# Stockage d'information sur un périphérique non sécurisé

Vincent Jantet, Philippe Pucheral, Luc Bouganim

Stage du 15 Mai au 29 Juillet 2006

**Résumé**

Ce document expose quelques solutions pour protéger des données dans une mémoire non sécurisée, par l'utilisation d'un composant matériel sécurisé. En effet, protéger matériellement une mémoire n'est pas forcement possible, principalement pour des raisons économiques. C'est donc une protection cryptographique qui sera effectuée par le composant sécurisé. Les protections cryptographiques sont sensibles à quatre attaques. Les trois premières, qui sont l'examen de données chiffrées, la corruption et la substitution, ont déjà des solutions efficaces, qui ne seront que peu détaillées ici. Les solutions, proposées par l'état de l'art, contre la dernière attaque, qu'est le rejeu, demandent de lourds calculs supplémentaires, et utilisent beaucoup de mémoire. C'est sur cette quatrième attaque que nous porterons notre attention, détaillant quelques solutions innovantes.

## 1 Introduction

L'INRIA (Institut National de Recherche en Informatique et Automatique) a récemment ouvert le projet SMIS (Secured and Mobile Information Systems) qui a deux domaines de spécialisations plus ou moins corrélés. L'équipe SMIS travaille d'un coté sur la gestion de données embarquées dans des calculateurs ultralégers, notamment les cartes à puce. D'un autre coté, elle s'intéresse à la protection de la confidentialité des données, en définissant de nouveaux modèles de contrôle d'accès et de chiffrement des données, et en utilisant des composants matériels sécurisés. Ce projet est en accord avec l'ALDS (Association Locale de Développement Sanitaire) et Uni-Medecine, pour concevoir un système de gestion de dossiers médicaux informatisés, plus souple et mieux sécurisé que le système actuel. L'objectif est de centraliser sur un unique serveur les données médicales des patients de différents hôpitaux.

La solution proposée s'articule autour d'un nouveau composant matériel appelé SSMSC (Smart Secure Mass Storage Card), combinant la sécurité intrinsèque d'une carte à puce à une grande capacité de stockage persistant (de type FLASH). Chaque patient et chaque médecin seraient donc en possession d'une SSMSC qui leur permettrait de synchroniser leurs mémoires FLASH avec les données du serveur les concernant, et de travailler sur cette copie locale de l'information, jusqu'à la prochaine synchronisation. C'est la puce sécurisée qui assurerait le contrôle d'accès, permettant par exemple à un patient de consul-

ter ses données personnelles, ou à un médecin de modifier la liste des prescriptions de ses patients. La mémoire de grande capacité n'a cependant pas les propriétés de sécurité de la puce. Il est donc tout à fait judicieux de considérer qu'un attaquant est capable de lire son contenu et de le modifier. Un attaquant pourrait même remplacer cette mémoire par un calculateur corrompu, qui génèrerait des réponses aux requêtes de la puce, pour tenter de la tromper. Il apparaît donc indispensable de protéger les informations sur la mémoire FLASH, ainsi que sur le serveur central lui-même.

L'approche adoptée est celle de la dernière partie de la thèse de N. ANCIAUX. Nous considèrerons donc une architecture composée de trois éléments, différenciés par leurs niveaux de sécurité, et leurs droits sur les données.

- Le SOE (Secure Operating Environment) est le seul élément de confiance. Il est capable d'effectuer des traitements de manière sécurisée sur des données invisibles d'un attaquant. Il a tous les droits sur l'information, et c'est lui qui se charge de la politique d'accès aux données. C'est-à-dire que c'est lui qui permet à l'utilisateur d'accéder aux données le concernant, lui interdisant l'accès aux autres données.
- À l'inverse, l'UOE (Untrusted Operating Environment) est potentiellement la cible de toutes les attaques. Il ne peut garantir aucune sécurité, ni aux données qu'il conserve, ni aux traitements qu'il effectue. Ses droits sur l'information sont les même que ceux d'un attaquant, à savoir aucun. C'est cependant lui qui conserve l'information utile. En l'absence de protection matérielle de cette UOE, c'est une protection cryptographique qui assurera la sécurité et la confidentialité des données.
- Enfin, le RT (Rendering Terminal) est le dispositif de saisie et d'affichage de l'utilisateur. Il n'est pas considéré comme sûr, et ne doit pas avoir accès à des résultats intermédiaires compromettants. Il a les mêmes droits que l'utilisateur, et ne travaille que sur les données autorisées.

Dans le cadre du SSMSC, c'est la puce qui joue le rôle du SOE. Dotée d'une quantité raisonnable de RAM et de mémoire persistante, elle se comporte comme un ordinateur complet si elle est correctement connectée à une alimentation électrique. En l'absence de garantie quant aux traitements faits par l'UOE, nous avons pris la décision dans les SSMSC de les réduire à ceux d'une simple mémoire. C'est la mémoire FLASH qui joue le rôle de l'UOE, ne pouvant répondre qu'aux simples requêtes Lire(Adresse) et Ecrire(Adresse,Donnée). Tous les traitements sont donc effectués par la puce, qui doit en plus vérifier le bon fonctionnement de l'UOE. Le mécanisme cryptographique utilisé doit également être optimisé pour une utilisation sur des bases de données, c'est-à-dire avoir une granularité fine. Typiquement, la base de données ne doit pas être chiffrée d'un seul bloc, mais chaque champ doit être chiffré indépendamment des autres. Cela permet de ne pas avoir à déchiffrer un gros bloc d'information si seule une petite partie de celui-ci nous intéresse. Pour finir, le SSMSC est inséré dans un ordinateur ou un PDA, qui joue le rôle du RT, permettant aux utilisateurs de visualiser les réponses de leurs requêtes à la mémoire.

La protection cryptographique est le point sensible du système. Il doit être suf-

fisamment puissant pour assurer un haut niveau de sécurité, et suffisamment simple pour ne pas engendrer un surcoût rédhibitoire. Il doit résister aux quatre attaques que sont l'examen des données chiffrées, la corruption, la substitution et le rejeu.

- L'examen de données chiffrées permet à un attaquant de déduire de l'information du texte chiffré, sans avoir à le déchiffrer. L'algorithme de chiffrement doit donc être suffisamment opaque pour empêcher cette attaque.
- La corruption est la modification aléatoire de l'information chiffrée par l'attaquant. L'intégrité de l'information doit donc pouvoir être vérifiée.
- La substitution consiste à remplacer une partie de l'information par une autre. Les différentes parties de l'information doivent donc être identifiables.
- Enfin, le rejeu est la réutilisation d'une ancienne information, à la place de celle présente en mémoire. Vérifier la fraîcheur d'une donnée peut être fait avec un mécanisme de numéros de version associé à chaque information. Mais cette association est elle-même une information confidentielle devant être conservée et protégée.

La fraîcheur est certainement la propriété la plus difficile à vérifier, et les solutions proposées ne sont pour l'instant que des ébauches à approfondir.

Ce document s'organise autours de deux grands axes. Nous détaillerons dans un premier temps, une conception possible pour la SSMSC, avec une description plus précise des quatre attaques et de leurs solutions actuelles. Nous discuterons dans un deuxième temps, de quelques approches plus ou moins innovantes pour pallier au problème de la conservation des versions.

# 2 Contexte et formulation du problème

Le composant matériel sécurisé, appelé SSMSC, n'est pas encore commercialisé. Actuellement en phase de test, le prototype peut être amené à changer légèrement. Malgré tout, nous détaillerons dans cette partie une des architectures probables, afin de bien cerner les différents problèmes de sécurité. Nous étudierons ensuite les quatre attaques possibles pour corrompre le système, ainsi que leurs solutions les plus courantes, basées sur l'état de l'art. Pour finir, nous nous attarderons sur l'une des difficultés majeures, la conservation des numéros de version de manière efficace et condensée. Ces numéros de version sont utiles pour se protéger contre le rejeu, l'attaque la plus difficile à empêcher.

## 2.1 La SSMSC

La taille réduite des puces électroniques en fait l'acteur principal de la sécurité moderne. C'est par exemple une puce électronique qui assure la sécurité dans les téléphones portables. En effet, les téléphones fonctionnent en équipe avec une carte SIM. A l'initialisation, le téléphone demande un code PIN, qui sera vérifié par la puce. S'il n'est pas correct, la puce se bloque pour protéger les informations qu'elle contient. Le téléphone pourrait tenter d'en faire autant, en demandant son propre code avant d'effectuer certaines actions. Mais le code étant connu du téléphone, et sa vérification effectuée par le téléphone lui même, cette sécurité serait inefficace contre un attaquant motivé. Il lui suffirait d'ouvrir le téléphone pour aller y lire directement son contenu, y changer le mot de passe ou même modifier le protocole de vérification, pour qu'il renvoie toujours "valide". Actuellement, les té-

léphones se bloquent en l'absence de carte SIM valide, mais ils sont facilement déblocables, et ne protègent donc pas efficacement les données qu'ils contiennent. Seule la carte SIM, insérée dans le téléphone, est à même de protéger efficacement des données. Sa taille réduite et sa fabrication avec des matériaux photosensibles, en interdit l'ouverture, la lecture et la modification. Enfin, les puces n'ayant pas d'alimentation autonome, ni même de dispositif d'affichage (ce qui pourrait bientôt changer), elles nécessitent d'être connectées pour pouvoir fonctionner. La SSMSC serait donc dotée d'un connecteur USB2 pour pouvoir être insérée dans un terminal (le RT), qui se chargerait de ces tâches-là. Ce terminal n'est pas considéré comme de confiance par la puce, Il a les mêmes droits que l'utilisateur. Il est donc important que ce terminal ne stocke pas de résultats temporaires susceptibles de dévoiler des informations normalement bloquées.

Les puces électroniques sont des miniaturisations de circuit électriques, composées principalement de transistors, de condensateurs et de résistances. Gravées dans une plaque de silicium, les puces sont de plus en plus courantes. L'utilisation qui nous intéresse est celle des puces embarquées, type cartes de payement ou cartes SIM. La puce de la SSMSC est un véritable ordinateur, composée d'un processeur 32 bits fonctionnant à 50 MHz, et d'une mémoire interne de 256 Ko de EEPROM et de 510 Ko de ROM. La puce est également directement connectée à une mémoire flash plus volumineuse, actuellement de 128 Mo. Cependant, cette mémoire FLASH n'est pas aussi résistante aux attaques que la puce, un attaquant peut en lire le contenu et le modifier. On peut même imaginer qu'il débranche cette mémoire pour la remplacer par un calculateur. Il obtiendrait alors la liste des requêtes de la puce à la mémoire, et un historique des modifications apportées à la mémoire. Pour se prémunir de ces différentes attaques à la mémoire FLASH, il faut mettre en place une protection cryptographique assurant la confidentialité des données écrites, et l'intégrité des données lues. C'est-à-dire qu'il faut s'assurer qu'un attaquant ayant pris le contrôle de la mémoire n'est pas capable, d'une part, de lire son contenu et celui des nouvelles informations en provenance de la puce, et d'autre part, de tromper la puce en lui fournissant des valeurs erronées comme résultats de ses requêtes.

### 2.2 Les attaques possibles

Le but est donc de concevoir un dispositif sécurisé de transport de bases de données, capable de répondre rapidement à des requêtes et de mettre à jours son contenu. Mais la mémoire FLASH de la SSMSC n'a pas le niveau de sécurité demandé, et un attaquant à toute liberté pour agir dessus. Il peut lire son contenu sans passer par la puce, et même le modifier. Nous considèrerons donc cet UOE comme totalement corrompu, et pouvant ne pas suivre le protocole prévu. Dans ce cadre, un attaquant a quatre attaques à sa disposition : l'examen de données chiffrées, la corruption, la substitution et le rejeu. Après les avoir décrits plus en détails, nous commenterons les mécanismes couramment mis en place pour lutter contre, et nous discuterons du surcoût induit par chacun d'entre eux.

L'examen de données chiffrées est une attaque permettant de déduire de l'information du texte chiffré, sans pour autant connaître la clef utilisée. C'est une attaque dangereuse pour la confidentialité dés le

moment où l'on considère que l'attaquant à un accès en lecture à la mémoire.

En théorie, l'algorithme de chiffrement par bloc, ou algorithme de cryptage, utilisé n'a pas vraiment d'importance. Ils ont tous à peu près le même fonctionnement. Mais l'algorithme choisi doit tout de même vérifier les propriétés de sécurité attendue. Il ne servirait pas à grand-chose de développer un protocole infaillible, s'il se base sur un algorithme de chiffrement obsolète.

L'algorithme couramment choisi dans les implémentations est le standard de chiffrement avancé AES (pour Advanced Encryption Standard). Il est issu d'un concours, lancé en 1997, suite à la découverte des faiblesses de son ancêtre le DES. Adopté comme standard par le NIST (National Institute of Standards and Technology) en 2000, il peut être considéré comme sûr, mais il n'est pas exclus que sa relative simplicité recèle une faille à découvrir. L'AES n'a pour l'instant pas été cassé et la recherche exhaustive demeure la seule solution.

L'AES est un algorithme de chiffrement par bloc. Il prend en entrée un bloc de 128 bits qu'il remplace de façon déterministe et réversible, par un nouveau bloc de la même taille. La fonction bijective utilisée est basée sur une clef de longueur quelconque, généralement entre 128 et 256 bits, qui est utilisée à la fois lors du chiffrement et du déchiffrement. Deux blocs de données identiques chiffrés avec la même clef donneront donc deux blocs chiffrés identiques. Pour les messages de plus d'un bloc, il existe plusieurs modes opératoires différents que nous présenterons par la suite.

Le mode ECP (pour Electronic Code Book) est le mode le plus naïf, il consiste à découper les messages en bloc, chacun chiffré indépendamment des autres. Les multiples occurrences d'un bloc au sein d'un long message sont donc directement retrouvées dans le message chiffré. Un attaquant peut soit étudier les redondances et les nombres d'occurrences, soit utiliser ses connaissances d'une partie du message (comme la structure, ou l'en-tête), pour construire le dictionnaire de la fonction bijective. Sans connaître la clef de chiffrement utilisée, il est capable de déduire une quantité impressionnante d'informations. Le chiffrement utilisé doit être le plus opaque possible.

Une fois la confidentialité des données assurée, un attaquant peut encore tenter de corrompre les informations que contient la mémoire. Dès le moment où il a un accès en écriture à cette mémoire, il peut modifier aléatoirement les données chiffrées. Les données sont déchiffrées par la puce, et l'attaquant ne connaissant pas la clef utilisée, il ne peut pas prévoir le résultat du déchiffrement sur les valeurs aléatoire qu'il a écrit. Il peut cependant re-modifier le contenu de la mémoire jusqu'à ce que le résultat lui convienne. Si c'est le solde de son compte bancaire, il a une forte probabilité en modifiant au hasard de tomber sur un résultat positif, et très élevé. S'il n'y a pas de mécanisme de protection, la puce ne peut pas s'en rendre compte, et elle appliquera consciencieusement ses protocoles avec cette nouvelle valeur. Il est donc indispensable de mettre en place un dispositif vérifiant l'intégrité des données, c'est-à-dire vérifiant que la puce en est bien l'auteur.

Si modifier au hasard une donnée ne fonctionne pas, l'attaquant peut encore faire des substitutions dans la mémoire. C'est-à-dire qu'il peut remplacer une information par une autre, elle-même présente dans la mémoire. Si l'attaquant a accès à la mémoire à la fois en lecture et en écriture, il peut rem-

placer son solde par celui d'une autre personne de la base de données. Ce solde ayant été correctement chiffré par la puce, un mécanisme anti-corruption basé sur une vérification de l'auteur n'est pas suffisant. Il faut en plus que la puce puisse se rendre compte qu'une information a été déplacée, et qu'elle n'est plus à sa place.

La dernière attaque est sûrement la plus fréquemment utilisée, car la plus difficile à empêcher. Elle consiste à remplacer des informations de la mémoire par ces mêmes informations, mais à une date plus ancienne. Ce n'est bien sûr possible que si l'attaquant connaît une ancienne version de l'information, c'est-à-dire qu'il a déjà eu un accès en lecture à la mémoire par le passé. Il peut alors utiliser la copie qu'il en a faite pour remplacer les valeurs actuelles par leurs anciennes valeurs. Cette attaque est difficile à détecter, la puce recevant des informations qui étaient valables par le passé, continuerait à les considérer valables actuellement si un mécanisme de fraîcheur, permettant de remarquer qu'un message est périmé, n'était pas mis en place.

### 2.3 Les solutions actuelles

Les failles de sécurité que l'on a rencontrées sont depuis longtemps connues des cryptologues, qui nous proposent donc de bonnes solutions pour chaque problème. Nous allons comprendre les protocoles que nous propose l'état de l'art, et nous verrons comment ils peuvent être adaptés au cadre qui nous occupe, à savoir le stockage en mémoire non sécurisée.

Pour pallier aux problèmes d'opacité du mode opératoire trivial qu'est l'ECB, on préfèrera employer le chiffrement AES en mode CBC (pour Cipher Block Chaining), ou en mode CTR (pour CounTeR). Dans ces modes, le bloc de texte clair est initialement combiné par un XOR à une valeur quelconque, mais à chaque fois différente et connu lors du déchiffrement. Cette étape totalement réversible à pour effet de faire perdre au texte clair les particularités statistiques qui réapparaissaient dans le texte chiffré. La valeur choisie n'a pas à rester secrète, elle est d'ailleurs écrite en clair à coté du mot de passe chiffré dans les systèmes Unix. En mode CTR, c'est un compteur, incrémenté à chaque changement de bloc, qui est combiné au texte en clair, et en mode CBC, c'est la valeur chiffrée du bloc précédent qui joue ce rôle. Un vecteur d'initialisation (IV) est utilisé comme valeur de départ du compteur CTR, ou pour chiffrer le premier bloc en CBC. Celui-ci peut être connu de tous, mais doit idéalement n'être utilisé que pour un unique message.

Il existe d'autres modes opératoires qui peuvent également être utilisés. Ceux ici présentés, ont tout de même l'avantage de permettre un accès aléatoire aux données. C'est-à-dire que l'on peut déchiffrer une partie seulement de l'information chiffrée, sans perdre de temps à analyser le reste de l'information.

Contre la corruption, la solution courante consiste à donner à l'information une structure particulière telle qu'une valeur aléatoire n'ait que peu de chance d'avoir. On peut par exemple rajouter derrière l'information une constante prédéfinie, ou rendre l'information symétrique en la dupliquant. Une fois le tout chiffré en bloc, cette structure particulière ne se retrouve pas, et l'altération aléatoire de l'information devient impraticable. Du point de vu cryptographique cependant, donner une telle struc-

ture à l'information est une aberration. En effet, on se rappellera des précédents algorithmes de chiffrement, tel Enigma, qui ont pu être cryptanalysés à cause d'une redondance d'informations en en-tête de chaque message. On préfèrera donc associer à l'information un code détecteur d'erreur, ou une empreinte par une fonction de hachage. Le supplément de bits ajouté à l'information ressemble ainsi à une suite aléatoire et ne permet pas de déduction sur les données.

On a, à partir d'ici, l'assurance que les données renvoyées par le serveur sont bien des données écrites par la puce. Mais un serveur malhonnête pourrait vouloir inverser certaines informations de la base de données. Ce problème n'est en fait pas le plus complexe, en partie parce que certaines des solutions précédentes permettent de l'éviter. Le mécanisme d'opacification nécessite une valeur quelconque mais unique comme vecteur d'initialisation. On remarquera alors que si cet IV est directement dépendant de l'adresse de l'information, il n'est plus possible de déplacer des données, et de les interchanger. En effet, si un attaquant permute certaines données chiffrées, elles ne seront pas déchiffrées en utilisant le bon IV et n'auront donc pas, en clair, une valeur prévisible. Or ce sont sur les particularités du texte clair que se base le détecteur de corruption. Du fait du caractère aléatoire apparent du texte clair, celui-ci ne passera pas les tests contre la corruption.

Pour le problème du rejeu, il faut pouvoir vérifier que l'information que nous envoie la mémoire est bien la dernière version. On peut pour cela donner à chaque version différente de l'information, un numéro, qui peut être la valeur d'un compteur, ou bien choisi aléatoirement. Ce numéro de version, indissociable de l'information, car chiffré en même temps, permet de différencier les mises à jour entre elles. Deux versions n'ayant pas le même numéro, il suffit à la puce de mémoriser de façon sécurisée le numéro de la version la plus récente, pour pouvoir vérifier que les données envoyées par la mémoire sont bien à jour.

Nous avons mis en place ces différents mécanismes dans des protocoles d'écriture et de lecture en mémoire. Le premier protocole est plus adapté aux informations courtes, faisant quatre octets. Il consiste dans un premier temps, à dupliquer l'information pour lui donner une symétrie, utile contre la corruption. Le tout est ensuite chiffré par l'algorithme AES avec comme IV, la concaténation de l'adresse de l'information, et de sa version. Utiliser l'adresse dans l'IV permet à la fois de rendre l'information opaque, et de détecter les éventuelles substitutions, et utiliser la version interdit le rejeu.

Le deuxième protocole s'applique à des données longues, d'une centaine d'octets. L'information est initialement chiffrée par AES en mode CBC avec l'adresse comme IV, ce qui garantit l'opacité. Ce résultat est stocké tel quel en mémoire, ce qui permet, grâce aux propriétés du mode CBC, de ne déchiffrer que les parties désirées de cette grande information. Pour garantir les autres propriétés de sécurité, l'information chiffrée est également mixée à son adresse et à sa version, puis hachée par un algorithme à clef secrète, et mémorisée à part. L'utilisation de l'adresse et de la version interdit la substitution et certifie la fraîcheur, quant au hachage à clef secrète, il protège contre la corruption.

### 2.4 Le problème de la conservation des versions

Pour nous protéger contre les quatre types d'attaque, nous avons des protocoles fonctionnels, mais ils nécessitent tous que la puce connaisse, avant d'accéder à la mémoire, le numéro de version de l'information qu'elle cherche. Si ce numéro est stocké dans l'UOE, il devient lui-même une information qui doit être protégée contre toutes les attaques. Il doit en particulier disposer de son numéro de version pour prouver sa propre fraîcheur. Pour sortir de ce cycle sans fin, on peut mémoriser ces numéros de version directement dans la mémoire sécurisée de la puce. Chaque information est chiffrée dans la mémoire de l'UOE, et pour déchiffrer une des informations, la puce utilise le numéro de version qui lui est associé, puisque qu'elle le connaît à l'avance. C'est la méthode la plus simple, cependant, chaque information chiffrée nécessite un numéro de version, et comme nous avons choisi une granularité fine des informations, il peut potentiellement y avoir une grande quantité de numéros de version. Cette solution n'est pas exploitable en pratique, en raison de la faible quantité de mémoire stable sécurisée dans le SOE, comparé à celle de l'UOE. Il faut donc d'autres méthodes pour mémoriser, de façon sécurisée, un grand nombre de numéros de version. Le paragraphe suivant nous donnera quelques exemples de protocoles, mais leurs utilisations ont un coût non nul. Appliquer une protection cryptographique demande un surcoût de travail pour la puce, ce qui rallonge encore les temps de réponse. Sur les premières implémentations du Système de Gestion de Base de Données (SGBD), les temps de réponse étaient parfois en dehors des limites du timeout, et il ne prenait pas en compte la sécurité. Il est donc important de surveiller de près la charge de travail demandé à la puce.

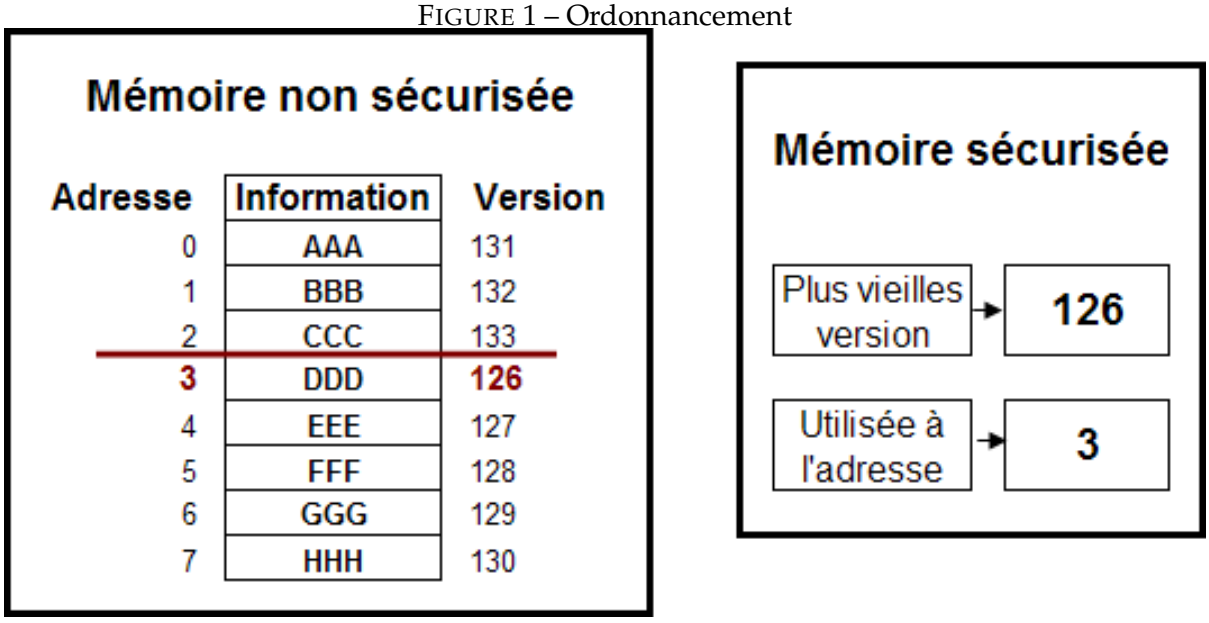
## 3 Conservation des numéros de version

Ce paragraphe explique les quatre ébauches de solutions, ainsi que leurs variantes, qui ont été proposées afin de conserver en mémoire, de façon sécurisée, l'association entre les informations et leurs numéros de version. La première approche repose sur un ordonnancement des numéros de version, pour simplifier la fonction de passage de l'adresse physique au numéro de version associé. La deuxième considère toute une plage de numéros de version, et ne mémorise que leurs états, c'est-à-dire s'ils ont été utilisés ou non. La troisième méthode mélange arbre d'indexation et arbre de fraîcheur, réduisant ainsi le surcoût de la vérification de la fraîcheur. La dernière solution enfin, n'utilise pas de mémoire sécurisée, mais repose sur un lien secret entre une information et l'adresse où est conservé son numéro de version.

### 3.1 Ordonnancement des N° V

Les numéros de version sont choisis par le système, on peut donc s'arranger pour qu'ils soient dans un certain ordre. Cet ordre étant connu à l'avance, il n'a pas à être mémorisé dans la mémoire de la puce. Concrètement, pour des informations qui sont écrites à des adresses consécutives de la mémoire, on utilise des numéros de version qui se suivent. Il n'est alors plus nécessaire de conserver la version de chaque information, mais uniquement la version de l'une d'entre elles, les autres pouvant facilement être recalculées le moment voulu. Pour vérifier la fraîcheur de l'information écrite à

FIGURE 1 – Ordonnancement




*Contenu des mémoires à un instant donné.*

une certaine adresse de la mémoire, il faut connaître à l'avance le numéro de version qui a été utilisé pour la chiffrer. Celui-ci peut être calculé à partir de la seule connaissance du numéro de version utilisé à la première adresse de la mémoire. Si l'information à l'adresse $0$ est chiffrée avec le numéro de version $V$, alors l'information à l'adresse $i$ est chiffrée avec l'adresse $V + i$. Cet ordonnancement particulier est facile à mettre en œuvre à la première écriture sur le disque, et il permet de vérifier l'intégrité des informations à coût constant.

Les difficultés relatives à cette méthode apparaissent lorsque les informations sont mises à jour. En effet, une information modifiée reste écrite au même emplacement sur le disque, mais ne doit pas être chiffrée avec le même numéro de version. Or si l'on utilise un autre numéro de version, celui-ci ne respectera pas l'ordre voulu. Il faut donc, pour modifier une seule information, en rafraîchir beaucoup d'autres, c'est-à-dire modifier leurs numéros de version, pour conserver l'ordre voulu. Ce rafraîchissement est coûteux puisqu'il implique des lectures en mémoire, des déchiffrements, des re-chiffrements en utilisant un nouveau numéro de version, et des écritures mémoire. On peut ne pas actualiser toute la mémoire à chaque mise à jour, mais seulement en moyenne la moitié. Pour cela on imagine la mémoire comme circulaire, et on mémorise à la fois le numéro de version de l'information la plus ancienne, et son adresse sur le disque. Initialement, cette information est à l'adresse $0$, et les informations qui suivent dans l'ordre chronologique, qui est également l'ordre des numéros de version. Lors de la mise à jour de l'information la plus ancienne, celle-ci devient l'information la plus récente, et c'est l'information suivante, à l'adresse 1, qui devient l'information la plus ancienne. Si l'information à modifier n'est pas la plus ancienne,

il faut préalablement rafraîchir toutes les informations antérieures, dans l'ordre chronologique, pour enfin pouvoir modifier l'information voulue. Pour une mémoire de taille $n$, si l'information la plus ancienne, inscrite à l'adresse $i_0$, est chiffrée à la version $V(i_0)$, alors la version $V(i)$ de l'information inscrite à l'adresse quelconque $i$, est déterminée par la formule : $V(i) = V(i_0) + (i - i_0)modn$. On a ainsi bien toutes les versions qui sont supérieures à la version la plus ancienne, et qui sont ordonnées dans la mémoire.

Le coût de la modification d'une information est donc variable en fonction de son âge. Dans le cas où l'information est la plus ancienne, on peut ne modifier qu'elle et garder tout de même l'ordonnancement. Dans le cas où l'information est la plus récente, la totalité de la mémoire devra être rafraîchie. Le coût d'une modification est donc en moyenne celui de $n/2$ mises à jour. vCette valeur moyenne est calculée dans le cadre de modifications aléatoires, mais il est envisageable que les données permettent des modifications séquentielles. Le coût d'une mise à jour pourrait alors s'approcher du coût constant.

Une autre difficulté apparaît lorsque les informations sont modifiées deux fois de suite, puisqu'il faut alors rafraîchir toute la mémoire. Cette méthode implique donc que toutes les informations de la mémoire aient la même fréquence de mise à jour. Si une information est fréquemment modifiée, elle oblige tout le reste de la mémoire à l'être également.

Ces difficultés peuvent être atténuées par l'utilisation de plusieurs mémoires. Cette méthode étant appliquée indépendamment sur chacune d'elles. Il faut alors stocker plusieurs numéros de version et d'adresse dans la mémoire sécurisée de la puce, mais chaque mémoire devenant plus petite, le coût de modification est réduit significativement. De plus, s'il est possible de déterminer à l'avance la fréquence de mise à jour d'une information, alors on peut regrouper les informations rarement modifiées dans une même mémoire.

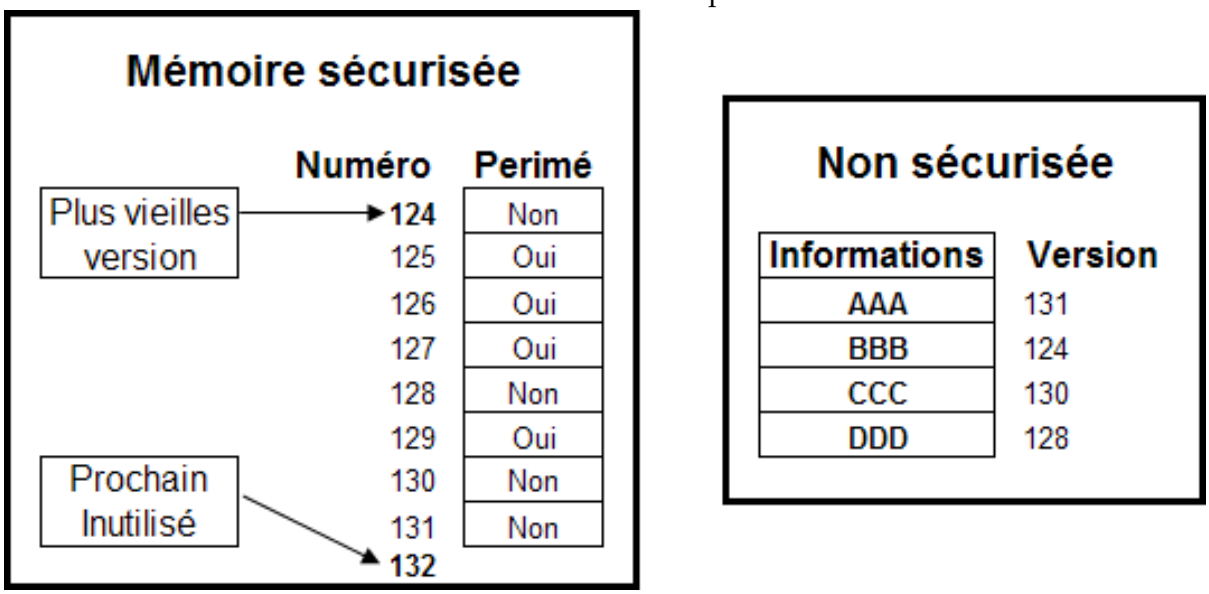
Cette méthode d'ordonnancement est donc très efficace en lecture, mais à un coût linéaire pour les écritures. Elle reste cependant bien adaptée à des données statiques ou séquentielles, puisque elle n'utilise que peu de la mémoire sécurisée du SOE.

### 3.2 Bitmap

Au lieu de mémoriser les numéros de versions qui ont été utilisés, une seconde approche consiste à mémoriser l'état d'utilisation de chaque numéro de version possible. Pour une mémoire de taille $2^n$, et des indicateurs de fraîcheur sur $k$ bits, conserver les $n$ numéros de version en cours d'utilisation consommerait $2^n * k$ bits de mémoire sécurisée. Dans le cadre de la bitmap, c'est l'état de chacun des $2^k$ numéros de version possible qui sont conservés. L'état, qui peut prendre les valeurs "non utilisé", "utilisé" et "périmé", ne consomme que peu de bits dans la mémoire sécurisée. De plus, il est possible d'utiliser un ramasse-miettes comme nous le verrons par la suite, qui permet de grouper les versions dans le même état, afin de réduire le coût de stockage.

L'information stockée dans la mémoire flash ne doit pas être chiffrée comme précédemment. En effet, l'utilisation de l'indicateur de fraîcheur dans l'IV lors du chiffrement, nécessite que celui-ci soit connu à l'avance lors du déchiffrement. Dans le cas

FIGURE 2 – Bitmap




*Contenu des mémoires à un instant donné.*

de la bitmap, il n'est pas connu à l'avance, et doit donc être concaténé à l'information, et le tout doit être chiffré en utilisant un IV basé seulement sur l'adresse. Le numéro de version n'est pas connu à l'avance, mais lors du déchiffrement, il est possible de vérifier qu'il est valide, c'est-à-dire qu'il a bien été utilisé pour chiffrer une information et qu'il n'est pas périmé. Son état, conservé en mémoire dans le SOE, doit donc être "utilisé". Si ce n'est pas le cas, alors l'information est corrompue. Soit elle à été modifiée aléatoirement, soit elle a été rejouée.

Pour modifier une information, il faut préalablement la déchiffrer pour récupérer l'indicateur de fraîcheur qui avait été utilisé la fois précédente. Ce numéro doit être dans l'état "utilisé" pour garantir l'intégrité de l'information. Puisque cette information doit être modifiée, son numéro de version doit passer dans l'état "périmé". Cette action est définitive, et permet de se protéger du rejeu. Si un attaquant rejoue une vielle information, il devra également rejouer l'indicateur de fraîcheur associé, qui sera noté comme "périmé" dans la mémoire du SOE.

La nouvelle information doit ensuite être chiffrée avec un nouveau numéro de version, qui n'a encore jamais été utilisé. Il suffit dont de choisir un remplaçant parmi les indicateurs de fraîcheur qui sont dans l'état "non utilisé", et de le noter comme "utilisé". Les changements d'état ne sont pas réversibles. Un numéro de version ne devant pas être utilisé plusieurs fois, il ne repassera pas dans l'état "non utilisé" après son utilisation.

Cette approche protège également contre la corruption, mais est un peu moins sécuritaire que la méthode de base, qui consiste à ajouter son empreinte à la donnée. Elle nécessite donc des numéros de version sur plus de bits que l'empreinte. En effet, l'indicateur de fraîcheur n'est pas connu

à l'avance, seule sa validité est vérifiée. Contrairement à une méthode classique où lors du déchiffrement, une unique empreinte est possible pour une certaine donnée, on a ici $2^n$ numéros de version dans l'état "utilisé". Quel que soit celui qui est lu dans l'information, le système considèrera que l'information est fraîche. Pour des numéros de version écrits sur k bits, il y a donc une probabilité de $2^(n-k)$ qu'une modification aléatoire génère un numéro de version dans l'état "utilisé". Cette probabilité reste cependant négligeable si $k$ est bien supérieur à $n$.

#### 3.2.1 Ramasse-miettes

Pour conserver en mémoire le tableau des états des $2^k$ indicateurs de fraîcheur possibles, commençons par lui donner quelques propriétés supplémentaires. Lorsque le système a besoin d'un nouveau numéro de version, il n'est pas obligé de le choisir au hasard parmi les numéros inutilisés. S'il les consomme dans l'ordre, alors mémoriser la frontière permet de connaître tous les indicateurs de fraîcheur inutilisés.

En plus de consommer les versions dans l'ordre, l'idéal serait de les rendre périmées dans l'ordre. Bien sûr ce n'est pas possible car ce sont les versions associées aux informations que l'on souhaite modifier qui doivent être rendues inutilisables. Cependant, la péremption d'un indicateur de fraîcheur étant irréversible, la proportion de numéros périmés ne fera qu'augmenter. Principalement le début du tableau, c'est-à-dire la partie la plus ancienne, ne contiendra pratiquement que des versions périmées. Si l'on force à rendre périmées, les quelques versions du début de tableau qui ne le sont pas encore, alors il sera inutile de conserver l'état de chacune. On définit donc une deuxième frontière en dessous de laquelle toutes les versions sont périmées.

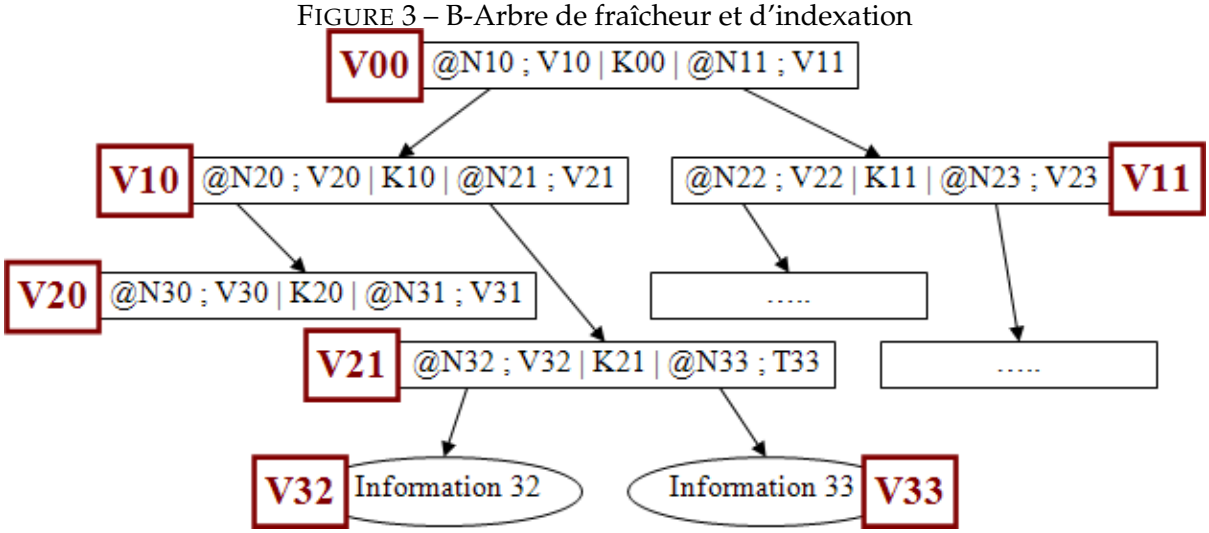
Pour forcer à périmer un certain numéro de version, il faut mettre à jour l'information qu'il protège, et donc connaître l'adresse sur le disque de cette donnée. Heureusement, cette information n'est ni confidentielle, ni pertinente, et peut être mémorisée dans l'UOE sans aucune protection. Seule la partie centrale du tableau, contenant une alternance de version dans l'état "utilisé" et dans l'état "périmé", doit être conservé dans la mémoire sécurisée. De plus, seuls deux états y sont possibles, ce qui permet de coder chaque état sur un seul bit.

Cette méthode est donc rapide en lecture, et en écriture. Les coûts d'accès et de modification sont tout les deux constants. La mémoire sécurisée nécessaire est plus importante que pour les méthodes précédentes, mais la solution est cependant tout à fait envisageable. Il reste à concevoir un ramasse-miettes efficace, qui n'entraîne pas un surcoût trop important.

### 3.3 Arbre de fraîcheur et d'indexation

Cette troisième méthode est beaucoup plus coûteuse que les précédentes, cependant, elle permet d'effectuer plusieurs tâches en même temps. Elle consiste à utiliser un même arbre, pour indexer les informations de la base de données, et les protéger contre le rejeu. Chaque nœud de l'arbre est chiffré à l'aide d'un numéro de version. Ce numéro de version n'est pas conservé en mémoire sécurisé, mais est contenu dans le nœud père. Un nœud contient donc à la fois les adresses de ses fils, et les numéros de versions qui ont été utilisés pour les chiffrer. Seule la version de la racine de l'arbre doit être conservée dans la mémoire sécurisée du SOE.

FIGURE 3 – B-Arbre de fraîcheur et d'indexation




*Chaque nœud à une version notée en gras et chaque pointeur est accompagné de la version de l'objet pointé.*

Vérifier la fraîcheur d'une information, contenue dans une feuille, nécessite que le système parcoure tout l'arbre. Il doit sur son parcours déchiffrer les nœuds rencontrés, vérifier leur fraîcheur grâce aux informations lues dans le nœud précédent, jusqu'à pouvoir enfin lire le numéro de version utilisé pour protéger l'information, et vérifier sa fraîcheur.

Le coût d'une simple vérification de fraîcheur est donc proportionnel à la hauteur de l'arbre, c'est-à-dire logarithmique en la taille de la mémoire. Cependant le parcours d'un arbre est indispensable pour trouver l'adresse de l'information que l'on cherche. De plus, l'arbre d'indexation est lui-même une information à stocker dans la mémoire de l'UOE, et donc à protéger contre le rejeu. Cette méthode a l'avantage de protéger chaque nœud, pour un coût faible. Les versions des nœuds traversés sont connues de la puce au moment où elle en a besoin.

Lors des modifications, un nouveau numéro de version est utilisé pour chiffrer l'information. Il doit donc être inscrit dans le nœud père, au coté du pointeur vers le fils. Le nœud père est donc modifié et doit lui-même être re-chiffré sous une nouvelle version. Ainsi de suite, tout les ancêtres de l'information qui est modifiée doivent être mis à jour, jusqu'à la racine. La nouvelle version de la racine est ensuite conservée dans la mémoire sécurisée. Encore une fois, le coût d'une modification est logarithmique en la taille de la mémoire, mais il se confond avec le coût de la réorganisation de l'arbre d'indexation. En effet, si l'information est modifiée, elle ne doit sûrement pas être indexée de la même façon, et l'arbre doit être modifié. Le surcoût induit est donc assez faible finalement.

Le véritable problème de cette méthode survient lorsqu'il y a plusieurs arbres d'indexation sur les mêmes informations. En effet, les tuples d'une base de données

sont fréquemment indexés sur plusieurs champs. Il y a par exemple deux pointeurs vers chaque feuille dans les implémentations de B-arbres. Ainsi, puisque chaque pointeur est suivi de la version de l'information vers laquelle il pointe, ils doivent tous être mis à jour lorsque ce numéro évolue. Multiplier les index accélère les recherches, mais ralentit également les mises à jour.

Bien qu'ayant un coût logarithmique, cette méthode est efficace car elle s'intègre bien au contexte des bases de données. Un mécanisme d'indexation étant obligatoire dans une base de données, la protection contre le rejeu n'induit que quelques contraintes supplémentaires.

### 3.4 Adresse secrète

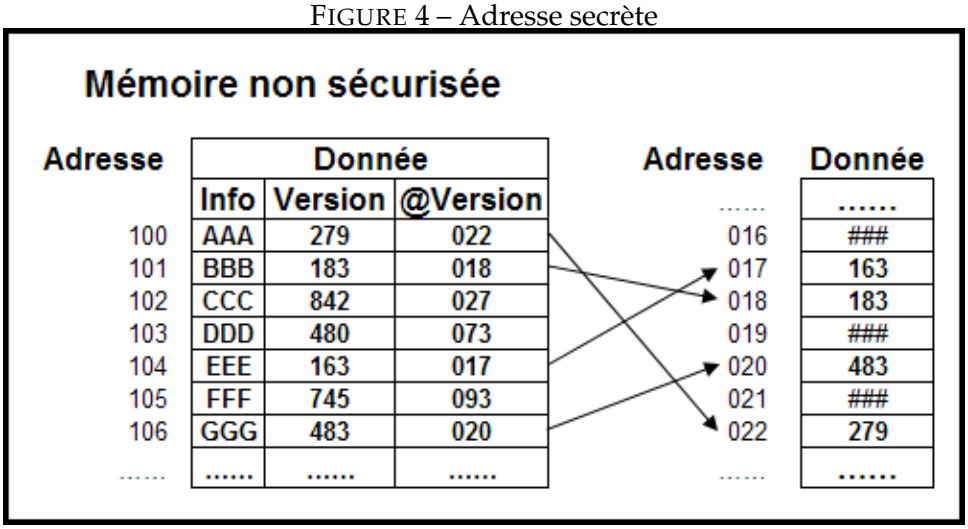
Ce quatrième protocole est innovant puisque qu'il n'utilise pas du tout de mémoire sécurisée, tout en restant efficace en lecture et écriture. Autant l'information chiffrée que le numéro de version utilisé pour la protéger sont enregistrés dans l'UOE. Ils sont par contre inscrits à des adresses différentes, et c'est l'association entre l'adresse de l'information et celle de sa version, qui est protégée et doit rester secrète. L'adresse où est stockée la version, est chiffrée au côté de l'information. Cette méthode présente cependant des failles importantes : un attaquant pouvant par exemple découvrir des associations secrètes en espionnant les différents accès faits par la puce dans mémoire et mettre ainsi le protocole en défaut.

Pour que cette méthode fonctionne, le protocole de chiffrement doit être légèrement adapté. En effet, il était expliqué dans le paragraphe précédent que l'adresse et la version des informations devaient être connues à l'avance pour pouvoir déchiffrer. C'est le cas si ces paramètres sont utilisés dans l'IV de l'algorithme de chiffrement. Dans la méthode ici présentée, seule l'adresse de l'information est connue par le système, elle sera donc le seul composant de l'IV, empêchant ainsi la substitution. Lorsque l'on souhaite lire une information, il faut déchiffrer la mémoire à l'adresse où elle est conservée. Le résultat est composé d'une part d'une adresse mémoire et d'autre part de l'information encore protégée contre la corruption et le rejeu, par l'utilisation d'un code correcteur d'erreur et d'un numéro de version. Pour continuer le déchiffrement, il faut récupérer la version de l'information, qui est écrite en clair dans la mémoire, à l'adresse qui vient d'être déchiffrée. Avec ce numéro de version, il est possible d'isoler l'information, et de vérifier par le code correcteur d'erreur qu'elle n'a été ni corrompue, ni substituée, ni rejouée.

Pour modifier une information de la mémoire, la méthode est similaire. Il faut déchiffrer l'ancienne information pour connaître l'adresse où est stocké le numéro de version. Ce numéro doit être détruit pour éviter le rejeu, une valeur aléatoire sera donc écrite en remplacement. Ensuite, un nouvel indicateur de fraîcheur est choisi, et est écrit à une adresse libre de la mémoire. L'information est ensuite protégée par un code correcteur d'erreur, et mélangée avec l'indicateur de fraîcheur puis concaténée à l'adresse secrète de numéro de version. Le tout est finalement chiffré de manière opaque, en utilisant l'adresse de l'information comme IV.

Si un attaquant souhaite rejouer une information, il doit également réécrire l'ancien numéro de version à l'adresse où il était

FIGURE 4 – Adresse secrète




*Contenu des mémoires à un instant donné.*

à l'époque. Cette adresse étant secrète, il n'a pas la possibilité de ne rejouer qu'une information précise. Il a par contre toujours la possibilité de rejouer la totalité de la mémoire. Cela pourrait être évité en gardant dans la mémoire sécurisée du SOE, quelques bribes de la mémoire de l'UOE. Si la totalité de la mémoire est rejouée, les bribes choisies ne coïncideront plus, et la fraudes sera découverte.

Il est plus difficile de remédier à la seconde faille, qui consiste pour l'attaquant à retrouver l'adresse secrète de numéros de version, en étudiant les accès mémoire qui sont faits. En effet, pour chaque lecture d'une information, il faut lire le numéro de version à l'adresse secrète. Si l'attaquant récupère la liste des accès mémoire, il peut donc faire le lien entre l'information et l'adresse de sa version, ce qui lui permet par la suite de rejouer les deux informations ensemble. La solution proposée contre cette attaque n'est pas logicielle, mais matérielle. Pour récupérer la liste des accès mémoire, un attaquant doit corrompre l'UOE. Dans le cadre de la SSMSC, l'UOE étant composé d'une mémoire flash seule, l'attaquant doit la remplacer par un calculateur qui conserverait un historique. Ce remplacement nécessite de dérober la clef et de l'ouvrir. Ce type d'intrusion est détectable.

La méthode proposée présente également un autre avantage, celui d'être adaptable au cadre multi-utilisateur. En effet, la difficulté rencontrée dans les exploitations d'une mémoire, par plusieurs utilisateurs, réside dans la synchronisation entre les utilisateurs des différentes informations nécessaires à la vérification de l'intégrité de la mémoire. Dans notre protocole, il n'y a pas de mémoire sécurisée dans le SOE. L'intégrité de la mémoire de l'UOE est vérifiable sans information supplémentaire, chaque utilisateur peut donc y lire et y écrire (à condition tout de même qu'il soit en possession de la clef de chiffrement).

Cette approche multi-utilisateur est donc à la fois facile à mettre en œuvre, et rapide à l'exécution. Les coûts de lecture et d'écriture sont tout les deux constants, et ne requièrent

que deux accès mémoire. La méthode présente tout de même des failles dangereuses, difficilement évitables.

# 4 Discussions

## 4.1 Arbre de Merkle

Contre le rejeu, la méthode décrite et utilisé précédemment consiste à mélanger l'information à un numéro de version, et à conserver de diverses façons ce numéro de version dans la mémoire sécurisée du SOE. Une méthode couramment utilisée consiste à remplacer le numéro de version par une empreinte de l'information. L'information est mémorisée dans l'UOE, et son empreinte dans le SOE. Cette façon de faire protège à la fois contre la corruption, la substitution et le rejeu. En effet, toute modification de l'information fera qu'elle ne coïncidera plus avec son empreinte.

En raison de la granularité très fine souhaitée, il y a un grand nombre de petites informations, et donc un grand nombre d'empreintes à stocker dans la puce. Comme cela n'est pas faisable, il faut mettre en place une structure d'arbre de Merkle qui permet de réduire le nombre d'empreintes à mémoriser, au prix de quelques calculs supplémentaires. Les données sont vues comme les feuilles d'un arbre, où chaque nœud contient l'empreinte de l'ensemble de ses fils. L'empreinte de la racine est en plus conservée dans le SOE.

Lorsque l'on accède à une ou plusieurs informations de la mémoire, il faut également lire l'ensemble des nœuds qui permettent de recalculer la racine de l'arbre. Cette racine étant en sécurité dans la mémoire de la puce, il n'est pas possible de modifier les valeurs lues par la puce.

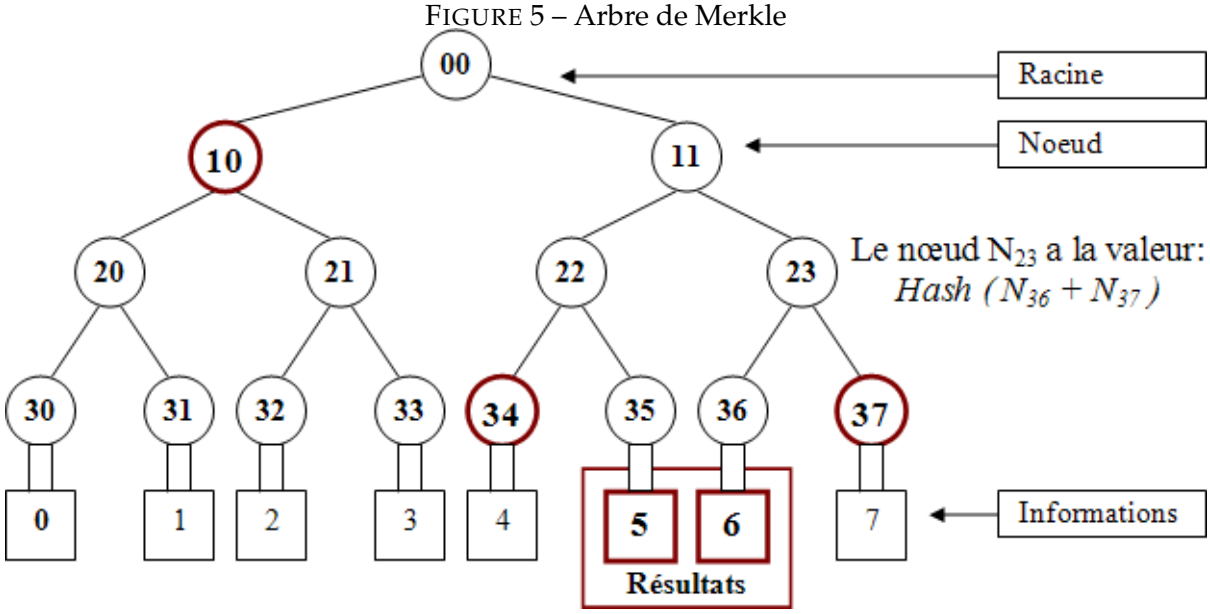
Cette méthode prend son sens lorsque l'on veut accéder à plusieurs informations, réparties sur une plage compacte des feuilles de l'arbre. Typiquement, si les données sont triées dans l'arbre, faire une sélection revient à lire toutes les feuilles d'un petit sous-arbre. L'UOE doit également renvoyer à la puce, toutes les racines des plus gros sous-arbres ne contenant pas d'information demandée. On obtient en plus l'assurance que le serveur n'a pas oublié certaines feuilles dans la plage demandée. S'il occulte une valeur dans la plage, la construction de l'arbre aboutira à une racine fausse. Ainsi, la complétude du résultat de la requête est vérifiée si l'UOE renvoie une plage de résultats légèrement plus vaste que le résultat seul. La puce se chargeant d'éliminer les deux résultats en frontière de plage, après avoir vérifié la cohérence du tout.

L'arbre de Merkle est très utilisé dans les systèmes sécurisés. Mais il nécessite le calcul de $log(n)$ fonctions de hachage qui ont un coût important. Cette solution n'est donc efficace que si elle sert à vérifier l'intégrité d'une grande plage d'informations. Pour cela les informations doivent être triées de sorte que le résultat de la requête soit un ensemble d'informations consécutives. C'est envisageable pour des sélections suivant un critère simple, mais plus du tout si l'on considère des jointures entre différentes bases de données.

# Conclusion

Les approches que nous avons détaillées, c'est-à-dire un environnement, composé d'un SOE et d'un UOE, peut être adapté à plusieurs autres situations. Par exemple celle d'un l'ordinateur client (le SOE) qui se connecte à un serveur sur Internet (l'UOE)

FIGURE 5 – Arbre de Merkle




*Pour que le SOE puisse recalculer la racine, l'UOE doit renvoyer tout les nœuds notés en gras*

pour y stocker ses données personnelles. Ou celle du processeur d'un ordinateur qui vérifie l'intégrité des données qui lui sont fournies par ses différents périphériques. Comme un processeur n'a aucune mémoire stable sécurisée, seule la méthode des adresses secrètes peut être utilisée.

## Références


- N. Anciaux. "Database Systems on Chip"
- B. Schneier. "Cryptographie appliquée - deuxième édition"
- C. Lauradoux. "La sécurité matérielle : le cas des consoles de jeux (modchip)"
- D. Clarke, G. E. Suh, B. Gassend, A. Sudan, M. V. Dijk, and S. Devadas. "Towards Constant Bandwidth Overhead Integrity Checking of Untrusted Data"
- G. E. Suh, D. Clarke, B. Gassend, M. V. Dijk, and S. Devadas. "Efficient Memory Integrity Verification and Encryption for Secure Processors"
- M. Naor, and G. N. Rothblum. "The Complexity of Online Memory Checking"
- M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. "Checking the Correctness of Memories"