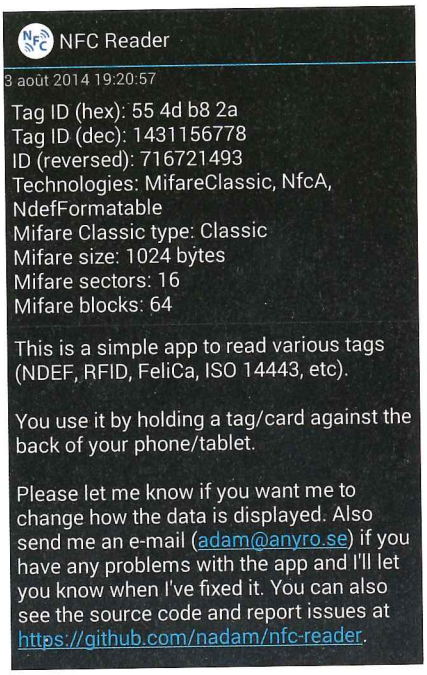
Lorsqu'on nous confie un nouveau tag, on souhaite généralement assouvir sa curiosité assez rapidement or, il arrive tantôt qu'à ce moment précis, on n'ait pas sous la main l'un de ses laptops et/ou un de ses lecteurs RFID/NFC (si si, ça arrive). Fort heureusement, on dispose presque toujours de son smartphone et, à l'aide des bonnes applications, on peut immédiatement se faire une petite idée sur notre nouvelle trouvaille.

Comme nous l'avons évoqué dans un précédent article, tous les smartphones et tablettes ne sont pas en mesure de gérer tous les types de tags. On portera alors ses préférences vers des modèles particuliers équipés de contrôleurs RFID/NFC de chez NXP, seuls capables de communiquer avec les tag Mifare Classic en plus de ceux compatibles NFC Forum. Les smartphones suivants sont connus pour ne PAS être équipés de contrôleurs NXP mais Broadcom : Google Nexus 4/5/7(2013)/10, LG G2, LG Optimus L7 II, Motorola Moto X, Samsung GalaxyAce3/Express2/Mega/Note3/S4/S5 et HTC One/M8. Ceux-ci ne peuvent lire/écrire des tags Mifare Classic S50 ou S70. Mais le plus simple est encore d'installer *NFC Smart Card Info* qui, dès son lancement, vous affichera le constructeur du contrôleur NFC de votre appareil (https://play.google.com/store/apps/details?id=com.inoapp.cardinfo).


*NFC Smart Card Info, une des rares applications à vous informer sur le type de contrôleur RFID qui équipe le smartphone*

## 1 NFC Reader

Cette application de Adam Nybäck est relativement basique mais permettra de complémenter un jeu plus complet d'outils. Sa simplicité est un avantage lorsqu'il s'agit d'en explorer les sources, car celles-ci sont disponibles sur GitHub (https://github.com/nadam/nfc-reader). L'auteur avoue qu'il s'agit d'un programme construit sur le code exemple du SDK Android mais l'objectif n'est, semble-t-il, pas de fournir la *killer app* NFC/RFID. Simplement diffuser une « démo » avec des sources que tout un chacun peut étudier.




*NFC Reader est une application relativement basique mais avec les sources à disposition*

https://play.google.com/store/apps/details?id=se.anyro.nfc_reader

## 2 NFC TagInfo by NXP & NFC TagWriter by NXP

Comment ne pas parler des deux principales applications du domaine, signées NXP ? Ces deux programmes propriétaires et en particulier *NFC TagInfo*, seront des compagnons de choix lorsque vous tomberez sur un tag mystérieux. *TagInfo* est en mesure de vous fournir toutes les précieuses indications techniques aussi bien sur le tag lui-même que sur son contenu et ce aussi bien pour des Mifare Classic, des Ultralight, des DESFire ou encore des tags ne provenant pas de chez NXP comme un EM4035 (non sans quelques

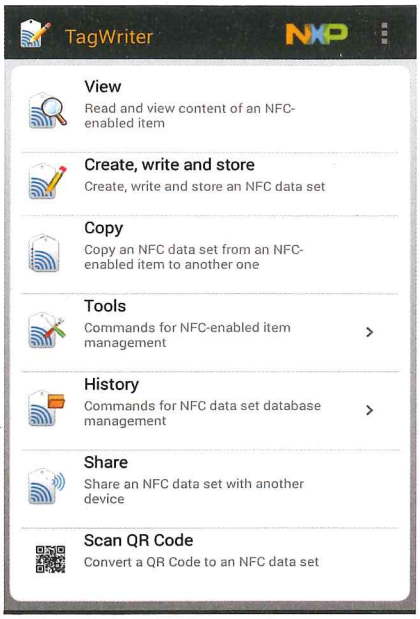
difficultés et un comportement étrange (lecture en boucle)), ou encore tout simplement des clones Mifare Classic sous licence (ou non).


*NFC TagInfo by NXP apporte un nombre d'informations incroyable. Tout ce qu'il faut pour identifier un tag.*

NFC TagWriter va un peu plus loin en proposant des outils pour lire les tags mais également pour les écrire avec des données formatées NFC (NDEF). Il s'agit davantage d'une application presque *enduser* puisque l'interface permet de rapidement enregistrer des informations comme des contacts, des URL, des mails, des SMS, etc. Concernant l'interface de l'application, on aime ou on aime pas. Les choix fait dans le design sont clairement orientés lecture/écriture avec, par exemple, l'ensemble des informations lues et écrites, conservé sous forme d'un historique réutilisable.

## 3 NFC TagInfo

Cette application, de chez *NFC Research Lab*, est tout bonnement, selon moi, la plus adaptée à l'investigation de nouveaux tags mais également pour la vérification de vos codes. Nous avons testé cette application avec des Mifare Classic, des DESFire EV1, des Ultralight




*NFC TagWriter by NXP est une application générique, orienté NFC qui peut être vue comme un outil de lecture/écriture NDEF.*

https://play.google.com/store/apps/details?id=com.nxp.taginfolite

https://play.google.com/store/apps/details?id=com.nxp.nfc.tagwriter

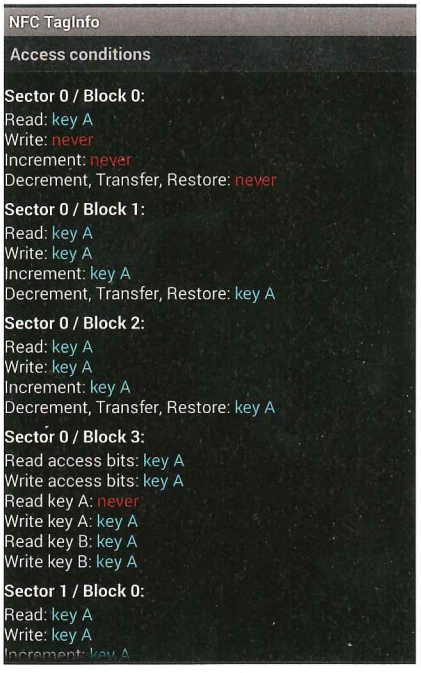
mais également avec notre seul exemplaire de EM4035. *NFC TagInfo* s'en est sorti haut la main, nous présentant non seulement les détails techniques de ce tag alors inconnu, mais également son contenu qui ne semblait aucunement protégé. C'est la seule application qui a été capable de lire ce tag provenant d'un système de restauration en *self service*.

Nous n'avons pas eu le loisir de tester tous les tags annoncés comme étant pris en charge, mais la liste est des plus intéressantes. Jugez plutôt :

- NFC Forum Type 1 ;
- NFC Forum Type 2 / MIFARE Ultralight (EV1) / NTAG ;
- NFC Forum Type 3 / FeliCa Lite ;
- my-d(TM) NFC / my-d(TM) move ;
- MIFARE Classic ;
- MIFARE DESFire ;
- MIFARE DESFire EV1 ;
- FeliCa ;
- ISO/IEC 15693 ;
- ePassport avec extraction de photo.

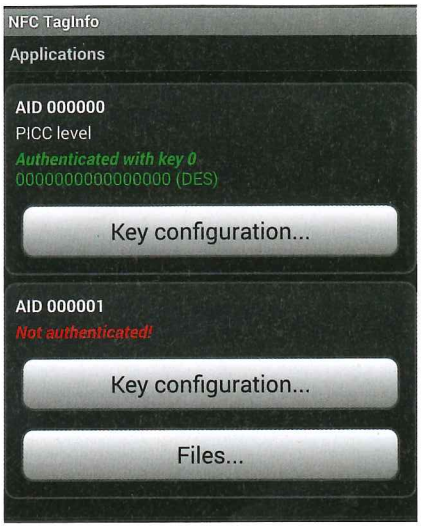

*L'écran principal de NFC TagInfo, juste après la lecture d'une Mifare Classic S50*

*NFC TagInfo* sera clairement votre associé indispensable en explorant les technologies RFID/NFC d'un point de vue de développeur. Sa simplicité d'utilisation et la présentation qui est faite, à la fois des données et des



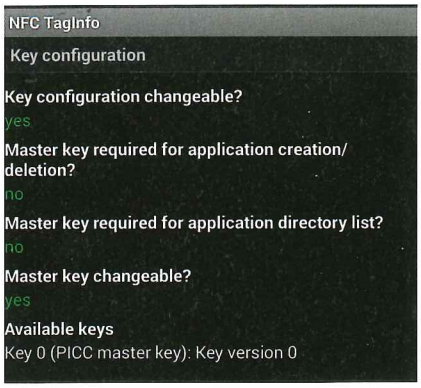

*L'une des fonctions très intéressantes de cette application est l'affichage clair des conditions d'accès de chaque bloc Mifare Classic*

paramètres, vous permettra de valider les changements faits par vos programmes et d'éluder les doutes sur l'usage des différentes configurations.




*NFC TagInfo est également capable de manipuler les informations de DESFire avec ici les différentes applications détectées*

On ne peut regretter qu'une seule chose : la non disponibilité des sources, même si l'application est gratuite et exempte de publicité. Michael Roland, l'homme qui se cache derrière *NFC Research Lab*, spécialiste en système embarqué et doctorant à l'université de Linz, met toutefois à disposition un tracking de bugs permettant de remonter d'éventuels problèmes.




*Les paramètres de clés aussi bien pour le tag DESFire lui-même que pour les applications sont présentés sur simple demande*

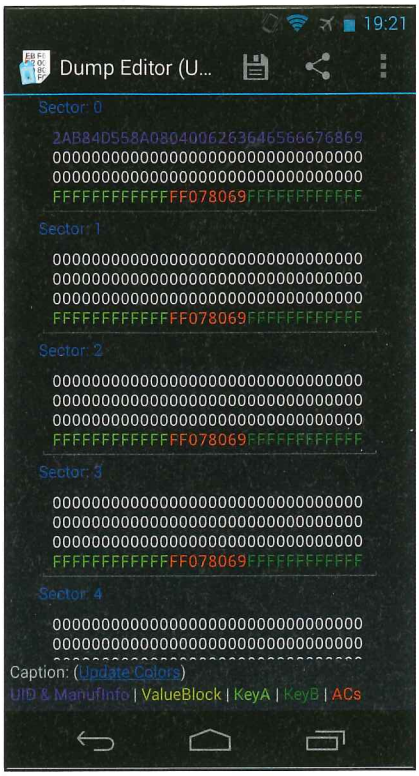
https://play.google.com/store/apps/details?id=at.mroland.android.apps.nfctaginfo

## 4 Mifare Classic Tool - MCT

Et voici incontestablement notre application préférée, créée par IKARUS Projects. Il s'agit d'une application open source (https://github.com/ikarus23/MifareClassicTool) absolument indispensable si vous souhaitez vous pencher sur les Mifare Classic. C'est le seul type de tag supporté (à condition que votre matériel les supporte) mais quel support ! Il s'agit d'une véritable boîte à outils permettant la lecture mais également l'écriture de tags, ainsi que la copie.


*L'écran d'accueil de MCT présente un accès direct à toutes les fonctionnalités. Impressionnant !*

L'un des principaux avantages de cette application tient dans la gestion des clés. Contrairement aux autres citées ici, MCT permet de stocker toute une collection de clés pour ainsi facilement lire et écrire n'importe quel Mifare Classic. Vous pourrez également organiser vos *dump* et les transférer sur PC pour une analyse plus poussée puisque tout cela (liste de clés incluse) est placé sous forme de fichiers sur la SD du smartphone.




*L'affichage du contenu hexa d'un tag est soigné, avec mise en évidence des blocs spéciaux*

https://play.google.com/store/apps/details?id=de.syss.MifareClassicTool

## Conclusion

Bien que nous n'ayons pas abordé le développement Android dans le cadre de ce dossier RFID/NFC, faute de place et de temps, l'existence d'application open source, et de MCT en particulier, offre une voie toute tracée permettant de débuter. Si, en revanche, ce type de développement ne vous intéressent pas, cet ensemble d'applications devrait faire office d'outil de pointe pour les premières phases d'exploration sur le terrain. De quoi patienter avant de vous précipiter chez vous pour en apprendre un peu plus sur ce nouveau tag qu'on vous aura confié, pensant faire de vous un client dévoué, docile et naïvement confiant dans la sécurité entourant ces points de fidélités transitant magiquement sur bout de plastique... ■

SEPT. / OCT. / NOV. 2014 N°12

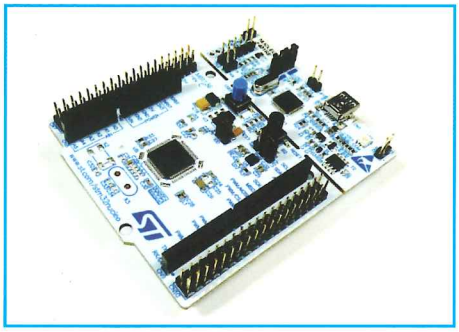
# Open Silicium

MAGAZINE
INFORMATIQUE
OPEN SOURCE
EMBARQUÉ
INDUSTRIEL ET R&D

LE MAGAZINE 100 % TECHNIQUE, 100 % PRATIQUE, 100 % EMBARQUÉ !

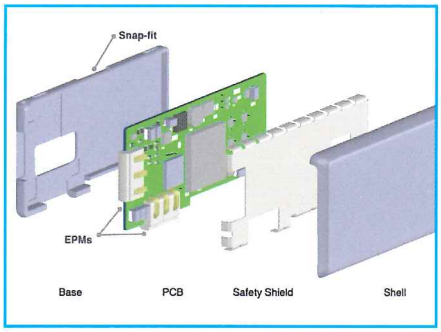
## DÉVELOPPEMENT / WEB

Les plateformes compatibles mbed ou comment mettre un IDE dans un navigateur Web p.04



## ANDROID / BRIQUES

Projet Ara : les débuts d'un smartphone modulaire, composable et adaptable à chacun p.52




## ÉNERGIE / MOBILE

Applications connectées en 3G : pourquoi la ressource radio impacte tant votre batterie ? p.46

## PHOTO / 3D

MicMac ou comment reconstruire des structures tridimensionnelles à partir de simples photographies p.59


## GOOGLE / API

Collecte et traitement des statistiques d'un site Web en Python avec l'API Analytics p.39

## BOOT / STM32

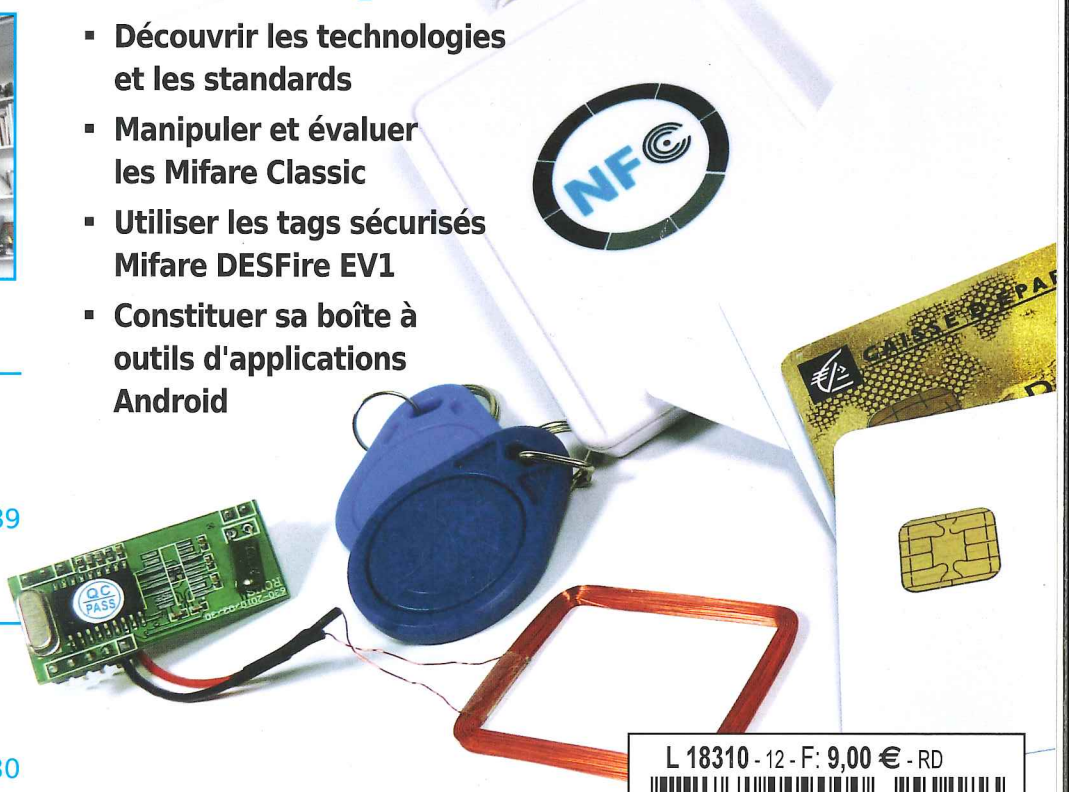
Utilisation du bootloader DFU avec les plateformes STM32F4-DISCOVERY et STM32F429I-DISCO p.80

## RFID / NFC

**Cartes, tags, étiquettes, capsules... Elles sont partout autour de nous !**

# PRENEZ EN MAIN LES TECHNOLOGIES RFID/NFC p.09

- **Découvrir les technologies et les standards**
- **Manipuler et évaluer les Mifare Classic**
- **Utiliser les tags sécurisés Mifare DESFire EV1**
- **Constituer sa boîte à outils d'applications Android**




L 18310 - 12 - F: 9,00 € - RD

France Métro : 9 € / Belgique - Luxembourg : 9,50 € / Suisse : 14 CHF / DOM : 9,90 € / CAN : 15,50 $CA / N. CAL/S : 1200 CFP / POL/S : 1300 CFP