Module de confiance pour externalisation de données dans le Cloud

Levent Demir

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTE UNIVERSITE GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Levent Demir

Thèse dirigée par **Dr. Vincent Roca**
et codirigée par **Dr Jean-Louis Roch** et **Jean-Michel Tenkes**

préparée au sein de l'**Équipe Privatics, Institut National de Recherche en Informatique et en Automatique**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Module de confiance pour l'externalisation de données dans le Cloud

Thèse soutenue publiquement le **7 décembre 2017**,
devant le jury composé de :

M. Pr Benjamin Nguyen
Professeur, INSA Centre Val de Loire, Examinateur et Président

M. Dr Luc Bouganim
Directeur de recherche, Inria Saclay-Ile de France, Rapporteur

M. Dr Pascal Lafourcade
Maître de conférence, Univesité de Clermont Auvergne, Rapporteur

Mme Pr Maryline Laurent
Professeur, Télécom SudParis, Examinatrice

M. Jean-Michel Tenkes
Ingénieur, INCAS IT-Sec Gardanne, Co-Directeur de thèse

M. Dr Vincent Roca
Chargé de recherche, Inria Montbonnot, Directeur de thèse

M. Dr Jean-Louis Roch
Enseignant-chercheur, Grenoble-INP/Université Grenoble-Alpes, Co-Directeur de thèse

Resumé

L'externalisation des données dans le Cloud a engendré de nouvelles problématiques de sécurité. L'enjeu est de protéger les données des utilisateurs et leur vie privée. En ce sens, deux principes ont été suivis durant cette thèse : le premier est d'avoir une confiance limitée envers l'hébergeur de données (entre autres), le deuxième est d'établir une architecture basée sur un module de confiance placé en rupture entre le poste client et le Cloud, d'où l'approche "Trust The Module,Not The Cloud".

Déléguer donc les opérations de sécurité à un module matériel dédié permet alors plusieurs bénéfices : d'abord s'affranchir d'un poste client davantage vulnérable face à des attaques internes ou externes ; ensuite limiter les composants logiciels au strict minimum afin d'avoir un meilleur contrôle du fonctionnement et enfin dédier les opérations cryptographiques à des co-processeurs spécialisés afin d'obtenir des performances élevées.

Ainsi, les travaux menés durant cette présente thèse suivent trois axes. Dans un premier axe nous avons étudié les défis d'un Cloud personnel destiné à protéger les données d'un particulier, et basé sur une carte nano-ordinateur du marché peu coûteuse. L'architecture que nous avons définie repose sur deux piliers : une gestion transparente du chiffrement grâce à l'usage d'un chiffrement par conteneur appelé Full Disk Encryption (FDE), initialement utilisé dans un contexte de protection locale (chiffrement du disque d'un ordinateur ou d'un disque dur externe) ; et une gestion transparente de la distribution grâce à l'usage du protocole iSCSI qui permet de déporter le conteneur sur le Cloud. Nous avons montré que ces deux piliers permettent de construire un service sécurisé et fonctionnellement riche grâce à l'ajout progressif de modules "sur étagère" supplémentaires.

Dans un deuxième axe, nous nous sommes intéressés au problème de performance lié à l'usage du FDE. Une étude approfondie du mode de chiffrement XTS-AES recommandé pour le FDE, du module noyau Linux *dm-crypt* et des co-processeurs cryptographiques (ne supportant pas tous le mode XTS-AES), nous ont conduit à proposer différentes optimisations dont l'approche *extReq*, qui étend les requêtes cryptographiques envoyées aux co-processeurs. Ces travaux nous ont ainsi permis de doubler les débits de chiffrement et déchiffrement.

Dans un troisième axe, afin de passer à l'échelle, nous avons utilisé un module de sécurité matériel (Hardware Secure Module ou HSM) certifié et plus puissant, dédié à la protection des

données et à la gestion des clés. Tout en capitalisant sur l'architecture initiale, l'ajout du module HSM permet alors de fournir un service de protection adapté aux besoins d'une entreprise par exemple.

Mots-clés : Cloud, XTS-AES, FDE, conteneur, co-processeurs cryptographiques, Linux kernel.

Abstract

Data outsourcing to the Cloud has led to new security threats. The main concerns of this thesis are to protect the user data and privacy. In particular, it follows two principles : to decrease the necessary amount of trust towards the Cloud, and to design an architecture based on a trusted module between the Cloud and the clients. Both principles are derived from a new design approach : "Trust The Module, Not The Cloud ".

Gathering all the cryptographic operations in a dedicated module allows several advantages : a liberation from internal and external attacks on client side ; the limitation of software to the essential needs offers a better control of the system ; using co-processors for cryptographic operations leads to higher performance.

The thesis work is structured into three main sections. In the first section, we confront challenges of a personal Cloud, designed to protect the users' data and based on a common and cheap single-board computer. The architecture relies on two main foundations : a transparent encryption scheme based on Full Disk Encryption (FDE), initially used for local encryption (e. g. hard disks), and a transparent distribution method that works through iSCSI network protocol in order to outsource containers in Cloud.

In the second section we deal with the performance issue related to FDE. By analysing the XTS-AES mode of encryption, the Linux kernel module *dm-crypt* and the cryptographic co-processors, we introduce a new approach called *extReq* which extends the cryptographic requests sent to the co-processors. This optimisation has doubled the encryption and decryption throughput.

In the final third section we establish a Cloud for enterprises based on a more powerful and certified Hardware Security Module (HSM) which is dedicated to data encryption and keys protection. Based on the TTM architecture, we added "on-the-shelf" features to provide a solution for enterprise.

Keywords : Cloud, XTS-AES, FDE, container, cryptographic co-processors, Linux kernel, HSM.

Remerciements

Je souhaite ici remercier de nombreuses personnes qui m'ont apporté aide et soutien tout au long de ce travail de recherche.

Je tiens tout d'abord à remercier Pr. Benjamin Nguyen, Dr. Luc Bouganim, Dr. Pascal Lafourcade et Pr. Maryline Laurent pour avoir accepté d'évaluer mes travaux de recherche et de faire partie de mon jury de thèse.

J'adresse des remerciements tout particuliers et chaleureux à mes encadrants de thèse, Dr. Vincent Roca et Dr. Jean-Louis Roch pour leurs conseils, mêmes contradictoires, qui m'auront été précieux durant ces années. Merci également à eux pour leur patience dont j'ai usée au moment d'écrire des articles.

Je remercie également Jean-Michel Tenkes, président de INCAS IT-Sec sans qui cette thèse CIFRE n'aurait pas eu lieu. Je retiens les nombreuses discussions et réflexions que l'on a pu avoir, même tard le soir. Les échanges permanents m'ont permis de toujours avoir une réalité industrielle en perspective et m'ont donné une vue plus globale sur mon travail.

Je tiens également à chaudement remercier les personnes qui m'ont apporté leur précieux soutien : Dr. Cédric Lauradoux, Mathieu Thierry et Djibril DIONG pour leurs conseils et les discussions que l'on a eus.

Je souhaite par ailleurs remercier tous les membres de l'équipe PRIVATICS. La bonne ambiance qui règne à l'institut de recherche m'a permis de réaliser cette thèse dans un cadre de travail très agréable. Je pense notamment aux nombreux barbecues que l'on a pu faire !

Je tiens ensuite à remercier ma famille qui m'a apporté un soutien et un réconfort permanent.

À l'ensemble de ces personnes, merci !

Pour mon épouse et ma fille, Nessibè*, née durant ma thèse.*

Table des matières

Liste des tableaux

Table des figures

Chapitre 1

Introduction

Sommaire

1.1 Motivation

Le Cloud a transformé la manière avec laquelle on travaille avec les données personnelles ou professionnelles. L'exploitation, le traitement et le stockage de ces données ont été déportés dans des services externes proposés par des fournisseurs. Ces changements ont créé un climat de doute chez les utilisateurs perdant le contrôle direct de leurs données. Ces doutes sont légitimes surtout après les révélations publiques de novembre 2013 où le journal Washington Post a révélé des accès à des données personnelles pour des ressortissants non-américains par l'Agence Nationale de Sécurité (NSA) américaine, à l'insu de ces derniers et dans des proportions importantes. Ces révélations montrent que les agences de surveillances ont accédé aux données par deux moyens [RvH14] : le premier via une procédure de coopération avec les fournisseurs de Cloud ou les compagnies de télécommunications, le deuxième à l'insu de ces mêmes entités par l'installation de backdoors (portes dérobées) permettant de déchiffrer les données en transit ou d'y accéder directement. A ce titre, citons comme programme pour le premier PRISM et MUSCULAR pour le deuxième, où les agences accédaient aux données non chiffrées transitant entre les centres de données.

Cette remise en question de la confiance accordée aux hébergeurs a permis de mettre en place des solutions de protection de données côté client [KL14, WA14]. Ainsi le client prend en charge une partie de la sécurité de ses données. Il est important de préciser que l'enjeu n'est pas simplement l'action de chiffrer et de déchiffrer les données côté client mais réside plutôt dans la gestion des clés de chiffrement [CIC13]. En d'autres termes, savoir où sont générées et stockées ces clés. En effet, si l'hébergeur détient une copie des clés ou les a générées, chiffrer les données côté client renforce la sécurité face aux attaquants extérieurs espionnant le réseau entre le client et l'hébergeur, mais cela n'empêche en rien l'accès aux données par l'hébergeur.

Ces considérations nous ont amené à travailler sur la mise en place de différents services Clouds sécurisés, allant du Cloud personnel basé sur une solution économique au Cloud Entreprise garantissant un niveau de sécurité plus élevé. L'élément de confiance dans notre architecture est un module physique placé en rupture entre le client et le Cloud qui offre un **chiffrement transparent** et **une distribution transparente** des données, le tout avec également une gestion des clés de chiffrement maîtrisée et sécurisée. Le module regroupe toutes les opérations cryptographiques, les logiciels installés se limitent aux strictes besoins de sécurité pour diminuer les risques de portes dérobées (backdoor) ou codes malicieux.

1.2 Problématique et méthodologie adoptée

Notre objectif tout au long des différentes architectures que nous proposons a été le maintien d'une simplicité et d'un système modulaire. Nous souhaitons à cet effet, mettre en place des architectures Cloud permettant de gérer les données des utilisateurs via un module de confiance placé en rupture entre le client et le Cloud. La problématique est de pouvoir stocker les données sur le Cloud, les déchiffrer dans le module et y accéder sur le client tout en gérant de manière sécurisée les clés de chiffrement.

Ainsi le *modus operandi* a été de trouver une solution de base répondant à notre besoin principal qui est le chiffrement des données dans un contexte d'externalisation, puis d'ajouter des briques à cette architecture au fur et à mesure. De ce fait, nous nous éloignons dans notre approche des différentes solutions existantes tout-en-un, à savoir regrouper (ré-implémenter) tous les services et les fonctionnalités dans une application unique.

Cette méthodologie nous permet d'avancer à petit pas, tout en garantissant une stabilité et des performances dans les différents services proposés. Nous avons appelé le principe général de notre approche **TTM : "Trust The Module, Not The Cloud"**.

1.3 Contributions et organisation du document

Il existe de nombreuses solutions de chiffrement côté client, la majorité étant des solutions logicielles intégrées au poste client. Notre première contribution est l'analyse de ces différentes solutions et en particulier les différents niveaux où ce chiffrement opère afin de nous positionner par rapport aux solutions existantes.

Notre deuxième contribution est de proposer une architecture de Cloud personnel modulaire basée sur un module de confiance en **rupture entre le client et le Cloud** regroupant toutes les opérations cryptographiques. Cette architecture repose sur une solution Full Disk Encryption (FDE) de Linux avec un conteneur déporté sur le Cloud.

Notre troisième contribution porte sur le co-processeur cryptographique AES des cartes Atmel et spécifiquement sur le mode XTS-AES. Nous avons, dans un premier temps, implémenté le mode hybride XTS-AES sur une carte comportant un co-processeur cryptographique AES avec seulement les modes communs (dont le mode ECB-AES) mais sans le mode XTS-AES. Puis, l'analyse de ces co-processeurs et du module dm-crypt de Linux, nous ont conduit à l'optimisation *extReq* qui permet un gain ($\times 2$) des débits de chiffrement et déchiffrement par le biais de requêtes étendues. Nous avons enfin discuté et donné des lignes directrices permettant d'implémenter *extReq* dès la conception du co-processeur cryptographique par le fabriquant.

Notre quatrième contribution concerne le Cloud Entreprise et l'usage d'un module de sécurité matériel (Hardware Security Module - HSM) certifié. Tout en capitalisant sur l'architecture initiale, nous avons adapté notre solution afin de passer à l'échelle. Pour cela, nous avons ajouté des services comme la gestion des clés de chiffrement basée sur une Public Key Infrastructure (PKI) et utilisé un HSM comportant des performances plus élevées.

En dernier lieu, nous proposons des mécanismes d'urgence au sein du HSM concernant les clés de chiffrement en s'appuyant sur la flexibilité du mode XTS-AES. L'officier de sécurité sera en mesure de contrôler l'arrêt ou la destruction des conteneurs (espaces de stockage).

Ce document de thèse suit les différentes contributions citées ci-dessus.

1.4 Publications

L. Demir, A. Kumar, M. Cunche, C. Lauradoux, "The Pitfalls of Hashing for Privacy", *In IEEE Communications Surveys and Tutorials*, 2017.

L. Demir, V. Roca, J-M Tenkes, "Chiffrement de la Master Key de Cryptsetup à l'aide d'une PKI : pourquoi et comment ?", *In RESSI*, 2017.

L. Demir, M. Thiery, V. Roca, J-L. Roch, J-M. Tenkes, "Improving dm-crypt performance for XTS-AES mode through extended requests : first results", *In GreHack*, 2016.

L. Demir, M. Cunche, C. Lauradoux., "Analysing the privacy policies of Wi-Fi trackers.", *In the Workshop on Physical Analytics (ACM)*, Bretton Woods, 2014.

M. Cunche, L. Demir, C. Lauradoux., "Anonymization for Small Domains : the case of MAC address.", *In Atelier sur la Protection de la Vie Privée (APVP)*, 2013.

C. Lauradoux, L. Demir, "Guesswork", *In Multi-system & Internet Security Cookbook (MISC)*, 2013.

En cours de soumission

L. Demir, M. Thiery, V. Roca, J-L. Roch, J-M. Tenkes, "Optimizing dm-crypt for XTS-AES : Getting the Best of Atmel Cryptographic Co-Processors"

Chapitre 2

Etat de l'art

Sommaire

2.1 Introduction

La protection des données dans le Cloud repose sur les primitives cryptographiques. Nous allons voir dans cette partie les différents algorithmes utilisés majoritairement dans les systèmes Cloud. Nous allons aussi décrire les propriétés attendues dans tout système telles que la confidentialité ou l'intégrité.

2.2 Les fondamentaux de la cryptographie appliquée au Cloud

Les premières techniques de chiffrement modernes étaient principalement utilisées dans le domaine militaire, les services diplomatiques et les gouvernements en général. La cryptographie était un outil pour protéger les secrets et stratégies nationaux. Avec l'évolution des techniques de cryptographie et de la puissance des appareils électroniques, la cryptographie s'est démocratisée et est utilisée dans de nombreux domaines, de la protection des transactions financières à la signature des courriels jusqu'à la protection de la donnée elle-même. La cryptographie est utilisée pour garantir des principes comme :

- **la confidentialité :** consiste à garantir que seules les personnes autorisées aient accès au message.
- **l'intégrité des donnée :** consiste à garantir que la donnée n'a pas été altérée et correspond à sa forme originelle.
- **l'authentification :** consiste à garantir l'identité prétendue d'un interlocuteur.
- **la non répudiation :** permet de garantir qu'une action ne peut être niée.

Historiquement, les premiers algorithmes de chiffrement symétrique standardisés sont apparus dans les années 1970, comme le Data Encryption Standard (DES). Puis en 1976, un nouveau concept révolutionnaire de cryptographie à clé publique et d'échange de clés a été introduit par Diffie Helmann[DH76]. En 1978, Rivest, Shamir et Adleman[RSA78] ont concrétisé dans la pratique les recherches de Diffie Helmann en proposant la première implémentation de cryptographie à clé publique, appelé RSA aujourd'hui. Ces derniers ont ajouté à cette implémentation la signature numérique qui prend la forme d'un standard en 1991 avec la norme ISO/IEC 9796. Par la suite, Koblitz et Miller ont proposé une nouvelle implémentation asymétrique basée sur les courbes elliptiques[Kob87]. Parallèlement, les algorithmes de chiffrement symétrique ont aussi évolué. En 2001, l'algorithme Advanced Encryption Standard (AES) a été choisi comme standard par le National Institute of Standards and Technology (NIST). De nombreux modes sont apparus. Un des plus récents est le mode XTS dédié au chiffrement pour les appareils de stockage de type bloc. Dans cette partie, nous présenterons d'abord les concepts du chiffrement symétrique (Section 2.2.1). Puis, nous allons décrire en détail les concepts du chiffrement asymétrique (Section 2.2.2).

2.2.1 Chiffrement symétrique

Les chiffrements symétriques répondent principalement au principe de confidentialité. Le partage d'une clé secrète entre deux entités et la présence de deux fonctions (chiffrement et déchiffrement) permettent l'échange sécurisé de données. En exemple, prenons deux utilisateurs Alice et Bob. Notons M l'espace des messages en clair, C l'espace des messages chiffrés et K l'espace

des clés de chiffrement/déchiffrement. L'algorithme de chiffrement est noté E, et l'algorithme de déchiffrement est noté D, ils sont définis comme suit :

- L'algorithme de chiffrement $E : M \times K \rightarrow C$ prend comme entrée le message en clair m, et la clé secrète k, et retourne le message chiffré c.
- L'algorithme de déchiffrement $D : C \times K \rightarrow M$ prend comme entrée le message chiffré c, et la clé secrète k, et retourne le message en clair m.

Ces deux algorithmes sont validés s'ils répondent à la propriété suivante :

$$\forall m \in M, k \in K, D(E(m, k), k) = m \tag{2.1}$$

La figure 2.1 montre l'échange d'un message entre Alice et Bob, on remarque la présence de la clé partagée k.

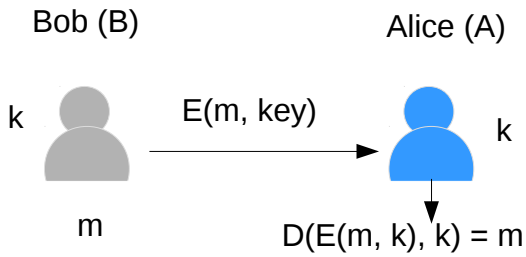


FIGURE 2.1 – Chiffrement symétrique.

Un des algorithmes de chiffrement les plus connus est celui de Vernam, proposé en 1917. Vernam prétendait que la clé de chiffrement k était une suite aléatoire de la même taille que le message m. Reprenons les mêmes utilisateurs Alice et Bob. Supposons qu'Alice souhaite envoyer un message m avec la clé partagée k, elle calcule le message chiffré tel que $c = E(m, k) = m \oplus k$. Puis, Bob utilisant la même clé k déchiffre le message chiffré reçu c tel que $m = D(c, k) = c \oplus k$.

Cet algorithme présente cependant des conditions fortes pour assurer la confidentialité, la clé doit être générée aléatoirement et être unique pour chaque message. Shannon[Sha49] a prouvé que la taille de la clé k doit être au moins égale au message m. Enfin, lors d'une communication, un canal sécurisé permanent doit être établi entre les deux interlocuteurs. Cette contrainte est clairement un frein dans la pratique, car la moitié des échanges seront gaspillés dans les échanges de clés.

Afin de contrer ces problèmes liés à l'algorithme de Vernam, de nouveaux algorithmes symétriques apparurent appelés chiffrement par bloc. Ces algorithmes chiffrent les données par bloc, à l'aide de clés de taille fixe. Ces chiffrements par bloc sont composés essentiellement de permutations. Les algorithmes les plus connus sont le Data Encryption Standard (DES), et le Advanced Encryption Standard (ce dernier est largement déployé dans les fournisseurs de Cloud).

Nous allons maintenant voir les différents modes existants pour l'algorithme de chiffrement AES, certifié par le NIST en décembre 2001. Nous allons décrire seulement les trois premiers modes parmi les cinq modes de base suivant : ECB, CBC, CTR, OFB et CFB. Ces modes ont été utilisés durant les différents travaux de cette thèse. La taille de clé peut prendre 3 valeurs : 128, 192 et 256 bits, plus la taille augmente et plus le nombre de tours dans l'algorithme augmente.

ECB-AES - Electronic Code Book

Le premier mode est appelé Electronic Codebook (ECB) est une application directe du principe de chiffrement par bloc. Tous les blocs de 16 octets d'un message m $(P_1 \dots P_n)$ sont chiffrés à l'aide de la clé E_k pour obtenir un message chiffré C $(C_1 \dots C_n)$:

$$\forall i \; tel \; que \; 1 \leq i \leq n, C_i = E(P_i, E_k) \tag{2.2}$$

Le déchiffrement suit le même principe :

$$\forall i \; tel \; que \; 1 \leq i \leq n, P_i = D(C_i, E_k) \tag{2.3}$$

Notons que ce mode a deux défauts majeurs. Le premier est que le même bloc en clair chiffré génère le même texte chiffré. Autrement dit, si un message contient le même motif répété, un adversaire sera en mesure de reconnaître la même information envoyée (canal) ou stockée (disque). Avec le mode ECB-AES, le chiffrement et le déchiffrement sont complètement parallélisables. Le deuxième concerne la propagation des erreurs. L'effet de ce mode est très limité. Si un bit est erroné dans un bloc du message chiffré, seul le bloc déchiffré correspondant sera impacté.

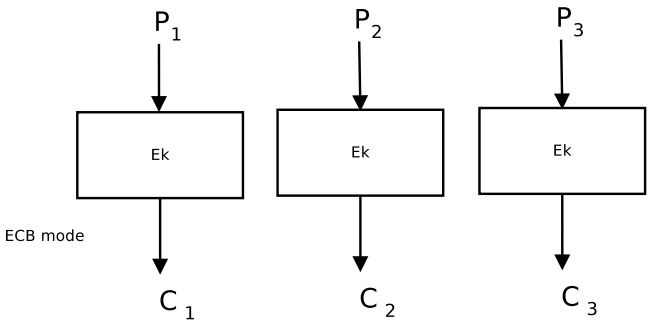


FIGURE 2.2 – Schéma du mode ECB-AES (chiffrement).

CBC-AES - Cipher Block Chaining

Le second mode, plus utilisé, est le Cipher Block Chaining (CBC). Contrairement au mode ECB, un IV est utilisé en amont du chiffrement :

$$Pour\ i = 1,\ C_1 = E(P_1 \oplus IV, E_k) \tag{2.4}$$

Les blocs suivants sont liés les uns aux autres, tous les textes en clair d'un message m $(P_i \dots P_n)$ sont *XOR*é avec le chiffré du bloc précédent C_{i-1} :

$$\forall i\ tel\ que\ 2 \leq i \leq n,\ C_i = E(P_i \oplus C_{i-1}, E_k) \tag{2.5}$$

Plusieurs blocs identiques ne généreront pas des textes chiffrés identiques. De plus, une modification dans un bloc chiffré C_i a un impact sur le texte en clair associé P_i et le suivant P_{i+1}. Cependant, un message complet chiffré deux fois avec une même clé génère deux fois le même chiffré, si le **même IV est utilisé**. Un moyen de surmonter ce défaut est d'utiliser un IV différent pour chaque message m.

Il faut savoir que le chiffrement est chaîné : le début d'un bloc est tributaire de la fin du chiffrement du bloc précédent, et donc non parallélisable. Le déchiffrement, à contrario, est parallélisable puisque tous les blocs sont disponibles dès le début de l'opération. Ce mode en particulier a été utilisé dans le chiffrement de conteneur par défaut jusqu'en 2008 dans LUKS/*dm-crypt*.

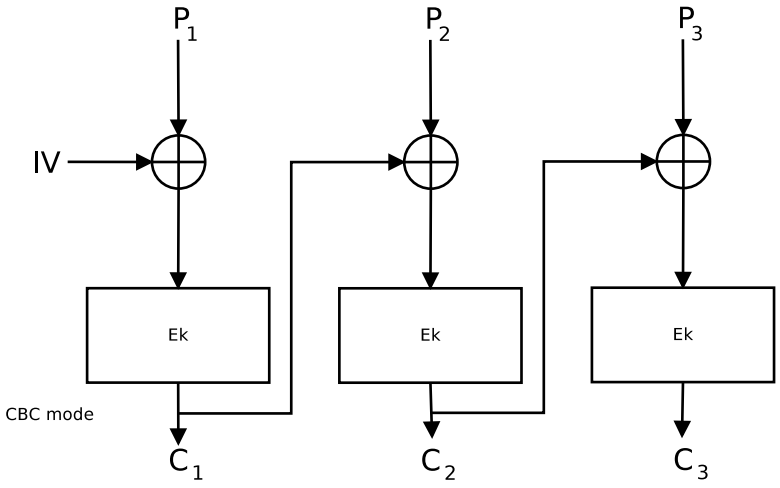


FIGURE 2.3 – Schéma du mode CBC-AES (chiffrement).

CTR-AES - Counter Mode

Le mode CTR est un mode différent des deux premiers puisqu'il est considéré comme un chiffrement de flux. Pour chaque bloc en entrée i parmi n, une clé S_i est générée en fonction de l'IV initial ($S_1 = IV$) :

$$\forall i \; tel \; que \; 2 \leq i \leq n, \; S_i = S_{i-1} + 1 \tag{2.6}$$

Le texte en clair est *XOR*é avec l'IV chiffré (avec la clé K) :

$$\forall \; i \; tel \; que \; 2 \leq i \leq n, \; C_i = E(S_i, E_k) \oplus P_i \tag{2.7}$$

L'avantage considérable de ce mode est qu'il est complètement parallélisable (voir Figure 2.4). L'accès à un bloc ne requiert pas de chiffrement/déchiffrement des précédents. L'inconvénient néanmoins concerne la séquence S_n, car toutes les valeurs utilisées doivent être uniques (voir annexe B de la section 6 de [Dwo01]).

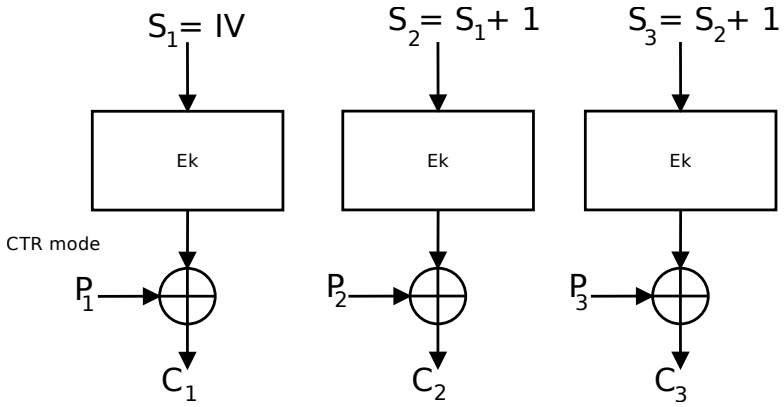


FIGURE 2.4 – Schéma du mode CTR-AES (chiffrement).

Les autres modes

Nous pouvons ajouter la présence d'autres algorithmes proposant à la fois la confidentialité et l'authentification appelés AEAD (chiffrement authentifié avec données associées). Le plus répandu est le mode Galois Counter mode (GCM). Cet algorithme propose de chiffrer les données avec le mode CTR-AES. Les données associées sont découpées en bloc de 16 octets puis chaînées à l'aide de la multiplication dans les corps de Galois. D'autres opérations sont aussi effectuées. Ce mode est parallélisable. Il est utilisé dans les chiffrements des flux de données comme IPSEc ou le chiffrement de fichiers.

Nous parlerons plus loin dans le chapitre 4 du mode XTS-AES.

2.2.2 Chiffrement asymétrique

Le chiffrement asymétrique garantit aussi la confidentialité. Contrairement aux chiffrements symétriques, les deux entités communicantes ne partagent pas de clé secrète, et le canal d'échange d'information peut ne pas être protégé. Chaque entité possède une paire de clés : *publique* et *privée*, *la clé publique* étant partagée auprès de toutes les autres entités alors que *la clé privée* doit rester secrète. Cette implémentation s'appelle Cryptographie à Clé Publique (Public Key Cryptography -PKC-). Elle est décrite dans la figure 2.5.

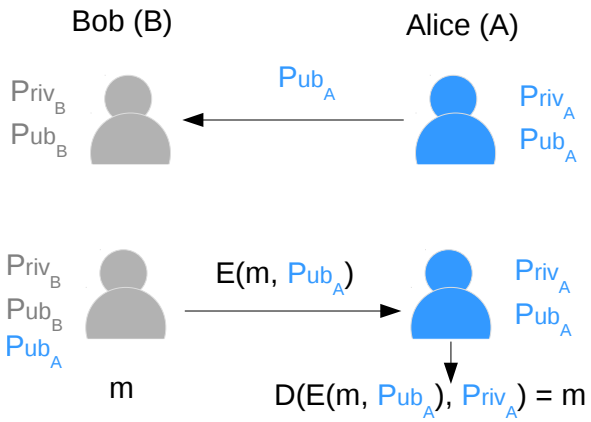


FIGURE 2.5 – Chiffrement asymétrique.

Ainsi, si Bob souhaite envoyer un message m à Alice, il récupère sa clé publique Pub_A et chiffre le message avec : $E(m, Pub_A)$. Ce message chiffré peut être envoyé via un canal non sécurisé. Lorsque Alice souhaite lire le message chiffré, elle le déchiffre avec sa clé privée $Priv_A$: $D(E(m, Pub_A), Priv_A) = m$.

La PKC peut paraître idéale, mais ce protocole comporte une faille importante quant à la vérification de la clé publique. En effet, supposons un attaquant Eve situé entre les deux utilisateurs Alice et Bob. Ce dernier pourrait récupérer la clé publique d'Alice et usurper son identité. En clair, se faire passer pour Alice. Cette attaque suit deux phases, la première (Figure 2.6) consiste à récupérer la clé publique d'Alice Pub_A et envoyer celle d'Eve Pub_E à Bob. Ainsi lors de la phase 2 (Figure 2.7) Eve sera capable de déchiffrer le message destiné à Bob avec sa clé privée $Priv_E$. De plus, afin de ne pas laisser de trace et permettre à Alice et Bob de communiquer sans se douter de l'usurpation, Eve chiffre à nouveau le message m destiné à Alice avec la vrai clé publique d'Alice. Ainsi les deux interlocuteurs ne se douteront point de l'attaque.

Afin de solutionner l'authentification des clés publiques, une infrastructure de clé publique (PKI) reposant sur une entité de confiance appelée CA (Autorité de Certification) existe. Le CA

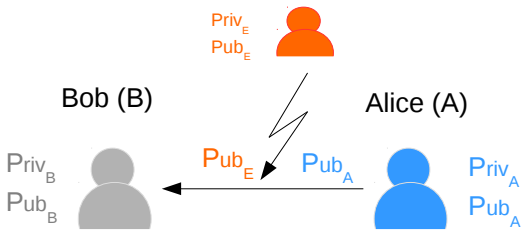


FIGURE 2.6 – Usurpation d'identité d'Alice par Eve : phase 1.

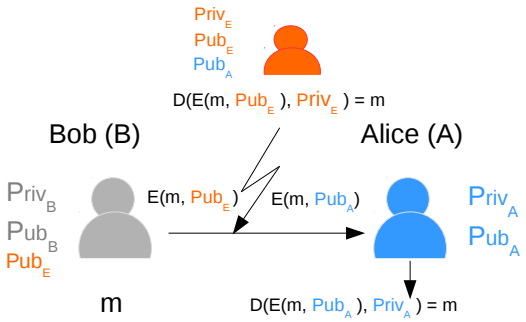


FIGURE 2.7 – Usurpation d'identité d'Alice par Eve : phase 2.

peut être interne à une entreprise ou une entité privée externe. Ce procédé repose sur la signature d'un certificat contenant la clé publique de l'utilisateur et les informations d'identification de ce dernier. Nous allons illustrer la signature à l'aide d'un échange entre Alice et Bob. Par simplification, m est considéré comme le message à authentifier, or dans la réalité la signature s'effectue sur un condensé (court) d'un message plus long (par ex. la fonction de hashage SHA-256 génère un condensé de 256 octets). La figure 2.8 illustre l'échange d'un message authentifié entre Alice et Bob. Précisément, avant d'envoyer le message, Bob génère une signature avec sa clé privée $Priv_B$ du message m. Il envoie par la suite le message m avec la signature s et sa clé publique Pub_B. Encore une fois la clé publique n'est pas transmise directement en réalité, notre schéma est simplifié. En dernier lieu Alice vérifie en recalculant la signature du message avec la clé publique de Bob. Si les deux signatures correspondent (celle envoyée par Bob et celle recalculée par Alice) alors le message m est authentifié.

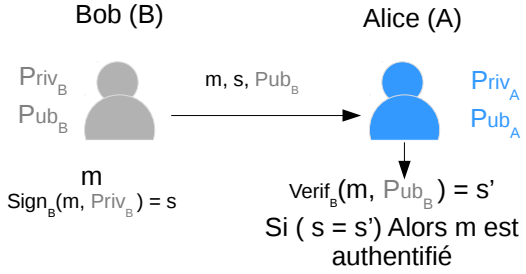


FIGURE 2.8 – Signature d'un message.

D'autres implémentations permettant l'authentification et l'échange de clés existent. Citons par exemple Kerberos, un protocole réseau composé de 3 entités : le client A, le serveur proposant un service B et le serveur (d'authentification Kerberos) T. Dans un premier temps C et B ne partagent pas de secret directement alors que chacune des deux entités partage un secret avec T, K_AT et K_BT. L'objectif premier de B est de vérifier l'identité de A ; le second objectif est l'établissement d'une clé partagée entre A et B.

Voici un résumé simplifié du protocole. A demande à T les identifiants relatifs au serveur B. T joue le rôle du centre de distribution des clés (KDC), retournant à A une clé de session chiffrée pour A et un *ticket* chiffré pour B. Le ticket, transféré par A vers B, contient la clé de session et l'identité de A. B peut ainsi authentifier A en se basant sur un message (authenticator) créé par A contenant un horodatage chiffré avec la clé de session.

Après avoir vu ces différents types de chiffrement, nous allons analyser les différents niveaux de chiffrement.

2.3 Différents niveaux de chiffrement

Nous nous focaliserons essentiellement sur le chiffrement des données de l'utilisateur. Pour y arriver nous détaillerons les éléments constitutifs des systèmes de fichier. Il est important au préalable de souligner que ce système de fichier comporte quatre catégories distinctes à protéger :

- **Le nom du fichier :** peut être sensible et il est essentiel dans certains cas de le protéger.
- **Le corps du fichier :** (ou payload) est généralement toujours chiffré.
- **Les meta-données :** représentent les informations annexes des fichiers (e.g. la dernière date d'accès, la taille) qui peuvent aussi être sensibles.
- **L'arborescence des fichiers :** peut révéler des informations. Prenons une structure de fichier connue comme une installation de système d'exploitation avec n dossiers, et m fichiers dans chaque dossier. Par analogie, un attaquant serait capable de détecter la présence de cette structure de fichier et ainsi connaître le type de contenu.

Dans cette partie, nous allons décrire des solutions réparties à 3 niveaux : le premier est basé sur le chiffrement exclusif du contenu des fichiers, le second niveau est un système de fichier chiffrant tout ou une partie des différentes catégories énoncées précédemment (noms de fichier, meta-données, structure des fichiers). Le troisième niveau, que nous avons choisi dans le cadre de cette thèse, est un chiffrement au niveau du bloc, la plus petite unité logique adressable au sein d'un disque.

2.3.1 Chiffrement au niveau du fichier

Le chiffrement de fichier est le plus ancien type de chiffrement utilisé pour protéger les données des utilisateurs. C'est pourquoi de nombreux logiciels le propose automatiquement tels les deux suites bureautiques Libre Office et Microsoft Office. Ces derniers indiquent la possibilité de chiffrer un fichier et de le protéger à l'aide d'un mot de passe. L'algorithme de chiffrement utilisé pour Microsoft Office est AES 128 bits [off16], et son mode choisi est CBC-AES.

Ces chiffrements portent uniquement sur le contenu mais les noms de fichier restent visibles. De plus, chaque fichier peut être protégé par un mot de passe, ce qui permet une gestion fine pour l'utilisateur. Cependant, la sécurité repose sur de nombreux mots de passe que l'utilisateur doit retenir. Or Adams et Sasse[AS99] ont conclu que choisir un mot de passe sécurisé est une tâche difficile pour la majorité des utilisateurs.

Enfin, les implémentations cryptographiques sont dans la majorité des cas logicielles et exécutées au niveau de l'espace utilisateur.

2.3.2 Chiffrement au niveau du système de fichiers

Ce niveau permet de chiffrer tout ou une partie des catégories citées précédemment. Les anciennes solutions *ecryptfs* et *encFs* ne protègent pas les meta-données et les tailles de fichiers par exemple. Deux audits de sécurité ont mis en exergue de nombreuses incohérences dans leur implémentation. Depuis 2015, de nouvelles applications tentent, en se basant ou non sur ces deux logiciels, de combler les lacunes et proposent une protection plus complète. Parmi elles, CryFs se pose comme une alternative fiable adaptée à Dropbox. Les fichiers sont découpés en morceaux appelés *chunk* et synchronisés avec le Cloud. Une analyse détaillée de ces solutions est présentée dans la section 3.4.1.

2.3.3 Chiffrement de bas-niveau : bloc

Les disques physiques comme les disques durs internes ou externes, les clés USB, les CD-ROM ont pour unité de stockage de base le bloc. Cette notion de bloc peut cependant être source de confusion car il faut en distinguer deux types :

- **le bloc physique :** correspondant à un secteur sur le disque. La taille des secteurs, dans la majorité des systèmes, est de 512 octets.
- **le bloc logique :** est une abstraction du système de fichiers et correspond à la plus petite unité logique adressable. Les valeurs communes sont de 512, 1024 et 4096 octets.

Le FULL Disk Encryption (FDE) consiste à chiffrer tous les blocs d'un conteneur : partition logique ou conteneur de fichier. L'utilisation est répandue dans les systèmes d'exploitation

Unix : par exemple Fedora propose de chiffrer la partition complète lors de l'installation, ou la clé USB lors de sa détection. De plus pendant de nombreuses années, les systèmes Android proposaient le FDE pour chiffrer les données stockées dans le smartphone. Ces solutions sont toutes basées sur les outils LUKS/cryptsetup et le module noyau Linux *dm-crypt*. Dans les systèmes Windows, le logiciel Bitlocker permet de gérer le chiffrement FDE.

La première étape du FDE consiste à créer un périphérique virtuel, puis à le formater avec un système de fichier (comme ext4, FAT etc.). Puis de manière transparente, chaque écriture d'un bloc se traduit par un chiffrement en amont et chaque lecture de bloc par un déchiffrement en aval. L'avantage du FDE est de garantir nativement la protection des quatre catégories citées précédemment (nom de fichier, contenu, meta-données, arborescence) de manière intrinsèque. Le système de fichier joue son rôle de gestionnaire de fichiers sans nécessité d'être modifié.

2.4 Le Cloud

Dans cette partie nous allons analyser les systèmes existants offrant un service Cloud. De nombreux fournisseurs de Cloud existent sur le marché, les services proposés dépendant de la politique de chaque fournisseur. Il est utile de discerner les Clouds publics et privés. Le premier implique une externalisation (i.e. une perte du contrôle) des données : une fois les données stockées dans le serveur du fournisseur, le client n'a en effet aucun moyen de s'assurer de la non-divulgation des données à des entités tierces ou que le fournisseur lui-même n'accède pas à l'information. A l'inverse un Cloud privé garantit une meilleure protection, les données étant conservées au sein de l'entreprise.

Il est sûr que la sécurité des systèmes Cloud est un enjeu majeur. Le rapport de recherche de Thales et l'institut Ponemon[pon12] présente une enquête interrogeant plus de 400 entreprises de divers pays. Parmi les résultats, on note la question suivante : "est ce que votre organisation transfert ou prévoit de transférer des données sensibles et confidentielles dans le Cloud ?". A cette question, 49 % répondent positivement, 33 % prévoient de stocker des données sensibles dans les 2 ans et 19 % répondent négativement. Ainsi, en comptabilisant les entreprises 2 années plus tard, plus de 80 % des entreprises conserveront des données sensibles et confidentielles dans le Cloud. Cette information révèle le risque important d'une compromission des fournisseurs de Cloud. Une autre question révèle que les entreprises effectuant le chiffrement des données eux-mêmes (35 %) ne sont pas majoritaires. Enfin, l'étude soulève la question de gestion des clés de chiffrement. Les résultats montrent que 36 % des entreprises gèrent eux-mêmes les clés. Le reste des entreprises gèrent les clés via une tierce partie (22 %), via le fournisseur (22 %), via une combinaison des deux (18% -fournisseur + entreprise-), le reste autrement (1 %).

Regardons de plus près les façons dont les données peuvent être protégées.

2.4.1 Chiffrement côté serveur

Le chiffrement côté serveur n'est pas toujours appliqué, de nombreuses entreprises stockent les données en clair[csa15] alors que le chiffrement des données en transit est toujours mis en place. Les révélations Snowden ont changé le point de vue des entreprises, la "relation de confiance" a été brisée et le niveau d'exigence en matière de sécurité augmenté.

Depuis cette période, certains Cloud proposent de protéger les données selon plusieurs modèles. Certains Cloud proposent de chiffrer les données tout en conservant la gestion des clés comme Dropbox. D'autres proposent de chiffrer les données côté client tout en ayant une copie des clés pour des cas d'urgence. Enfin, la localisation géographique des données a pris de l'importance du fait des disparités entre les législations en vigueur des pays où sont stockées les données[PGB11].

Malgré ces efforts le chiffrement côté serveur est de plus en plus remis en cause.

2.4.2 Chiffrement côté client

Une des solutions ayant pris de l'ampleur dans la protection des données est le chiffrement côté client. Nous avons vu précédemment que certaines entreprises protègent les données en transit via le chiffrement alors que d'autres vont plus loin et chiffrent les données avant l'envoi vers le Cloud. Nous allons nous attarder sur cette solution. L'avantage majeur du chiffrement côté client est que la donnée sera, si le chiffrement est effectué de manière sûre (i.e., à condition que les clés de chiffrement soit aussi gérées côté client), illisible pour le fournisseur de Cloud lui même et a fortiori pour des entités externes accédant légitimement ou non à cette donnée.

Les contreparties sont le coût et la gestion de cette sécurité. En effet, ce chiffrement impose un surcoût de calcul/de bande passante et une gestion des clés complexe. La gestion devient également plus élaborée si une donnée est partagée par plusieurs utilisateurs : il peut être nécessaire par exemple en cas de chiffrement symétrique de partager cette clé entre tous les utilisateurs désirant accéder à la donnée.

Il existe différents logiciels et outils permettant d'effectuer le chiffrement des données côté client. Les solutions se basent sur un des 3 niveaux décrit précédemment : niveau fichier, niveau système de fichiers et niveau bloc. Nous décrivons rapidement chaque solution pour finalement insister sur celle suivie durant cette thèse.

— **Solutions niveau fichier :** les logiciels de bureautique comme LibreOffice ou Microsoft Office permettent de chiffrer individuellement les fichiers édités. La protection dans ces solutions se limite au contenu du fichier. Il est possible de chiffrer les fichiers avec ces logiciels et les transférer sur le Cloud.

— **Solutions niveau système de fichier :** les logiciels de cette catégorie peuvent se définir comme des solutions "à la Dropbox", chaque dossier étant synchronisé avec le Cloud. Toutes les protections de base (nom de fichier, contenu, meta-données, arborescence) plus d'autres services comme la dé-duplication et la gestion de version sont implémentés au sein d'une application. Cette flexibilité du choix de service engendre des complexités d'implémentation et des problèmes de performance. Par exemple l'application Cryptomator basée sur une solution JAVA et un système client serveur WebDav a récemment enlevé dans une nouvelle version la fonction d'obfuscation de la taille du fichier[cry16a] pour des raisons d'incompatibilités avec d'autres services.

— **Solutions niveau bloc :** les solutions intégrant le chiffrement de disque complet (FDE) n'ont pas ces problèmes : une protection complète des données (noms de fichiers, contenu ...) est garantie sans ajout complémentaire. La solution libre fonctionnant dans les systèmes Unix est le trio LUKS, cryptsetup et dm-crypt. La majorité des autres solutions sont propriétaires, telles que Dell Data Protection[del17] ou McAfee Complete Data Protection Advanced[MCA17]. Par ailleurs, les systèmes d'exploitation intègrent aussi des outils comme BitLocker pour Windows et FileVault pour macOS.

Toutes les solutions Cloud se basent sur un de ces 3 niveaux. Une fois le niveau choisi, un autre problème majeure concerne la gestion des clés. Dans la section suivante nous allons voir les différentes manières de gérer les clés de chiffrement.

2.5 Gestion des clés de chiffrement

La gestion des clés de chiffrement est primordiale. Dans cette partie, l'analyse portera sur les différentes méthodes utilisées pour gérer et protéger ces clés.

La première méthode consiste en *une dérivation de passphrase* avec un nombre d'itérations paramétrable (en fonction de la puissance du poste client afin d'avoir un temps d'attente adapté) et une fonction de hashage. Le but est de calculer le condensé de la passphrase des milliers voire centaines de milliers de fois afin de maximiser le temps de récupération de la clé principale (Master Key -MK-). Les mécanismes utilisés auparavant étaient basés sur la fonction de hashage MD5 itérée 1000 fois par exemple (e.g. *MD5 crypt[Kam94]*), ou sur l'algorithme de chiffrement blowfish (e.g. [PM99]).

Aujourd'hui de nombreuses solutions Cloud utilisent le PBKDF2 (Password Based Key Derivation Function) créé par le laboratoire RSA[Kal00]. Une limite de ces algorithmes est de ne pas être suffisamment robuste face à des attaquants possédant du matériel dédié aux attaques par force brute, capables d'exécuter des tests en parallèle via des cartes FPGA. Toutefois, *scrypt*[Per09], un algorithme récent, diffère dans son implémentation car il est non seulement

consommateur en CPU (temps) mais aussi en mémoire. Ainsi, ces mêmes attaquants ne pourront pas tirer pleinement profit de leur matériel dédié.

Il est important de mentionner l'apport considérable de Broz et. al[BM15] dans l'étude des perspectives des différents nouveaux algorithmes dans le cadre de la compétition *Password Hashing Competition*. Ils avaient retenu et annoncé trois algorithmes pour le FDE : Argon2, Lyra2[SJAA+14] et yescrypt[Pes14]. Finalement, l'algorithme finaliste a été Argon2[BDK16]. Ce dernier permet de paramétrer trois coûts : le temps, la mémoire et le parallélisme. En conséquence, chaque environnement travaillant avec cet algorithme peut adapter le niveau de sécurité en fonction de ces paramètres. Il est certain que plus ceux-ci seront exigeants et plus les postes clients devront être performants.

Une deuxième méthode de protection des clés de chiffrement est basée sur un *Trusted Plateform Module[Kin06] (TPM)*. Il consiste en un module matériel physiquement rattaché à la carte mère permettant la génération et la conservation des clés en zone mémoire protégée. Le TPM comporte une paire de clés RSA non extractibles. La Master Key est alors protégée par *la clé publique* du TPM. On peut alors stocker la Master Key dans une mémoire quelconque du disque ou la mémoire *très limitée* du TPM. On peut en outre la déchiffrer avec *la clé privée*.

Néanmoins, la paire de clés RSA étant générée lors de la fabrication, les fabriquants sont théoriquement en mesure de conserver une copie de ces paires de clés. Mais si l'on passe outre cette limite et si on accorde sa confiance au fabriquant, alors le TPM peut être exploité pour vérifier la signature des logiciels de bas-niveau utilisés lors de la phase de démarage et détecter toute altération de ces derniers.

Chapitre 3

Les défis du Cloud personnel TTM : "Trust The Module, Not The Cloud"

Sommaire

3.1 Introduction

Le chiffrement des données est une protection complémentaire à toutes les autres protections telles que les systèmes de détection d'intrusion (IDS) ou bien les pare-feux permettant un

contrôle avancé des entrées sorties. Il faut savoir que de nombreux outils permettent de protéger les données personnelles conservées localement sur le poste client. Cette protection peut être mise en place via des systèmes sur plusieurs niveaux : fichier, système de fichier et bloc (cf. section 2.4.2).

Pour procéder au choix du type de chiffrement, il est important de déterminer les besoins et le niveau de protection que l'on souhaite obtenir. Ci-dessous quelques critères non exhaustifs :

- **le chiffrement total/partiel :** les fichiers stockés dans un système de fichier suivent une organisation interne. Chaque fichier est composé du contenu et des informations annexes appelées *meta-données* (nom du fichier, date de modifications, taille etc.). En fonction du besoin, il peut être nécessaire de masquer les meta-données. Effectivement, un nom de fichier peut fournir des informations critiques sur une entreprise par exemple.
- **le lieu de stockages des données en clair / chiffrées :** de nombreuses solutions fonctionnent *"à la Dropbox"* où les fichiers sont synchronisés avec le serveur de stockage. Ceci implique qu'une version en clair est conservée dans le poste client. D'autres Cloud proposent de synchroniser les fichiers chiffrés avec le serveur de stockage. Enfin, la dernière solution est de ne rien conserver dans le poste client mais uniquement sur le Cloud.
- **le chiffrement en session ouverte :** ce critère est aussi essentiel dans le choix du niveau de chiffrement. A titre d'exemple, le FDE protège contre le vol ou la perte de l'ordinateur, à condition que ce dernier soit éteint ou en veille prolongée. En revanche, si la session ainsi que la partition protégée étaient ouvertes, un attaquant dérobant l'ordinateur pourrait lire les données ou les copier sur un appareil de stockage externe (USB). Ce risque reste valable pour les autres niveaux de chiffrement mais dans une moindre mesure. Manifestement, il est courant d'ouvrir les partitions chiffrées au démarrage de l'ordinateur alors que ce n'est pas forcément le cas pour un chiffrement de fichier spécifique : le déchiffrement du fichier en question est nécessaire uniquement durant son accès. Une attention particulière doit également être portée sur la RAM où les clés sont stockées car de nombreuses recherches ont mis à nu les risques d'extraction de clé en redémarrant le système à froid (*cold boot attack*)[HSH+09].
- **la gestion des clés :** est d'une importance majeure dans la protection des données personnelles. Suivant le niveau de chiffrement choisi, deux possibilités existent. Le FDE par exemple utilise une seule Master Key (MK), responsable du chiffrement de tout le conteneur. Les autres niveaux par contre (fichier et système de fichiers - voir section 2.4.2) permettent plus de granularité, on peut utiliser une clé par fichier ou par répertoire. Plusieurs mécanismes de protection de la clé de chiffrement existent, le plus simple parmi eux réside dans une dérivation de passphrase avec une fonction de hashage. Un second mécanisme est l'utilisation d'un module matériel physiquement rattaché à la carte mère appelé TPM[JH10]. Ce dernier comporte une paire de clés RSA non extractibles. La

Master Key est alors protégée par la clé publique et stockée dans une mémoire quelconque du disque, puis déchiffrée avec la clé privée.

Tous ces critères sont à prendre en compte dans l'établissement d'une architecture Cloud. Dans cette même veine, nous avons choisi le principe suivant décrivant notre ligne directrice : *"Trust The Module, Not The Cloud"* que nous appellerons TTM par la suite. De ce fait, toutes les opérations cryptographiques ainsi la sécurité en général (liée à la protection des données) sont déportées sur un module placé entre le poste client et le Cloud. Si nous devions résumer le fonctionnement de notre architecture, nous souhaiterions :

— **stocker** les données **sur le Cloud** ;
— **chiffrer et déchiffrer** les données **sur le module** ;
— **accéder** aux données **en clair sur le poste client**.

3.2 Modèle d'attaque

Afin de concevoir une protection fiable dans le cadre de la solution TTM, nous avons défini deux adversaires :

— **un Cloud honnête mais curieux :** l'hebergeur de données est considéré comme honnête. Il exécute les différents services proposés et ne va pas tenter de corrompre les données. Il est cependant considéré comme curieux, susceptible de lire les données stockées à sa propre initiative ou par obligation vis-à-vis d'une autorité (gouvernementale par exemple).
— **un attaquant extérieur :** peut s'infiltrer au milieu de la connexion module-Cloud (en général Internet) et interagir avec le flux (écoute, insertion, supression de paquets). Il peut également s'introduire dans l'infrastructure du fournisseur de services Cloud et accéder à toutes les données chiffrées.

Il est utile de définir deux notions à propos du chiffrement des données : *le chiffrement en transit* et *le chiffrement au repos*. *Le chiffrement en transit* correspond à une protection des données transférées d'un point A vers un point B. Ces données sont chiffrées à la sortie du point A, transférées de manière sécurisée vers le point B, puis déchiffrées. *Le chiffrement au repos* quant à lui correspond à une protection des données stockées dans un support de stockage. Ces données ainsi chiffrées seront conservées dans le support de stockage.

Dans le schéma 3.1 qui illustre le modèle d'attaque, nous distinguons deux liens principaux :

— **Le lien Client - Module :** ce lien est considéré comme lien de confiance.
— **Le lien Module - Cloud :** ce lien est ouvert à l'extérieur et donc sujet à attaques.

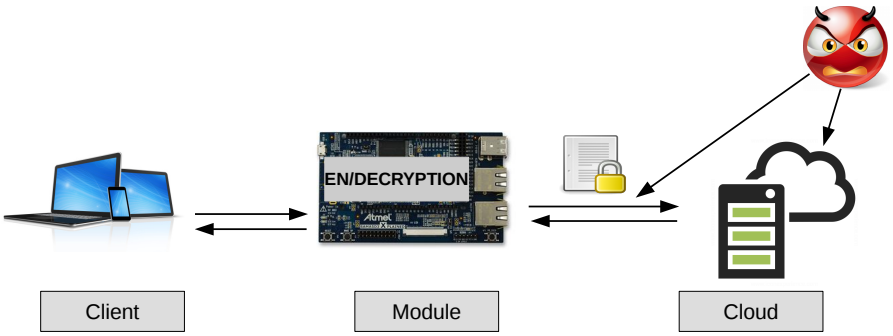


FIGURE 3.1 – Modèle d'attaque de la solution Cloud personnel TTM.

3.3 Énoncé du problème et des besoins

Le but est de mettre en place une **solution de Cloud personnel sécurisée et transparente** pour l'utilisateur basé sur un module. Les données seront chiffrées dans le module puis envoyées sur le Cloud. Ainsi le principe de *"Trust The Module, Not The Cloud"* (TTM) doit être respecté. L'usage est personnel et familial dans le but de protéger des fichiers tels que les photos, les musiques, les documents textes. Nous allons expliquer les raisons de la nécessité d'avoir un module comme pré-requis à notre solution.

3.3.1 Pourquoi utiliser un module dédié

La protection des données est un vaste domaine. Les solutions pour y parvenir existent à travers d'innombrables outils. En particulier, les solutions de chiffrement de données au sein même de l'appareil final existent (7-Zip, Veracrypt, Bitlocker). Contrairement à celles-ci, nous avons choisi, dans cette présente thèse, d'utiliser un **module en rupture** entre le **poste client** et le **Cloud**. Cette condition fait partie des exigences établies avec l'entreprise INCAS IT-Sec.

La Figure 3.2 illustre cette approche avec 3 entités :

- **Le Client :** se connecte au module afin d'accéder à une vue en clair des fichiers stockés dans son espace personnel.
- **Le Module :** placé en rupture entre le Client et le Cloud, il s'occupe du chiffrement et déchiffrement des données ainsi que toutes les opérations de sécurité.
- **Le Cloud :** a pour rôle de fournir des services de stockage où les données sont conservées de manière chiffrées.

Assurément le choix d'un module en rupture présente comme toutes solutions des avantages et des inconvénients en fonction du cas d'usage. Dans le cadre d'un Cloud personnel, l'avantage

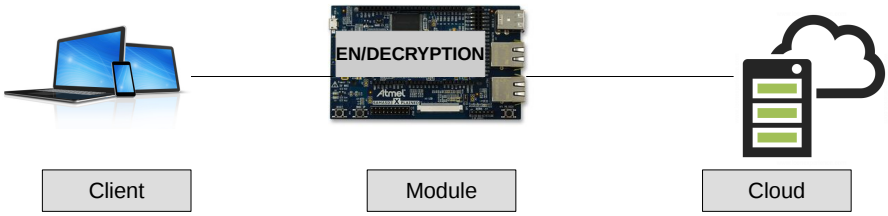


FIGURE 3.2 – Vue simplifiée d'une solution Cloud personnel basée sur un module dédié.

principal est une facilité d'utilisation pour le client, capable de connecter tous ses appareils au module (PC, tablette, smartphone) et d'accéder à ses fichiers de manière transparente. Un service de connexion et d'accès au fichier standard doivent cependant être présents sur le Client. Dans les lignes suivantes, nous détaillerons ces considérations.

3.3.2 Les besoins

La solution TTM que nous proposons exige le respect de certains critères de fonctionnalité et de sécurité.

- **La confidentialité des données :** les données doivent être protégées face à un hébergeur curieux ou un attaquant malicieux.
- **La modularité :** la solution doit être modulaire. En effet, elle doit permettre l'ajout des fonctionnalités sans remettre en cause l'architecture globale.
- **La transparence :** la solution doit être facile d'utilisation et transparente pour l'utilisateur. Pour ce faire, les pré-requis sur le Client doivent être minimaux.
- **L'abstraction du serveur Cloud :** la solution doit permettre l'usage d'un ou plusieurs Clouds de façon transparente pour l'utilisateur.
- **Pré-requis sur le serveur Cloud** : les installations sur le Cloud doivent être minimales. Par ce biais, un grand nombre hébergeurs seront compatibles avec la solution TTM.
- **Minimisation des données rapatriées sur le module et le client :** les fichiers doivent être transférés et déchiffrés uniquement lors de l'accès.
- **L'intégrité :** les données conservées dans le Cloud ne doivent pas être modifiées ou altérées à l'insu de l'utilisateur.
- **La mobilité :** le module doit être de petite taille, peu encombrant et facilement transportable.
- **L'accessibilité :** le module doit être économique afin de permettre l'accès au plus grand nombre. Dans un contexte d'externalisation massive de toutes les données personnelles, fournir un Cloud permettant de protéger la vie privée à un prix accessible reste un critère important.
- **La flexibilité du contrôle d'accès :** plusieurs utilisateurs doivent être en mesure de

posséder leur propre espace de stockage bien qu'ayant le même module.

3.4 Les solutions Cloud applicatives

3.4.1 Analyse des solutions existantes

Ces dernières années le chiffrement de données côté client a été à l'origine de nombreuses initiatives. La majorité d'entre elles se positionnent au niveau du système de fichier (comme décrit dans la section 2.4.2). Ce niveau permet de protéger plusieurs voire toutes les catégories, e.g., contenu, meta-données (cf. section 2.3). Nous allons analyser en détail quelques solutions anciennes et récentes.

La plus ancienne des solutions est *EncFs*. Il fonctionne au niveau de l'espace utilisateur. Seules les permissions d'utilisateurs régulières sont nécessaires grâce à la bibliothèque FUSE. Par ailleurs, pour son chiffrement, EncFs utilise par défaut la bibliothèque cryptographique OpenSSL. Le mode de chiffrement est le CBC-AES. En revanche EncFS protège uniquement le contenu des fichiers et leurs noms. Seulement, elle ne chiffre ni l'arborescence des fichiers ni les méta-données. Pour le premier, le nombre exact de répertoires et de fichiers est visible à l'hébergeur ou à un attaquant alors que pour le second, la taille ou la date d'accès des fichiers est visible à l'hébergeur.

Cette implémentation souffre cependant de nombreuses critiques quant à la sécurité. Effectivement, un audit de sécurité [Hor14] a révélé de nombreuses faiblesses sérieuses de cette implémentation. Dans sa conclusion, les auteurs décèlent un risque de compromission de la confidentialité si un attaquant est capable de récupérer 2 ou 3 versions (instantanées) chiffrées d'un même fichier. En conséquence, une utilisation de EncFs dans le cadre du Cloud n'est pas envisageable, puisque l'hébergeur a accès à de nombreux instantanés.

A la suite de EncFS, *eCryptfs* est une autre solution inclus dans le noyau Linux. Elle est utilisée dans la distribution Ubuntu afin de protéger le répertoire personnel des utilisateurs (homedir). Similairement à EncFs, les meta-données et l'arborescence des fichiers ne sont pas protégées.

Dans la même lignée, d'autres solutions comme CryFs et Syncany ont vu le jour. Elles découpent les fichiers en blocs de données, assurent leur intégrité ainsi que leur confidentialité tout en protégeant l'arborescence des fichiers. Cryptomator[cry16a] leur fait écho. Elle se démarque en utilisant non pas FUSE mais le protcole WebDAV. Celui-ci donne accès aux fichiers présents sur un serveur WebDAV via un client web intégré dans de nombreux systèmes d'exploitation.

Dans toutes ces solutions, le mot de passe de l'utilisateur est dérivé à l'aide de deux algorithmes : scrypt ou PBKDF2. Cette dérivation a pour but de protéger la clé de chiffrement tout en freinant

les attaques par bruteforce. Par ailleurs, en terme de fonctionnalité, les solutions CryFs[cry17] et Syncany délivrent une protection plus complète en proposant un chiffrement du contenu, des noms de fichier, des meta-données et une obfuscation de l'arborescence de fichiers.

Ces implémentations reposent pour chaque solution sur une gestion des fichiers différentes. Syncany par exemple s'appuie sur un échange de bases de données entre le client et le Cloud. D'autres proposent de conserver les informations liées au chiffrement dans l'en-tête du fichier. Un avantage non négligeable pour ces derniers demeure la flexibilité d'utilisation. Ainsi, un utilisateur peut protéger ses fichiers dans un dossier avec une clé, et d'autres dans un second dossier (i.e. avec une seconde clé). De plus, contrairement aux solutions avec FDE, il n'est pas nécessaire de connaître à l'avance la taille de l'espace de stockage.

Incontestablement dans toutes ces solutions monolithiques, où tous les services sont regroupés, deux limites apparaissent. La première traduit une complexité d'implémentation : plus le nombre de services augmente, plus les difficultés d'implémentation accroissent. En guise d'exemple, Cryptomator fut contraint de retirer l'obfuscation de la taille des fichiers suite à des problèmes de compatibilités avec d'autres services[cry16b]. La deuxième limite, quant à elle, concerne les performances : le regroupement et l'accumulation des services peuvent restreindre les débits globaux de chiffrement et déchiffrement. Ceci nous amène à tester la faisabilité de la solution Syncany sur une carte Raspberry Pi sous Linux.

3.4.2 Étude de cas avec Syncany et un Raspberry Pi

Syncany[Hec12] est un logiciel libre implémenté en JAVA permettant le stockage et le partage de données via de nombreux fournisseurs de Cloud (Amazon, Google) ou de serveurs standards (SFTP, Webdav). Son architecture est composée uniquement de postes clients et d'un serveur de stockage dépourvu d'"intelligence". Autrement dit de simples commandes comme l'envoi/la réception sont suffisantes à faire fonctionner Syncany avec ce dernier.

Pour cela, les fichiers sont découpés en morceaux de taille fixe, appelés *chunk*, puis répertoriés dans des bases de données. Une comparaison des bases entre le poste client et le serveur permet de synchroniser le tout pour détecter les changements : création, modification, suppression.

De plus, Syncany protège, à la manière du FDE, toutes les catégories du système de fichiers : les noms de fichier, les meta-données et l'arborescence sont chiffrés/obfusqués. On trouve aussi d'autres services :

— **La dé-duplication :** Syncany découpe les fichiers en bloc de données et conserve le condensé correspondant dans une base de données. Ces opérations sont effectuées du côté du client. Ainsi une fois le fichier modifié, seuls les blocs modifiés seront envoyés vers le Cloud.

— **Le chiffrement :** l'algorithme utilisé est un chiffrement symétrique authentifié et son mode associé est le GCM. Ce dernier aide non seulement à chiffrer la donnée ainsi que de s'assurer de son intégrité. Sa clé est dérivée à partir de PBKDF2 pour environ 100 blocs. Ceci correspond à environ 400 MB de données (chaque bloc ayant une taille maximum de 4MB).
— **La synchronisation :** est effectuée via des traces de logs (historique des fichiers) pour chaque client. La détection de nouvelles versions s'effectue en analysant et comparant les historiques de tous les clients connectés au répertoire commun. Les éléments de comparaison sont les identifiants des fichiers, la version du fichier, les condensés et les horodatages. En outre, en cas de conflit, la version la plus à jour est récupérée et le fichier en conflit renommé.

La plateforme expérimentale

Nous avons utilisé une carte Raspberry Pi dotée d'un processeur ARMv6 (@700 MHz). Ce nano-ordinateur, a un prix accessible, regroupe une communauté importante. De nombreux projets libres existent et utilisent les ressources offertes par ce composant. Elle comporte plusieurs périphériques de communication. Sa liaison avec l'extérieur peut se faire via le port ethernet ou par Wifi en utilisant une clé usb spécifique [1]. De ce fait, la communication entre le client et le module peut s'effectuer à l'aide du Wifi, tandis que le module communique avec l'extérieur par le biais du port ethernet.

A la suite, nous avons testé Syncany et mis en place un partage entre plusieurs utilisateurs. Pour cela, deux Raspberry Pi (RP) ont été connectés à un serveur, chacun avec un sous-réseau différent. La plateforme choisie présente trois éléments (voir Figure 3.3) :

— **Le serveur de stockage SFTP :** accessible à partir des deux sous-réseaux. A cet effet, nous avons créé un utilisateur pour chaque client.
— **Le Raspberry Pi :** le module de sécurité en rupture permettant de faire le lien entre le client et le monde extérieur.
— **Le Client :** l'appareil peut être un ordinateur, un smartphone ou bien une tablette.

Pour tout tester, Syncany a été installé sur les modules Raspberry Pi. Le test consistait à créer un répertoire pour le client A et le partager avec le client B. Les figures montrent l'accès à ce répertoire sur chacun des clients.

Pour mesurer les performances, deux scénarios ont été définis :

— **Avec le serveur local :** le dépôt (chiffré) et les fichiers synchronisés (en clair) sont stockés sur le même appareil (i.e. le Raspberry Pi).
— **Avec le serveur SFTP :** les fichiers synchronisés (en clair) sont stockés sur le Raspberry Pi et le dépôt (chiffré) dans le serveur SFTP.

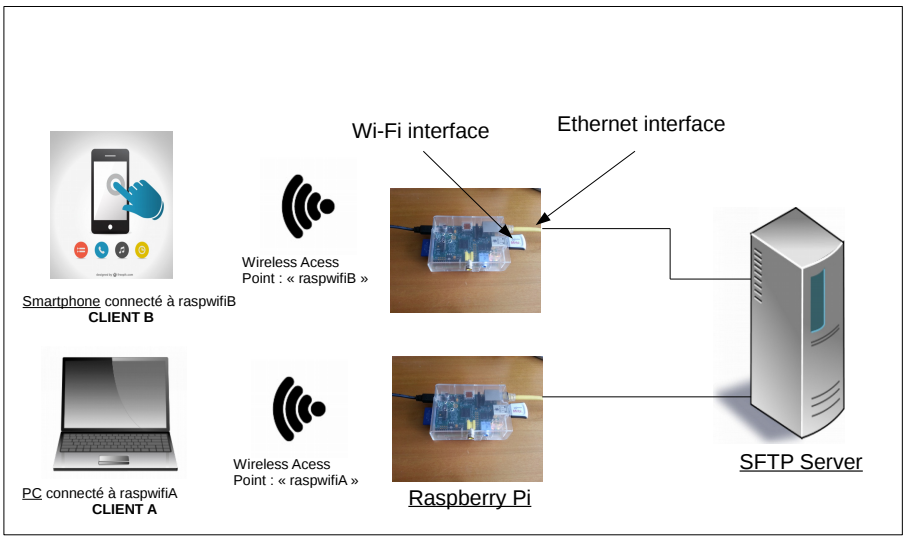


FIGURE 3.3 – Cloud personnel - syncany

	Serveur local		Serveur SFTP	
	Avec chif.	Sans chif.	Avec chif.	Sans chif.
5MB	80	75	140	90
100MB	∞	95	∞	170

TABLE 3.1 – Temps (en secondes) pour envoyer un fichier au dépôt local/SFTP avec Syncany.

Le tableau 3.1 montre les résultats. On remarque les temps très long qui s'expliquent par deux raisons :

— Le Raspberry Pi est un nano-ordinateur avec des ressources CPU très limitées.

— Syncany est un logiciel Java consommant beaucoup de ressources mémoires et CPU.

Ainsi, dans le cas d'un serveur local, le temps pour charger un fichier de 5MB est supérieur à la minute (75s). Le même fichier avec chiffrement nécessite un temps de 5s supplémentaire. On peut en déduire alors que l'impact du chiffrement dans ce cas est minime par rapport au temps total. En plus pour le même fichier avec un serveur SFTP, le temps passe de 90s sans chiffrement à 140 secondes avec chiffrement. Pour un autre fichier de 100MB le temps sans chiffrement passe de 95 secondes pour un serveur local à 170 secondes avec un serveur SFTP. On note toutefois pour un fichier de même taille, qu'avec le chiffrement, le temps devient extrêmement long (plusieurs heures).

Conclusion avec le Raspberry Pi

Tout bien considéré, la solution avec la carte Raspberry Pi a montré ses limites car ses performances sont très faibles. Pareillement, la solution Syncany souffre de problèmes de stockage et de sécurité. En effet, une version en clair des fichiers doit être conservée sur le Raspberry

1. pour les versions récentes de cette carte, le Wifi est intégré

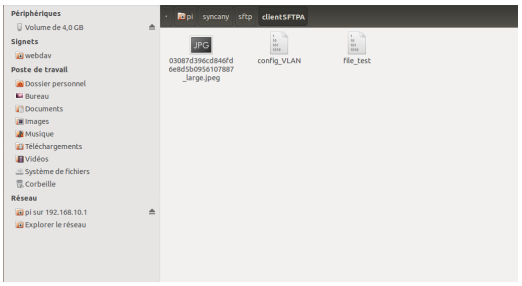


FIGURE 3.4 – Connexion au Raspberry Pi depuis le PC

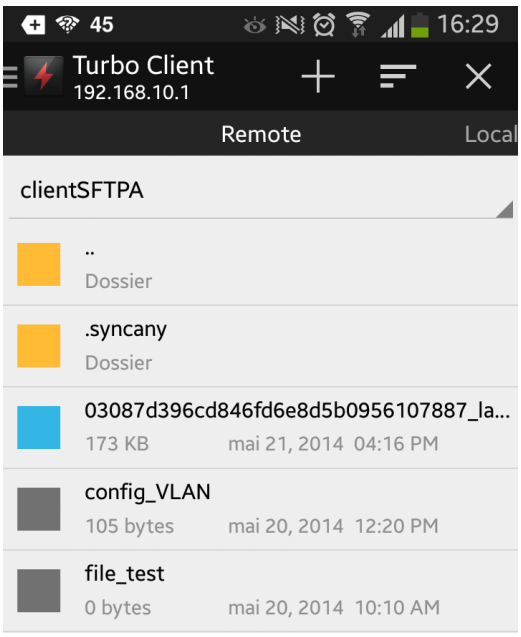


FIGURE 3.5 – Connexion au Raspberry Pi depuis le smartphone sous Android

Pi, ce qui va à l'encontre de notre approche TTM qui impose que les données chiffrées soit uniquement stockées sur le Cloud. D'ailleurs, ce principe renvoie à la solution connue Dropbox, qui, après installation du logiciel, télécharge et synchronise tous les fichiers sur le client.

C'est pourquoi nous avons choisi de nous orienter vers une autre solution orientée bloc. En guise d'exemple et par comparaison avec Syncany, nous avons mis en place un test simple de conteneur dans le cadre d'une solution FDE sous Linux avec le couple dm-crypt / cryptsetup. La carte Raspberry Pi reste identique. Les tests ont été effectués en local. Nous avons constaté des temps réduits avec une débit de déchiffrement de plus de 5 MB/s. Ce qui correspond à un temps de transfert (i.e. déchiffrement ou chiffrement), pour un fichier de 100 MB de 17,4 secondes (contrairement à 95 secondes pour la solution avec Syncany et un serveur local).

En somme, l'analyse des limites de la carte Raspberry Pi et de la solution Syncany nous ont confortés vers la solution TTM que nous abordons maintenant.

3.5 Notre architecture TTM : "Trust The Module, Not The Cloud"

3.5.1 Introduction à l'architecture TTM

Le module dans notre architecture a un rôle central. Après un rapide parcours des limites des solutions Cloud applicatives, où tous les services sont regroupés dans une application mono-

lithique, nous nous sommes orientés vers une solution où la distribution et le chiffrement des données sont descendus au niveau du "block device".

Zoom sur le block device

Un block device est une abstraction requise pour accéder aux supports de stockage comme les disques durs. Il permet l'accès au travers d'une mémoire tampon aux données stockées sur le support. La notion de "bloc" peut être source de malentendus, il faut distinguer deux types de "blocs" :

- le secteur physique d'un disque correspond à l'unité fondamentale de tous les supports de stockage. La taille de ces blocs est généralement de 512 octets.
- le bloc logique est une abstraction du système de fichiers. Il correspond à la plus petite unité logique adressable. Ces valeurs usuelles sont 512, 1024 et 4096 octets.

Notre solution TTM se veut modulaire. Elle s'appuie sur deux concepts architecturaux fondamentaux :

- **la distribution transparente** basée sur le protocole iSCSI.
- **le chiffrement transparent** basé sur le FDE.

Les besoins décris dans la section 3.3.2 sont adressés à travers ces deux principes.

3.5.2 Premier principe architectural : la distribution transparente

Le point essentiel de notre solution TTM est que les données doivent être intégralement stockées sur le Cloud et non sur le poste client ou sur le module. Notre solution reposant sur le FDE, nous avons besoin de transférer sur le Cloud des blocs logiques.

Pour cela, le protcole iSCSI est parfaitement adapté puisque ce dernier fonctionne aussi au niveau du bloc. Ce protocole est une version évoluée de SCSI qui est nativement présent dans la plupart des appareils possédant des supports externes tels que des clés USB, des disques durs. iSCSI permet de rassembler les ressources de stockage dans un centre de données SAN (Stockage Area Network) tout en donnant l'illusion que le stockage est *local*. La distribution de données est donc gérée de façon totalement transparente sur le module.

Zoom sur les protocoles de stockage à distance basés sur SCSI

Le standard Small Computer System Interface (SCSI) est un ensemble d'outils permettant de relier physiquement les ordinateurs avec les périphériques externes. Utilisé principalement pour les appareils de stockage, le standard SCSI est initialement prévu pour une utilisation locale. La portée étant de limitée à quelques mètres, une version évoluée basée sur le protcole IP a vu le jour appelé Internet Small Computer System Interface (iSCSI).

Cette nouvelle version a été standardisée par l'IETF en 2004[SMS+04].

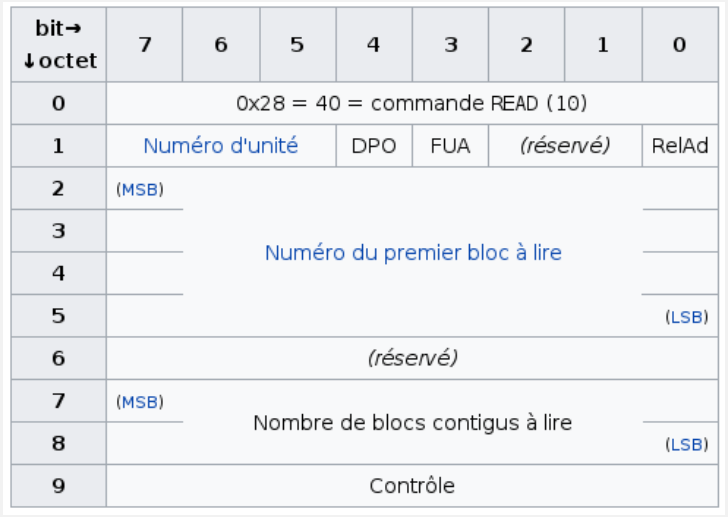
En transportant les commandes SCSI sur les réseaux IP, iSCSI est utilisé pour faciliter le transfert et le stockage de données sur de longues distances. iSCSI est prévu pour fonctionner dans différents types de réseaux, du réseau local (LAN) au réseau étendu (WAN). Des clients (appelés *initiateurs*) se connectent à des périphériques de stockage SCSI (appelés *targets*) sur des serveurs distants à travers des commandes SCSI (CDB). Le protocole permet de rassembler les ressources de stockage dans un centre de données SAN (Stockage Area Network) tout en donnant l'illusion que le stockage est *local*.

Les commandes SCSI envoyées respectent un format appelé CDB (Command Descriptor Block). Nous allons illustrer la commande de lecture d'un bloc appelée READ(10). La Figure 3.6 expose les différents champs présents :

- **Octet 0 :** correspond au type de la commande, pour READ(10) : 0x28
- **Octet 1 :** les bits *DPO* et *FUA* donnent des informations quant à l'utilisation d'un cache ou non.
- **Octet 2-5 :** cette valeur de 32 bits correspond au numéro du premier bloc logique à lire (Logical Bloc Address - LBA). Ceci permet de fonctionner avec un disque d'une taille allant jusqu'à 2 Tio avec des blocs de 512 octets.
- **Octet 7-8 :** cette valeur de 16 bits correspond au nombre de blocs à lire.

Il est intéressant de noter que plus de 4 versions existent pour la commande READ.Nous avons présenté la commande READ(10). La différence concerne principalement la capacité en octets (bits) des champs LBA et le nombre de bloc à lire. Les capacités des disques étant de plus en plus grandes, le nombre de bit pour ces champs a augmenté en conséquence (e.g. READ(12) avec 32 bits pour les 2 champs en question).

Enfin, il existe aussi d'autres protocoles dont le *Fibre Channel*. Une évaluation des performances est établie par Simitci et. al[SMG01]. Bien que les performances soient plus élevées avec la fibre optique, l'inconvénient est la nécessité d'ajouter des adaptateurs de bout en bout (entre les initiateurs et les targets), ce qui induit un coût supplémentaire. Un rapport technique présente en détail les caractéristiques des deux protocoles[Sac01].



bit→ ↓octet	7	6	5	4	3	2	1	0
0	0x28 = 40 = commande READ (10)							
1	Numéro d'unité			DPO	FUA	*(réservé)*		RelAd
2	(MSB)							
3				Numéro du premier bloc à lire				
4								
5								(LSB)
6	*(réservé)*							
7	(MSB)							
8	Nombre de blocs contigus à lire							(LSB)
9	Contrôle							

FIGURE 3.6 – Commande SCSI READ (source Wikipedia).

3.5.3 Deuxième principe architectural : le chiffrement transparent

Le chiffrement par disque complet (Full Disk Encryption -FDE-) permet de chiffrer des espaces de stockage appelés *conteneurs*. De nombreuses implémentations propriétaires ou libres existent. On distingue entre autres *Bitlocker* qui a remplacé l'obsolète *TrueCrypt* sous Windows, *File Vault 2* sous MACOS et *dm-crypt* sous Linux. Pour les besoins de notre architecture notre choix s'est porté sur dm-crypt.

FDE sous Linux

Lors du chiffrement par conteneur, le support entier est chiffré. Ce type de chiffrement est apparu sous Linux pour protéger les données sur les disques durs des ordinateurs à partir de la version du noyau Linux 2.6. Le chiffrement ne se limite pas au disque de données. Il est aussi possible de protéger l'espace de pagination (swap space) ou la partition de démarrage (/boot). A l'origine, le chiffrement de conteneur est donc utilisé pour le chiffrement local. Ce faisant, les données sont protégées lors d'une perte ou d'un vol de l'appareil. Toutefois, si l'attaquant accède à l'appareil pendant une session ouverte, alors toutes les données lui seront accessibles.

Le chiffrement de disque sous Linux repose sur le module noyau *dm-crypt* permettant de créer une carte de périphérique (device mapper) chiffrant. Cela signifie que de manière transparente, lorsqu'un bloc du conteneur est lu, ce dernier est déchiffré à la volée et stocké en mémoire virtuelle temporairement. Réciproquement, lors de l'écriture d'un bloc, ce dernier est chiffré puis stocké dans le disque physique. En outre, le module *dm-crypt* s'appuie sur le Linux kernel crypto API. Son interface, intégrée au noyau Linux, permet de faire appel à des algorithmes

de chiffrement présents en tant que module-noyau. Ainsi, on retrouve dans le noyau Linux des modules pour chaque mode de l'AES par exemple (cbc.ko, ecb.ko ...). L'API comporte aussi des primitives pour appeler toutes les transformations cryptographiques telles que le calcul de condensé, les algorithmes de chiffrement symétrique et asymétrique, les chiffrements authentifiés etc. A ce titre, on trouve deux types d'implémentations :

Les opérations synchrones : les opérations cryptographiques sont bloquantes et lors d'une requête à un algorithme de chiffrement, le programme attend le résultat avant de lancer la requête suivante.

Les opération asynchrones : ces opérations cryptographiques sont exécutées parallèlement au CPU. L'appel de fonction déclenche l'opération mais le résultat n'est pas attendu de manière bloquante. En effet, avant l'invocation, l'utilisateur se doit de fournir un appel de retour (callback) qui sera rappelé lors de la fin du traitement de la requête. Il est aussi essentiel pour l'utilisateur d'appliquer des mécanismes de verrouillage des données pour leurs éviter un traitement avant la fin de l'opération cryptographique. Le fonctionnement asynchrone suppose un matériel indépendant permettant d'exécuter de manière autonome les opérations cryptographiques. Un exemple est le co-processeur cryptographique des cartes Atmel que nous aborderons plus tard.

Parmi les modules noyau, outre l'aspect synchrone et asynchrone, nous pouvons les classer en 3 catégories :

Les modules 100 % logiciels : Les modules intégrés de base dans le Linux kernel crypto API s'appuient sur le processeur principal (CPU) et sont synchrones.

Les modules 100 % matériels : Les opérations cryptographiques sont déportées et effectuées dans des co-processeurs dédiés.

Les modules hybrides : Les processeurs intégrants des jeux d'instructions spécifiques entrent dans cette catégorie. Intel en guise d'illustration, a développé le jeux d'instructions aes-ni[ADF+10] qui fournit des gains de performance allant de 3 jusqu'à 10 pour les modes parallélisables. Ces instructions sont optimisées pour des tâches très précises.

Soulignons que si plusieurs de ces modules sont disponibles dans le noyau, celui dont la priorité est la plus élevée sera choisi par le Linux kernel crypto API. En règle générale, les modules matériels et hybrides ont une priorité supérieure au module 100 % logiciels.

Zoom sur le FDE sous Linux

Attardons nous un peu sur le fonctionnement de plus haut niveau du chiffrement de disque. L'outil **cryptsetup** permet de gérer les disques en s'appuyant sur le format **LUKS** (Luks Unified Key Setup). En d'autres termes, le conteneur est scindé en deux parties :

— **L'entête :** comporte les informations cryptographiques liées au chiffrement
— **Les données**.

Exemple du chiffrement des données dans un conteneur (Figure 3.7) : les données proviennent de l'espace utilisateur. Elle sont envoyées au device mapper via le module **dm-crypt**. Ensuite, une requête cryptographique est alors créée puis envoyée, en fonction de l'algorithme choisi et sa priorité, vers le module sélectionné automatiquement par le **Linux kernel crypto API**. Les données sont alors chiffrées puis conservées dans le conteneur. En revanche, le module peut être logiciel ou, comme illustré dans la Figure 3.7, basé sur un co-processeur (via un driver bas-niveau intégré au noyau **Linux** e.g. *atmel-aes*).

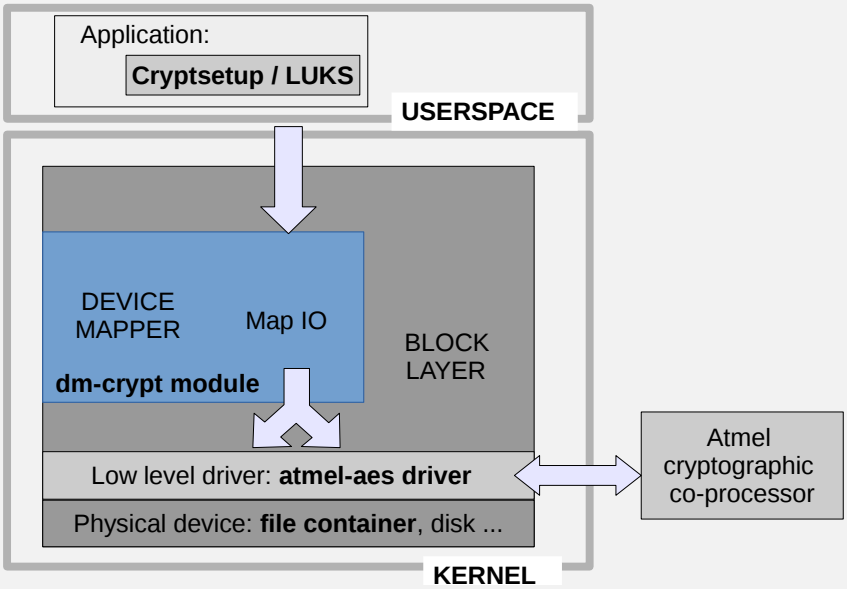


FIGURE 3.7 – Vue globale du FDE dans Linux.

3.6 Preuve de concept de l'architecture TTM pour un Cloud personnel

3.6.1 Vue d'ensemble

Nous avons implémenté une preuve de concept pour l'architecture TTM. Notre choix s'est porté sur la carte Atmel SAMA5D3 pour plusieurs raisons :

- la carte intègre un co-processeur cryptographique supportant les modes de base de l'algorithme AES.
- la carte intègre un générateur de nombre aléatoire PRNG (Pseudo Random Number Generator).
- la carte coûte environ 60 €.

Zoom sur la carte SAMA5D3

Voici les caractéristiques de cette carte :

- Processeur ARM cortex A5,
- 2 ports ethernet : un Gbit et un 100 Mbit,
- au moins 256 MB de mémoire vive (RAM),
- un co-processeur cryptographique dédié comprenant les modes ECB,CBC, CTR, OFB et CFB,
- un lecteur de carte SD sur lequel le système d'exploitation est installé.

3.6.2 Implémentation

L'architecture repose sur les principes de la **distribution transparente** et du **chiffrement transparent.** Les opérations cryptographiques se concentrent sur le module Atmel.

Différents protocoles de communication existent entre les composants (Figure 3.8) :

- **Entre le Client et le Module :** nous avons mis en place un serveur *sshfs* sur le Module. Le Client peut alors se connecter avec la sécurité du protocole de transfert de fichier SSH avec un *chiffrement en transit*. Un avantage de cette approche est que l'on accède au dossier comme un dossier local si bien qu'il est possible d'appliquer des outils de mesures de performances tel que *Iozone*. En somme, la plupart des systèmes d'exploitation modernes proposent nativement une connexion au serveur via l'explorateur de fichier, ce qui simplifie la procédure de connexion côté Client.
- **Entre le Module et le Cloud :** un serveur iSCSI target est installé dans le Cloud. Ceci est le seul pré-requis pour le serveur de stockage. Un initiateur iSCSI est installé sur le module Atmel.

La mise en place de cette preuve de concept obéit aux étapes suivantes :

1. *L'initialisation du conteneur sur le serveur de stockage* : un conteneur (appelé dans le protocole iSCSI) de taille fixe est alloué sous forme de partition logique ou fichier. Puis, un identifiant unique est attribué à cette target qui respecte, conformément au standard, la forme suivante : "iqn.domaine.com.nom.spécifique".
2. *La connexion au conteneur sur le module :* la connexion s'effectue via un outil d'administration iscsiadm en spécifiant l'adresse IP du serveur. Une fois connecté au conteneur, ce dernier apparaît comme un disque local dans le module sous la forme */dev/sda* par exemple.
3. *L'initialisation du conteneur sur le module :* le conteneur est initialisé via la commande *cryptsetup luksFormat*. Cette commande prend en argument une passphrase/clé qui est par la suite dérivée afin d'obtenir la Key Encryption Key (KEK). En parallèle, une

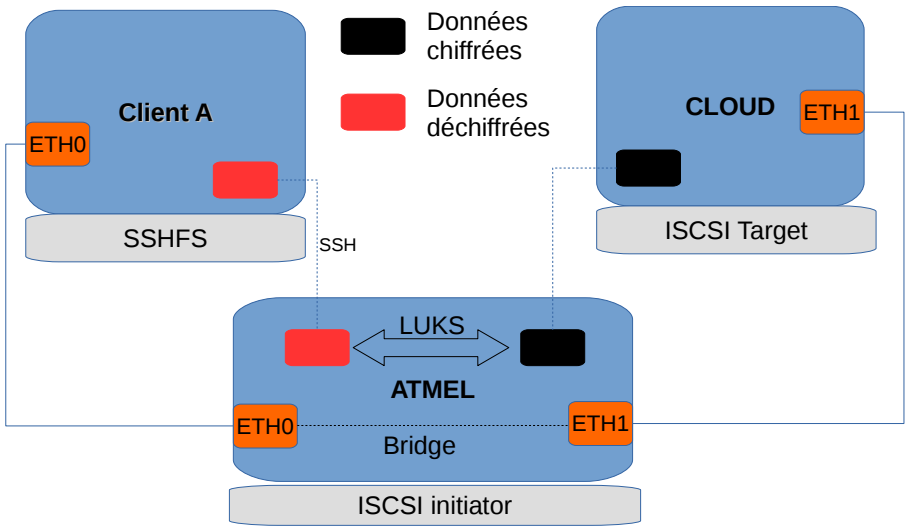


FIGURE 3.8 – Vue détaillée d'une solution Cloud personnel basé sur un module Atmel.

Master Key (MK) est générée aléatoirement. Elle est utilisée pour le chiffrement des données, et protégée par la KEK. (cf. annexe A).

4. *L'ouverture du conteneur :* le conteneur est ouvert avec la commande *cryptsetup luksOpen*. Il faut alors créer un système de fichier comme ext4. Pendant ces deux opérations, le module *dm-crypt* intervient en créant un périphérique virtuel qui se charge de chiffrer ou déchiffrer les blocs d'E/S.
5. *Le montage du périphérique virtuel :* le périphérique virtuel est monté dans un dossier du module afin d'accéder au contenu.
6. *L'accès au dossier à partir du poste client :* le dossier monté sur le module est accédé via le poste client par plusieurs protocoles (SFTP dans notre cas).

Les 3 entités (Client, Module et Cloud) sont présentes au sein du même réseau local, limitées à 100 Mbps (limite imposée par une des deux interfaces ethernet du Module).

Concernant les détails cryptographiques :

— **Le FDE :** est basé sur le mode de chiffrement XTS de l'algorithme AES. La taille de la clé était de 256 bits. Le module logiciel intégré au Linux Kernel Crypto API a été utilisé car le co-processeur de la carte ne supportait pas le mode XTS-AES. Ce mode s'appuie sur des valeurs intermédiaires appelés "tweak" afin de produire le texte chiffré. Chaque bloc est indépendant des autres. Seulement, deux blocs identiques ne généreront pas le même texte chiffré. Les motifs du choix de ce mode de chiffrement seront abordés en détail dans le chapitre 4.

— **Le protocole SFTP :** est basé sur le mode CTR-AES (avec une taille de clé de 128 bits) pour le chiffrement des messages et sur l'algorithme *umac-64-etm* pour l'authentification. Concernant le mode CTR, il est supporté par le co-processeur. Mais le protocole SFTP

appelé via la librairie OpenSSL n'était pas configuré pour l'utiliser. Nous n'avons donc pas pu modifier ce comportement dans le cadre de notre plateforme de test.

3.6.3 Résultats et performances

A la suite de l'implémentation de l'architecture TTM, nous avons défini au préalable huit configurations afin d'en analyser plus finement les performances. Notons que la carte Atmel possède un processeur mono-coeur ARM Cortex A5 alors que le poste client est équipé d'un processeur Intel Core 2 duo plus puissant. En outre, les configurations se distinguent au niveau du protocole de transfert choisi et de la présence ou non de FDE. De cela, découle trois variables :

- **FDE** : signifie que le FDE a été mis en place à l'aide de LUKS / dm-crypt soit dans le module soit sur le Client.
- **Le protocole de transfert iSCSI :** correspond à un transfert de fichier au niveau du bloc.
- **sshFs :** signifie que le transfert de fichier s'effectue via le protocole réseau SSH FTP (SFTP).

En résumé, le protocole de test consiste à copier des fichiers sur le Cloud et mesurer le temps de transfert pour y accéder soit à partir du Module ou du Client. Sans oublier que le chiffrement dépend de la configuration choisie (cf. Tableau 3.2).

Nous avons distingué deux groupes :

- **Les configurations avec la carte Atmel :** ce groupe correspond à toutes les configurations (Tableau 3.2) faisant intervenir la carte Atmel. La première *configuration 0* permet de rendre compte les performances maximales à atteindre avec un conteneur chiffré dans la mémoire de l'Atmel alors que la *configuration 1* se rapproche de l'architecture TTM. La seule différence se situe au niveau du chiffrement en transit qui est effectué entre le Client et le Module au lieu d'avoir lieu entre le Module et le Cloud. Nous déduisons au bout du compte que les autres configurations ne comportent pas de Client.
- **Les configuration avec le Client :** ce groupe correspond à toutes les configurations (cf. Tableau 3.3) avec un Client mais sans Module (sauf la configuration 1 correspondant à l'architecture TTM complète).

Analysons les résultats obtenus dans la Figure 3.9 dans l'ordre des configurations.

La *configuration 0* permet de donner une mesure de référence sur les performances maximales que l'on peut atteindre. Son chiffrement et son accès aux conteneurs sont effectués en local. Le débit de déchiffrement atteint est d'environ 4 MiB/s quelque soit la taille du fichier. A ce propos, nous verrons que ce débit pourrait être amélioré grâce à une optimisation présentée dans le chapitre 4.

TABLE 3.2 – Détails des différentes configurations avec l'Atmel.

Configuration	**Caractéristiques**
Toutes les configurations	Mode de chiff. pour FDE : aes-xts-256 Mode de chiff. pour sshFs : aes-ctr-128 + umac-64-etm Débit réseau : 100 Mbps CPU client : Core 2 Duo CPU P9400 @ 2.40GHz CPU Atmel : Cortex A5 @ 536 MHz
Config 0 : local	Atmel (FDE)
Config 1 : TTM	Client ⟷ Atmel ⟷ Cloud sshFs (FDE) iSCSI
Config 2	Client ⟷ Atmel ⟷ Cloud sshFs iSCSI
Config 3	Atmel ⟷ Cloud sshFs
Config 4	Atmel ⟷ Cloud iSCSI
Config 5	Atmel ⟷ Cloud (FDE) iSCSI

La *configuration 1* correspond à l'architecture TTM que nous proposons. Le débit de déchiffrement obtenu est de 1.8 MiB/s (16 Mbit/s). Ce débit est estimé suffisant dans le cadre d'une utilisation personnel avec une connexion internet ADSL. Toutefois insuffisant dans le cadre d'une connexion fibre optique où les débits peuvent atteindre 100 Mbit/s. Il est aussi important de souligner que nous avons intégré uniquement un chiffrement en transit entre le Client et le Module. Or, dans une configuration plus réaliste le chiffrement en transit devrait être mis en place entre le Module et le Cloud. Ceci, dans le but d'encapsuler les trames réseaux et empêcher des attaques de rejeux ou une analyse de ces dernières. Le protocole réseau IPSEC pourrait donc être ajouté au sein de la connexion iSCSI. Nous pouvons par contre estimer le débit obtenu (1,8 MiB/s) comme étant similaire au cas idéal puisque dans les deux cas un seul chiffrement en transit sera effectué dans le module. Remarque : ce résultat de 1.8 MiB/s est à nuancer car nous n'avons pas pu exploiter pleinement le co-processeur cryptographique, qui nécessitait l'utilisation de deux chiffrements.

La configuration 2 permet de mettre en évidence le surcoût du FDE, puisque le conteneur, stocké sur le Cloud et accédé via iSCSI par le Module Atmel, n'est pas chiffré. De ce fait un gain de 25 % est identifié et le débit passe de 1.8 MiB/s à 2.25 MiB/s.

Avec la *configuration 3* on obtient un débit de 3.8 MiB/s. Cela découle de l'accès direct entre l'Atmel et le Cloud via le protocole SFTP.

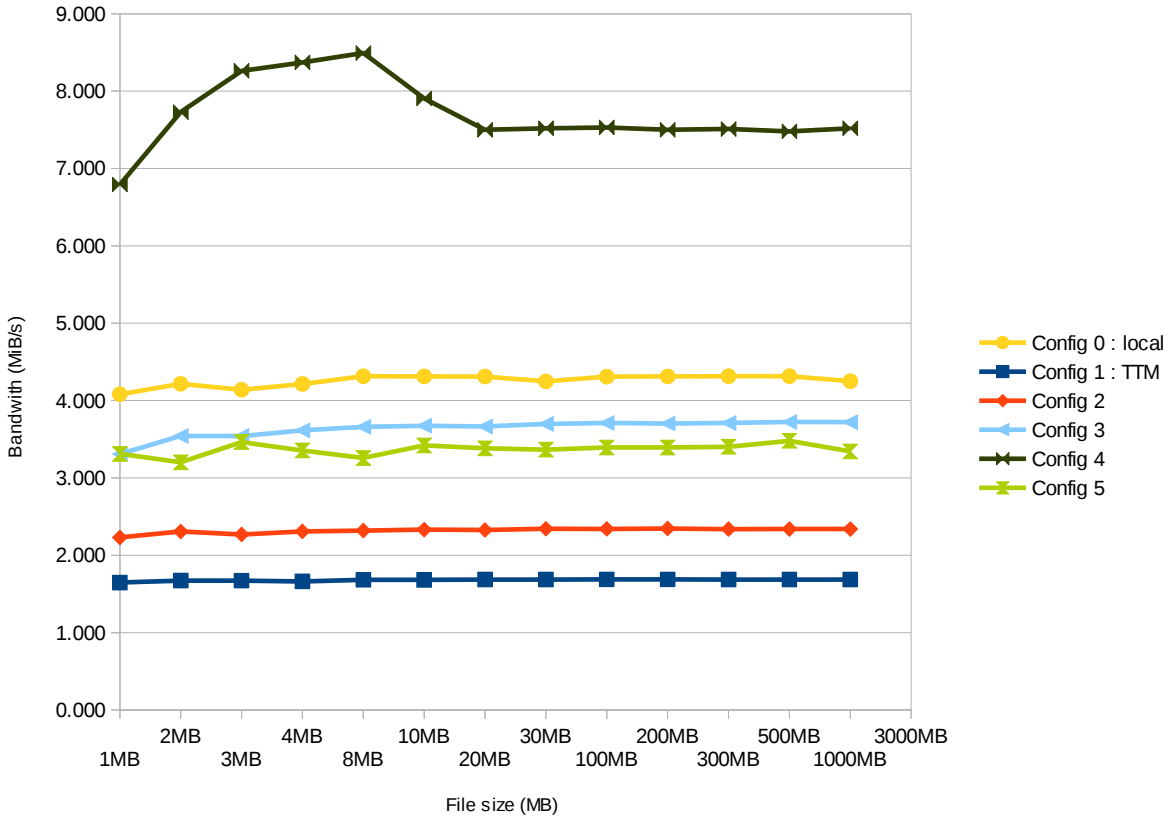


FIGURE 3.9 – Débit de transfert avec ou sans chiffrement pour différentes configurations avec l'Atmel.

La *configuration 4* met à nu la performance du protocole iSCSI. Son débit débit atteint 7,6 MB/s, une performance proche de la limite théorique (100 Mbps).

La *configuration 5* ré-intègre le FDE parallèlement à la connexion iSCSI. Les fichiers sont par contre accédés directement sur l'Atmel et non le Client. Ainsi, le débit obtenu (3,5 MiB/s) est proche de la configuration 3 (3,8 MiB/s) avec uniquement le protocole SFTP qui intègre nativement un chiffrement supplémentaire.

Le deuxième graphique 3.10 met en évidence l'impact de la puissance de calcul du processeur. Quelque soit la configuration (iSCSI, sshFs, iSCSI et FDE), les débits de transfert avec ou sans chiffrement avoisinent les limites du réseau (100 Mbit/s) avec une stabilisation à 7.8 MB/s. Ainsi, l'élément limitant dans l'architecture TTM est clairement la faible puissance de calcul du processeur de l'Atmel. Notons au passage un biais sur le graphique 3.10. En effet, on observe des pics de débits à 9 Mib/s qui s'explique par la présence de mémoire cache.

TABLE 3.3 – Détails des différentes configurations avec le client.

Configuration	Caractéristiques
Toutes les configurations	Mode de chiff. pour FDE : aes-xts-256 Mode de chiff. pour sshFs : aes-ctr-128 + umac-64-etm Débit réseau : 100 Mbps CPU client : Core 2 Duo CPU P9400 @ 2.40GHz CPU Atmel : Cortex A5 @ 536 MHz
Config 1 : TTM	Client ⟷ Atmel ⟷ Cloud sshFs (FDE) iSCSI
Config 6	Client ⟷ Cloud iSCSI
Config 7	Client ⟷ Cloud sshFs
Config 8	Client ⟷ Cloud (FDE) iSCSI

3.7 Discussion

Cette preuve de concept nous a permis de mettre au point une première version fonctionnelle d'un Cloud personnel basé sur le principe de TTM. Certaines exigences sont respectées. D'une part les données sont conservées uniquement sur le Cloud. D'autre part, le Module concentrent toutes les opérations cryptographiques. Enfin, le Client accède uniquement à une vue des fichiers en clair.

Contrairement à la majorité des systèmes Cloud existant, nous avons choisi un chiffrement au niveau bloc au sein des conteneurs grâce à l'approche FDE. Par ce biais, tout le contenu est protégé. Deux principes guident notre architecture :

— **Une distribution transparente des données** est gérée par le protocole réseau *iSCSI*, qui permet le transfert des données au niveau bloc à des débits élevés.

— **Un chiffrement transparent** s'appuyant sur le module noyau Linux *dm-crypt* et sur le logiciel de configuration *cryptsetup* tout en respectant le format *LUKS*. La flexibilité du module noyau *dm-crypt* est un avantage considérable pour les opérations cryptographiques. Elles se basent soit sur les algorithmes logiciels du Linux Crypto API, soit sur des implémentations tirant profit de matériels dédiés comme les co-processeurs cryptographiques.

Il est important de toujours sélectionner le module en fonction de tous les algorithmes de chiffrement nécessaires et les co-processeurs cryptographiques qui les supportent. En effet, les résultats présentés dans la section 3.6.3 montrent que lorsque l'on intègre le chiffrement XTS-AES pour le FDE et un chiffrement en transit entre le module et le client ou entre le module et le Cloud, les performances globales baissent significativement. Cette baisse s'explique dans notre

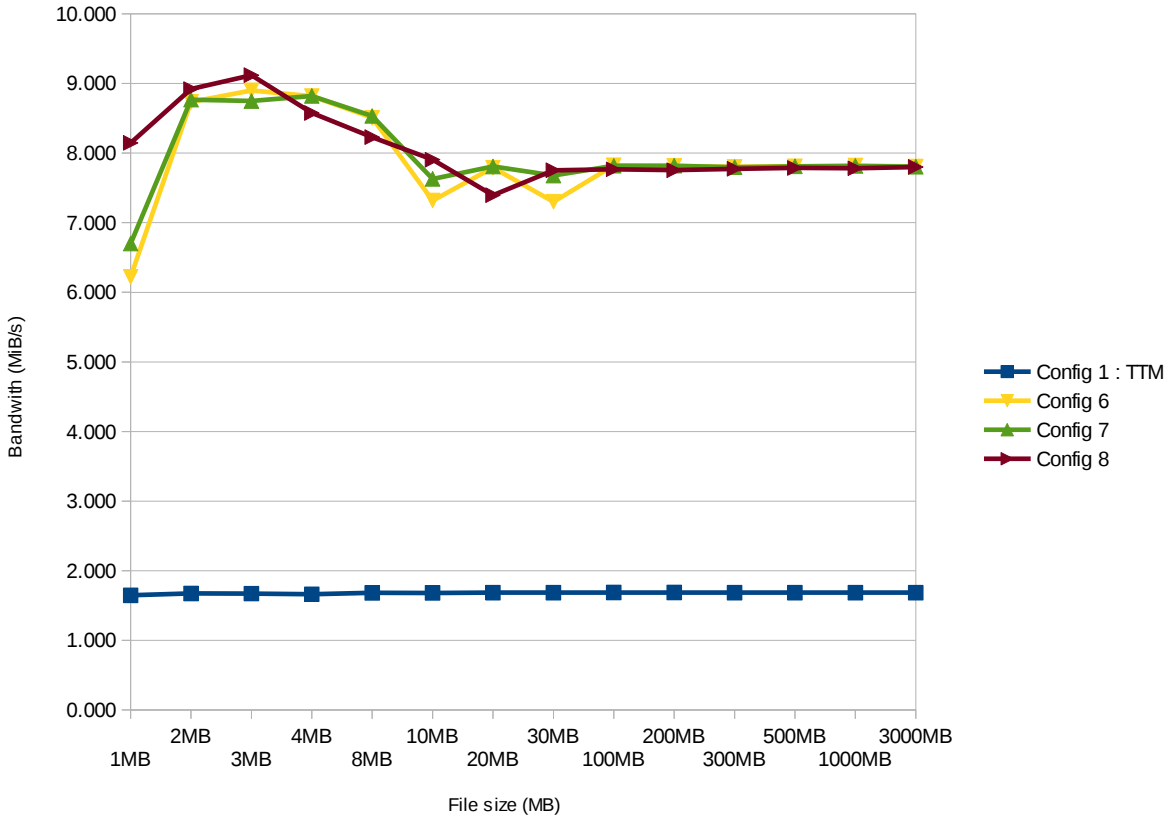


FIGURE 3.10 – Débit de transfert avec ou sans chiffrement pour différentes configurations avec le client.

preuve de concept par une surcharge du CPU puisque la première carte ATMEL SAMA5D3 utilisée par notre démonstrateur supporte divers modes de l'AES mais pas le mode XTS.

Nous allons voir dans le chapitre 4 que malgré le manque du support natif du mode XTS-AES par le co-processeur, nous pouvons significativement augmenter les performances. Pour remplacer XTS-AES, nous avons testé le mode CBC-AES avec le FDE en local sur la carte Atmel. Afin d'observer le gain possible, nous avons comparé l'implémentation logicielle et matérielle (i.e. basée sur le co-processeur). Nous avons obtenu un débit de chiffrement et déchiffrement qui passe de 4.2 à 7.8 MiB/s. En d'autres termes, les performances ont été doublées, ce qui montre le gain potentiel et la viabilité de notre solution.

Par ailleurs, l'architecture TTM préserve la confidentialité des données face à un hébergeur curieux et un attaquant. Toutefois, elle ne protège pas de ces deux adversaires en cas de modification du texte chiffré. Cette notion d'intégrité sera analysée dans le chapitre 5.

Certains points de sécurité supplémentaires sont à prendre en considération. Le premier est que dans l'étape de création du conteneur dans le serveur de stockage, ce dernier doit être initialisé non pas avec des constantes (souvent 0 à l'aide de la commande dd et l'argument /dev/zero) mais avec des nombres aléatoires. Ceci empêchera le Cloud de déterminer par exemple le taux

d'occupation du conteneur.

Le second point concerne la sécurité et la protection de l'accès au conteneur via le protocole iSCSI dont l'accès par défaut est public. Puisque la connexion s'effectue en deux étapes, il faut au préalable exécuter une commande de découverte des conteneurs disponibles (targets) sur le serveur de stockage avant de s'y connecter. Il est essentiel de sécuriser ces deux étapes :

- **La découverte des targets :** il est fortement recommandé de limiter la découverte des targets à certaines adresses IP (bien que des attaques de spoofing d'adresses existent). Aussi, une authentification des initiateurs doit être exigée lors de la découverte des targets. Ces deux dispositifs empêcheront l'attaquant d'exploiter les identifiants.
- **La connexion à la target :** une authentification CHAP peut être mise en place afin de protéger l'accès au conteneur.

Un dernier point touche à la gestion des clés. En effet, toute la sécurité de l'architecture repose sur la robustesse du mot de passe ou de la passphrase entré par l'utilisateur. C'est une limite fondamentale à prendre en compte. Une autre solution serait de générer une clé forte et de la conserver sur un support de stockage externe. Ce genre de solution pourrait être étudiée dans des travaux futurs.

Chapitre 4

Amélioration des débits de chiffrement/déchiffrement sous Linux du mode XTS-AES avec extReq

Sommaire

4.1 Introduction

Le FDE se base sur un chiffrement symétrique afin de protéger les données. La sécurité mise en place repose sur une clé de chiffrement utilisée pour le chiffrement et le déchiffrement de grandes quantités de données (les conteneurs peuvent avoir une taille dépassant le téra-octet). Cette quantité importante engendre une contrainte forte concernant le débit de chiffrement. En outre, le mode de chiffrement préconisé pour le FDE est le mode XTS-AES qui est un mode intrinsèquement coûteux.

Dans ce chapitre, le fonctionnement de ce mode va être décrit. Puis, nous allons donner des détails sur notre nouvelle implémentation sur la carte SAMA5D3 tirant profit du co-processeur cryptographique bien qu'il ne supporte pas le mode XTS-AES. Enfin, nous discuterons de notre optimisation *extReq* dédié au FDE sous Linux.

4.2 Le mode XTS-AES et son implémentation hybride.

4.2.1 Le choix du mode XTS-AES pour le FDE

Dans le FDE sous Linux, le logiciel cryptsetup avait comme paramètre par défaut, jusqu'à la version 1.6.0, [1](2013) l'algorithme de chiffrement AES avec le mode CBC. Puis, le mode XTS-AES a été choisi. Une des raisons de ce choix est le gain en débit de chiffrement. En effet, ce mode est dédié au chiffrement de disque, et contrairement au mode AES-CBC, les blocs de 16 octets sont indépendants les uns des autres au sein d'un secteur de 512 octets. Par conséquent, les opérations peuvent être effectuées dans un ordre quelconque et être parallélisées. Certaines implémentations (e.g. driver aes-ni d'Intel[AHT13]) ont profité de cette possibilité pour accélérer significativement les performances de chiffrement et déchiffrement. Pour preuve, les processeurs Intel i7 permettent d'obtenir un débit de chiffrement / déchiffrement de plus de 1.5GB/s lors de test en mémoire RAM (résultats obtenus à l'aide de la commande *cryptsetup benchmark*).

Description

XTS-AES est un mode standard pour les disques de stockages orientés bloc depuis 2007 [IEE08]. La taille d'un secteur traité par ce mode correspond à celle des secteurs des disques de stockage traditionnels de 512 octets. Chaque secteur peut être traité indépendamment des autres car le vecteur d'initialisation (IV) correspond à l'index du bloc de 512 octets au sein du conteneur.

1. https ://gitlab.com/cryptsetup/cryptsetup/blob/master/docs/v1.6.1-ReleaseNotes

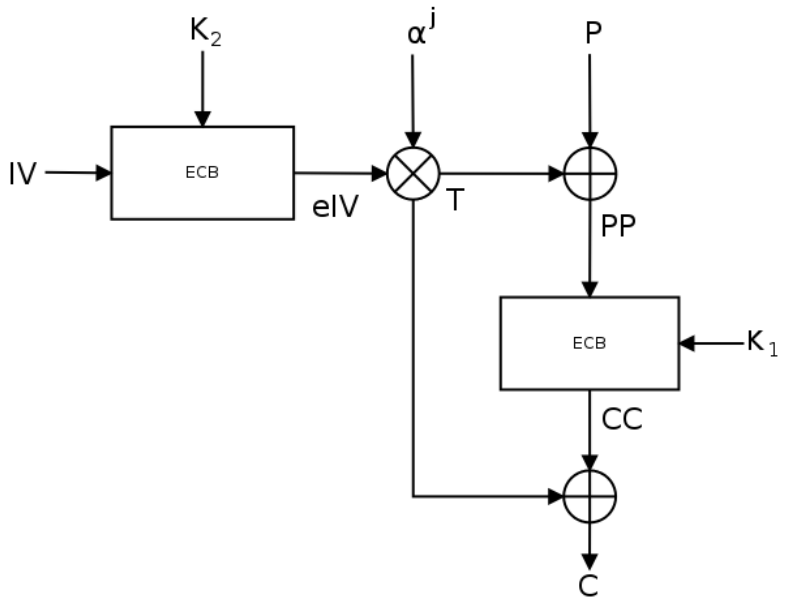


FIGURE 4.1 – Chiffrement XTS-AES de la j^{ime} unité de 128 bits d'un bloc de 512 octets.

Considérons le chiffrement XTS-AES (le déchiffrement peut-être étudié dans ces articles [Mar10, IEE08]). L'algorithme prend 3 paramètres en entrée :

- **La clé,** K**,** de taille 256 ou 512 bits est partagée en deux sous-clés de taille identique, K_1 et K_2. K_1 est utilisée pour dé/chiffrer les données alors que K_2 est utilisée pour chiffrer l'IV.
- **Le vecteur d'initialisation,** IV**,** a pour taille 128 bits/16 octets et représente le numéro de secteur (i.e., l'index au sein du disque).
- **Le texte en clair** P a une taille de 512 octets et constitue le texte à chiffrer.

Considérons un bloc de 512 octets. Il est composé de 32 unités de 128 bits/16 octets. Soit j l'index de l'unité au sein du bloc. La figure 4.1 montre le chiffrement pour cette unité. La première étape consiste à chiffrer l'IV avec la clé K_2 en utilisant le mode ECB-AES pour générer eIV. Le résultat est multiplié dans le corps de Galois avec α^j pour produire le tweak, T, où α est l'élément primitif de GF(2^{128}). Puis les 128 bits de données sont `XOR`és avec T, chiffrés avec le mode ECB-AES et la clé K_1, produisant CC. La dernière étape consiste à `XOR`er CC avec T, afin de produire le chiffré C pour cette unité de 128 bits. Les même opérations sont exécutées pour tous les blocs de 128 bits, successivement.

4.2.2 Implémentations du mode XTS-AES

L'implémentation des algorithmes de chiffrement peut être effectuée à plusieurs niveaux. Au niveau de l'espace utilisateur, nous trouvons les librairies *openSSL[ope17]*, *GnuTLS*[gnu17] ou *Gcrypt[gcr17]*. L'avantage réside dans l'utilisation des fonctions sans droits administrateurs. Les

autres implémentations sont exécutées dans l'espace noyau. Nous rappelons qu'il existe trois types d'implémentations dans cet espace :

— les implémentations 100 % logicielles ;
— les implémentations reposant sur des instructions matérielles spécifiques comme AES-NI [Gue12] pour les processeur Intel, ou "ARMv8 Crypto Extensions" pour les processeurs ARM ;
— les implémentations reposant sur un accélérateur matériel (co-processeur cryptographique).

4.2.3 Implémentation de la version hybride sur la carte Atmel SAMA5D3

Dans le cadre du démonstrateur du chapitre 3 Cloud personnel, seule l'implémentation logicielle du mode XTS-AES avait été utilisée. Nous avions constaté dans les résultats des performances limitées. Pour rappel, la carte Atmel SAMA5D3 supporte les 5 modes usuels de l'algorithme AES mais pas le mode XTS-AES. Cependant, il est possible d'implémenter un mode hybride basé en partie sur l'implémentation logicielle et sur le co-processeur cryptographique.

Le mode XTS-AES est en effet découpé en 5 opérations :

1. **Le chiffrement ECB-AES de l'IV :** l'IV est chiffré avec la clé K_2 pour obtenir eIV.
2. **Le calcul du tweak :** à partir de eIV, le tweak T est calculé à l'aide de la multiplication dans le corps de Galois.
3. **Le premier** XOR **:** le texte en clair P est `XOR`é avec T pour obtenir PP.
4. **Le chiffrement ECB-AES :** PP est chiffré avec la clé K_1 pour obtenir CC.
5. **Le second** XOR **:** CC est `XOR`é avec T pour obtenir le chiffré final C.

Dès lors, pour les cas où le co-processeur cryptographique supporte le mode ECB-AES mais non le mode XTS-AES, il est possible d'exécuter les opérations 2, 3 et 5 de manière logicielle et les opérations 1 et 4 avec le co-processeur.

L'algorithme 1 décrit ces étapes à l'aide de deux fonctions :

— **computeTweak(**IV**) :** calcul de 32 tweaks à partir de l'IV chiffré (eIV) et avec la multiplication dans le corps de Galois.
— **AESEncECB(**P**,** K**) :** chiffre du texte en clair P avec la clé K pour produire C. Cette fonction utilise le co-processeur cryptographique.

Algorithme 1 : Chiffrement XTS-AES , atmel_aes_xts_encrypt().

input : clé partagée en $(K_1 \parallel K_2)$;
IV (16B) ;
texte en clair P (512B) ;

output : chiffré C (512B)

```
1 eIV ← AESEncECB (IV, K1)
  // tw_buf est un tampon de 512 octets
  // contenant 32 tweaks de 16 octets
2 tw_buf ← computeTweak (eIV)
3 PP ← tw_buf ⊕ P
4 CC ← AESEncECB (PP, K2)
5 C ← tw_buf ⊕ CC
```

L'implémentation de ce mode au sein du driver *atmel-aes* a engendré de nombreuses difficultés. Initialement prévu pour l'envoi d'une seule requête cryptographique, nous avons modifié l'implémentation du driver afin de décomposer la requête globale en deux sous-requêtes : une pour le chiffrement de l'IV, une pour le chiffrement / déchiffrement du texte. De nombreuses instabilités liées au comportement asynchrone du driver ont rendu la mise au point difficile.

4.2.4 Evaluation de performances de la version hybride

Le tableau 4.1 montre les débits obtenus avec l'implémentation hybride du mode XTS-AES (logiciel + co-processeur cryptographique ECB-AES). On constate alors que les résultats de la version hybride sont inférieurs à ceux de la version purement logicielle (4.5 MB/s contre 5.0 MB/s). Les causes de cette baisse de performance s'expliquent par le surcoût des requêtes cryptographiques via le DMA. Effectivement, le temps nécessaire pour une requête étant constant, ce dernier ne peut être compensé que si les données en entrée ont une taille suffisamment grande.

Ces résultats décevants nous ont conduit à analyser plus en détails le fonctionnement à la fois du module *dm-crypt* et du driver atmel-aes. Nous allons voir dans la section suivante qu'une optimisation est possible.

TABLE 4.1 – Débit de chiffrement pour la carte SAMA5D3 avec le mode XTS-AES.

Type implémentation	**Débit chiffrement (MB/s)**
100 % logiciel	5.0
hybride	4.5

4.3 ExtReq : étendre les requêtes cryptographiques à la taille d'une page

4.3.1 Fonctionnement original

Lorsque le noyau décide de transférer un ensemble de secteurs depuis/vers un disque de stockage, il utilise une structure *bio* pour gérer cette opération. Afin de transmettre ces données dans le driver bas-niveau une autre structure intervient : *scatterlist*.

Zoom sur les structures de données *bio* et *scatterlist* sous Linux

Les données au niveau des blocs d'entrées et sorties (E/S) sont d'abord regroupées dans une liste de structure bio comme indiqué sur la figure 4.2. Chaque *bio* pointe vers un tableau de segments. Chaque segment est un vecteur de la forme <page, offset, longueur> où la page correspond à la page physique associée. Chaque page correspondant à une zone contiguë en mémoire alors que ce n'est pas nécessairement le cas pour un segment complet. La structure de données *bio* permet de faire des opération d'E/S au niveau bloc vu comme un unique tampon provenant de multiples emplacements en mémoire. L'agencement entre les structures est géré grâce à deux pointeurs : l'élément *bio_io_vec* pointe vers le premier segment alors que l'élément bi_next pointe vers la structure *bio* suivante. Ainsi les blocs d'E/S peuvent être représentés par une liste chaînée de structures *bio*.

Pour obtenir des performances élevées, l'utilisation de Direct Memory Access (DMA) est nécessaire. Le DMA permet au périphériques d'E/S de transférer des données depuis/vers la mémoire sans intervention du CPU via des bus spécifiques. Procéder alors à des opération de transfert regroupées nécessite d'avoir une représentation des données similaires à la structure *bio* mais avec quelques champs supplémentaires. Cette nouvelle structure est la scatterlist. Chaque structure (appelée segment) comporte le même vecteur composé de 3 champs <page, offset, longueur> mais avec (parmi d'autres) un élément *dma_address* supplémentaire correspondant à l'adresse du DMA. Initialement, la structure scatterlist était limité à un tableau de segment que l'on appelle (sg_list). Le nombre maximum de segment était contraint par la taille d'une page selon la formule suivante :

$$Nombre\_segments = PAGE\_SIZE \ / \ sizeof(struct\ scatterlist)$$

Une scatterlist était limitée de cette manière à une taille inférieur à 1Mo. C'est pourquoi une évolution à permis de transformer les scatterlists simples en scatterlist chaînées (4.3) à l'aide d'un ajout de pointeur pour le dernier segment. Une *sg_table* peut finalement être composée d'une liste chaînée de *sg_list* pour des tailles importantes d'E/S.

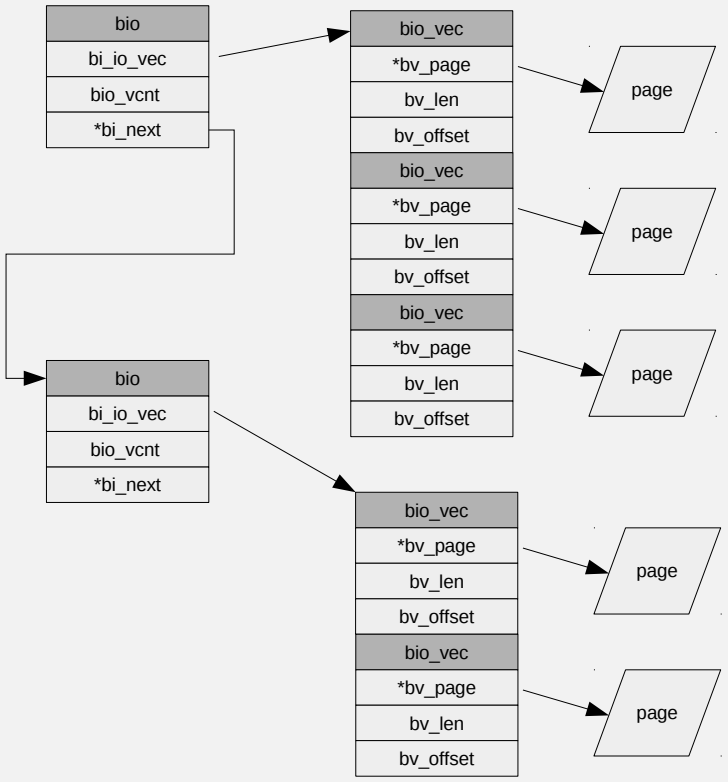


FIGURE 4.2 – *bio* structure

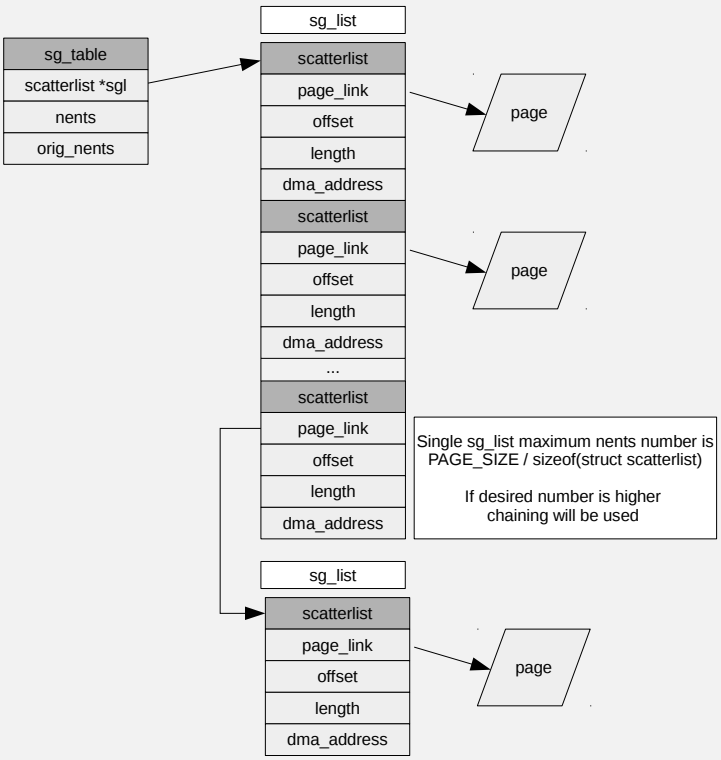


FIGURE 4.3 – *scatterlist* structure

Le module chargé du lien entre ces deux structures est *dm-crypt*. La figure 4.4 illustre le transfert

des données de la structure *bio* vers la structure *scatterlist*. L'unité commune entre ces deux entités est la taille d'une page (dans les descriptions du zoom, chaque page a été vue comme un segment). La première étape est de récupérer la page représentée par une *bio_vec*. La deuxième étape consiste à créer 2 scatterlists (seulement une est représentée), une pour la source et une pour la destination avec chacune un seul segment (une page). Puis, à l'aide de la fonction *set_page*, la scatterlist source est initialisée avec la page pointée par *bio_vec* commençant à l'offset 0 et avec une taille de données de 512 octets. La troisième étape permet de créer la requête dm-crypt avec des informations concernant le dé/chiffrement et les scatterlists récemment créées pour chaque secteur de 512 octets. La dernière étape consiste à envoyer la requête cryptographique à l'atmel-aes driver afin d'être traitée. Ainsi, pour traiter une page de 4096 octets, les étapes 2,3 et 4 sont exécutées 8 fois, avec un offset incrémenté de 512 à chaque fois.

Une fois la page terminée, la page suivante est récupérée de la structure bio (étape 1) et envoyée à l'atmel driver de la même manière.

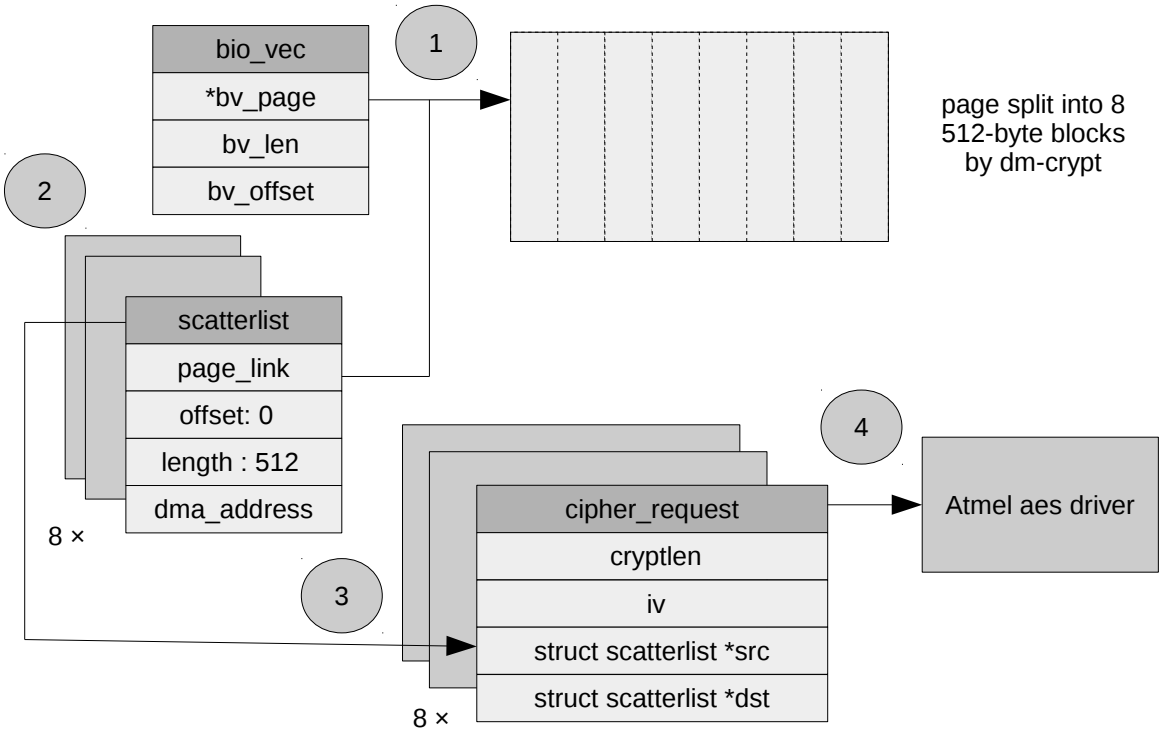


FIGURE 4.4 – De la structure bio vers la scatterlist

Algorithme 2 : Fonction origianale : crypt_convert().

input :

d'octets à transférer, nbytes ;
numéro de secteur initial, sec ;
taille du sector_shift , sec_shift ;
(sec_shift est codé en dur à 512B)
texte à chiffrer, in ;
texte chiffré,out ;

```
1 reste ← nbytes
2 while reste > 0 do
      // sec est utilisé comme IV ci-après
3     cryptConvertBlock (sec, sec_shift, in, out)
4     sec ← sec + 1
5     reste ← reste − sec_shift
```

Ce transfert au sein de dm-crypt est effectué via deux fonctions : *crypt_convert* et *crypt_convert_block*. L'alg. 2 illustre une version simplifiée de la première fonction.
Lorsque qu'une requête d'E/S est gérée par *dm-crypt*, cette dernière passe par la fonction *crypt_convert* qui parcourt le bio et chiffre / déchiffre les données associées. Le paramètre *reste* permet de comptabiliser la quantité de données restante à traiter depuis *in* vers *out*. *sector_shift* correspond à l'offset pour le secteur suivant et *req* est la requête créée par dm-crypt. Les scatterlists sont initialisées dans la fonction *crypt_convert_block* et associées à *req*. Puis la requête chiffrée / déchiffrée par l'algorithme spécifié (XTS-AES dans notre cas) est traitée. En d'autres termes, soit elle est stockée sur le disque soit disponible pour l'application (éditeur de texte, galerie photos ...).
L'inconvénient de cette implémentation est le découpage en blocs de 512 octets de la page par le biais du paramètre *sector_shift*.

4.3.2 L'optimisation extReq

Nous avons émis une hypothèse : les performances pourraient significativement augmenter si au lieu de travailler par secteur de 512 octets nous optons plutôt pour des secteurs de la taille d'une page. Il est cependant nécessaire, afin de conserver une compatibilité avec les systèmes existants et pour d'autres raisons comme le fait de traiter le dernier bloc d'une bio qui pourrait être inférieur à 4096 octets, de continuer à gérer aussi des secteurs de 512 octets.
Après avoir analysé en détail les structures de données au niveau bloc au sein du noyau Linux, nous allons maintenant voir l'implémentation de cette extension. Cette évolution est possible à condition de modifier deux entités :

— **le module dm-crypt :** afin de découper en secteur de taille plus important et envoyer la scatterlist.

— **le driver atmel-aes :** afin de recevoir des secteurs plus grands et gérer l'implémentation du mode XTS-AES.

Nous allons donc décomposer cette partie en deux sous parties, d'abord nous verrons la création de la scatterlist dans le module dm-crypt puis le traitement de cette scatterlist dans l'atmel driver.

4.3.3 Modification apportées à dm-crypt

Nous avons vu que dm-crypt recevait initialement des pages mémoires qui étaient par la suite découpées en secteurs de 512 octets. Nous allons donc modifier ce comportement et retirer le découpage. Cela engendre d'autres modifications quant à la gestion des IV. En effet, pour chaque secteur de 512 octets un IV est associé. Si nous envoyons une page alors il faudra incrémenter le compteur non pas de 1 mais de 8 (la page est composée de 8 secteurs). L'alg. 3 montre le fonctionnement simplifié de ces modifications. Les données sont, de cette façon, découpées en pages complètes. Pour chaque page, une *scatterlist* dédiée est initialisée puis envoyée au driver atmel-aes.

Algorithme 3 : Fonction crypt_convert() modifiée avec extReq.

input :
d'octets à transférer, nbytes ;
numéro de secteur initial, sec ;
taille du sector_shift , sec_shift ;
texte à chiffrer, in ;
texte chiffré, out ;

```
1 reste ← nbytes
2 while reste > 0 do
3     if reste > 4096 then
4         sec_shift ← 4096
5     else
6         sec_shift ← reste
      // sec est utilisé comme IV ci-après
7     cryptConvertBlock (sec, sec_shift, in, out)
8     sec ← sec + sec_shift/512
9     reste ← reste − sec_shift
```

4.3.4 Modifications apportées à l'atmel-aes driver

L'alg. 4 illustre les modifications apportées au driver atmel-aes. Une fois la requête reçue par le driver, nous devons récupérer le nombre *nblocks* de secteur de 512 octets à l'intérieur de la scatterlist. A ce titre, en guise de rappel, le chiffrement XTS-AES d'après le standard se

limite à traiter des blocs de taille maximum 512 octets. Une fois *nblocks* déterminé, la fonction *generateIVs* génère les IVs en incrémentant de 1 chaque IV précédent.
Nous avons aussi, pour le cas d'une page complète, 8 IVs. Cet ensemble sera chiffré (ligne 3) avec la clé K_2. Cette opération peut être vue comme un seul appel au mode ECB-AES. Ensuite, un tampon contenant toutes les valeurs de tweaks est pré-calculé à partir des IV chiffrés (ligne 4,5). Ce tampon contient 8×32 tweaks. Enfin, s'ajoutent les 3 dernières opérations correspondant au [XOR]-[ECB-AES]-[XOR]. Le gain maximum dans l'atmel-aes est attendu dans l'opération de chiffrement ECB-AES. En conséquence, au lieu d'exécuter 8 appels via les DMA, un seul appel de taille 4096 octets est effectué.

Algorithme 4 : Fonction atmel_aes_xts_encrypt() modifiée avec extReq.

input : clé découpée en $(K_1 \parallel K_2)$;
IV (16B) ;
texte à chiffrer P (multiple de 512B) ;
output : texte chiffré C

```
1 nblocks ← sizeof (P)/ 512
2 IVs ← generateIVs (IV, nblocks)
3 eIVs ← AESEncECB (IVs, K2)
  // tw_buf est un buffer
  // contenant 32 × nblocks
  // tweaks de 16B
4 for i ← 0 to nblocks do
5 |  tw_buf [i × 512] ← computeTweak (eIVs [i × 16])
6 PP ← tw_buf ⊕ P
7 CC ← AESEncECB (PP, K1)
8 C ← tw_buf ⊕ CC
```

4.4 Plateforme expérimentale

Nous avons implémenté toutes les versions décrites précédemment. Pour la mesure des performances, nous avons eu recours aux cartes SAMA5D3 et SAMA5D2 d'Atmel. Ces deux cartes possèdent un processeur mono-coeur ARM Cortex A5 de fréquence 500 MHz, avec un co-processeur cryptographique. Le co-processeur du SAMA5D2, contrairement au SAMA5D3, supporte les cinq modes communs de l'AES et le mode XTS-AES. Les deux cartes possèdent une mémoire vive de 256MB. Un système d'exploitation Debian est exécuté avec le noyau Linux 4.6 sur une carte SD Sandisk Classe 10. Durant tous les tests, nous avons utilisé la taille de clé par défaut de *dm-crypt*, à savoir une clé principale de 256 bits, partagée en 2 sous clés de 128 bits. Ainsi le chiffrement ECB-AES-128 a toujours été utilisé.

Voici un descriptif des différentes versions que nous avons employées :

SAMA5D3 :

- **la version logicielle :** est basée sur le module noyau "xts.ko" ;
- **la version hybride, avec le co-processeur ECB-AES, sans *extReq* :** le driver *atmel-aes* est modifié afin d'utiliser le co-processeur pour les opérations ECB-AES et le CPU pour les autres opérations, le tout avec des requêtes de 512 octets ;
- **la version hybride, avec le co-processeur ECB-AES, avec *extReq* :** identique au précédent mais avec des requêtes de 4kB (page complète) ;

SAMA5D2 :

- **la version logicielle :** est basée sur le module noyau "xts.ko" ;
- **la version avec le co-processeur XTS-AES :** le co-processeur cryptographique est utilisé pour toutes les opérations, sans l'optimisation *extReq*.

Durant l'étude, les deux configurations purement logicielles nous ont permis de calibrer les cartes Atmel. Ayant les mêmes spécifications, nous confirmons dans le section 4.4.3 que les deux cartes révèlent des performances similaires pour le mode purement logiciel. En somme, les résultats obtenus avec la carte SAMA5D2 peuvent être comparés à ceux de la carte SAMA5D3. La seule différence est le co-processeur cryptographique et non les autres caractéristiques matérielles.

4.4.1 Les avantages d'utiliser *extReq* au sein du driver atmel-aes

Nous allons dans cette partie analyser l'implémentation hybride utilisant le mode ECB-AES du crypto-processeur de la carte SAMA5D3 avec et sans *extReq*. Nous avons mesuré le temps passé dans chaque opération du mode XTS-AES pendant le chiffrement et le déchiffrement d'un fichier de 50 MB. Pour cela, nous avons utilisé la fonction *getnstimeofday* de la bibliothèque *time.h*. La précision a été déterminée en mesurant le temps entre deux appels consécutifs. Nous avons alors obtenu une valeur moyenne de 330 nanosecondes. Nous avons procédé en cinq étapes :

- **La mesure globale :** nous mesurons le temps lors de l'entrée dans le driver *atmel-aes* jusqu'à la sortie de ce dernier.
- **Le calcul des tweaks :** nous démarrons la mesure juste avant l'appel de la fonction de calcul des tweaks et l'arrêtons juste après.
- **Le premier XOR :** nous démarrons la mesure juste avant l'appel de la fonction XOR et l'arrêtons juste après.
- **Le second XOR :** nous démarrons la mesure juste avant l'appel de la fonction XOR et l'arrêtons juste après.
- **La mesure DMA + dé/chiffrement :** nous démarrons la mesure juste avant l'appel DMA, nous l'arrêtons après le chiffrement ou déchiffrement.
- **Autres :** ce temps est la différence entre le temps total et la somme des 4 mesures précédentes.

Il est important de préciser que ces mesures ont été répétées durant tout le chiffrement et

déchiffrement d'un fichier de 50 MB. Les résultats finaux en secondes sont les sommes de toutes les mesures pour chaque catégorie.

	Sans extReq		Avec extReq	
	Temps (s)	%	Temps (s)	%
Temps total	9.09	100.00	5.23	100.00
Calcul des tweaks	**2.62**	**28.90**	**1.70**	**32.61**
Premier XOR	0.31	3.41	0.31	6.08
Second XOR	0.63	6.99	0.41	7.86
DMA + chiffrement	**5.31**	**58.46**	**2.75**	**52.72**
Autres	0.20	2.24	0.03	0.73

TABLE 4.2 – Décomposition du temps pour le chiffrement au sein du driver *atmel-aes* d'un fichier de 50MB avec l'implémentation utilisant le co-processeur ECB-AES, sans et avec extReq.

	Sans extReq		Avec extReq	
	Temps (s)	%	Temps (s)	%
Temps total	9.30	100.00	5.13	100.00
Calcul des tweaks	**3.05**	**32.78**	**1.46**	**28.61**
Premier XOR	0.29	3.21	0.27	5.32
Second XOR	0.55	5.98	0.40	7.92
DMA + déchiffrement	**5.22**	**56.12**	**2.81**	**54.83**
Autres	0.17	1.90	0.17	3.32

TABLE 4.3 – Décomposition du temps pour le déchiffrement au sein du driver *atmel-aes* d'un fichier de 50MB avec l'implémentation utilisant le co-processeur ECB-AES, sans et avec extReq.

Analysons d'abord le chiffrement du fichier de 50 MB sans extReq. D'après le tableau 4.2, nous pouvons remarquer que les opérations DMA plus chiffrement occupent plus de la moitié du temps total dans la configuration par défaut, avec des requêtes de 512 octets : 5.31s parmi 9.09s. La seconde opération la plus longue est le calcul des tweaks avec un total de 2.62s. Les deux opérations représentent plus de 87% du temps total.
Avec extReq, le temps total est réduit d'un facteur 1.74 (5.23s au lieu de 9.09s). Les opérations DMA plus chiffrement représentent toujours la moité du temps total, mais ils correspondent plus qu'à 2.75s au lieu de 5.31s. Le temps de calcul des tweaks est aussi significativement diminué, il est de 1.70s au lieu de 2.62s. Ces constatations sont sensiblement identiques pour le déchiffrement. Retenons au final que l'impact de l'utilisation de requêtes étendues est considérable, le coût de mise en place des requêtes et des multiples transferts de données avec le co-processeur étant largement réduit.

4.4.2 Les avantages d'utiliser *extReq* sur le temps global de traitement

Nous allons maintenant considérer le temps global. Il inclut le traitement *dm-crypt*, les opérations d'E/S et tous les appels systèmes/surcoûts des traitements noyaux. Nous souhaitons

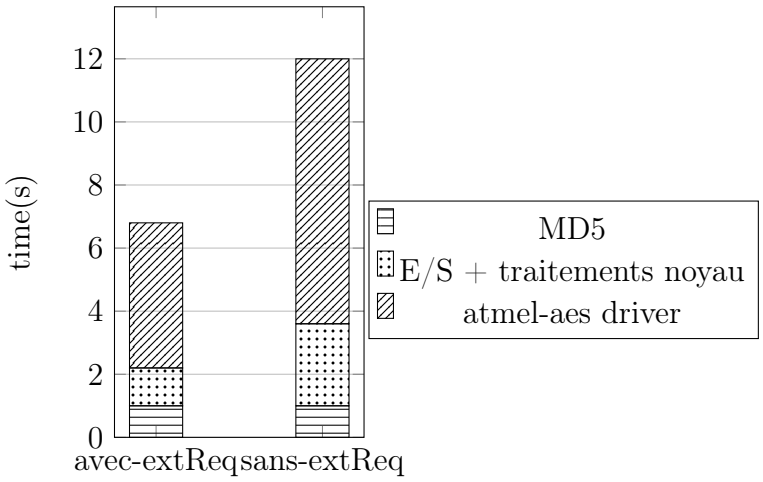


FIGURE 4.5 – Décomposition du temps global pour le chiffrement d'un fichier de 50MB avec l'implémentation utilisant le co-processeur ECB-AES, avec et sans extReq.

mettre en exergue les améliorations potentielles d'*extReq* sur le temps global, au-delà des avantages pour le driver atmel-aes lui-même.
Le temps total de chiffrement et déchiffrement est difficilement mesurable en raison de l'asynchronisme et de la présence de mémoires caches. Afin de surmonter ces difficultés, nous avons mesuré les temps de calcul des condensés (MD5) de fichiers chiffrés, i.e, le temps nécessaire pour déchiffrer et calculer le condensé MD5. Par conséquent, le temps total est composé du temps passé dans le driver *atmel-aes* (ligne 1 du tableau 4.3), du temps du MD5 qui est constant, du temps d'E/S et du temps de traitement noyau. Nous obtenons ainsi :

$$t_{total} = t_{md5} + t_{E/S\_et\_traitements\_noyau} + t_{atmel\_aes}$$

Dans cette analyse, nous nous focalisons sur l'implémentation hybride avec le co-processeur ECB-AES afin de mettre en évidence les améliorations d'extReq. La figure 4.5 montre la décomposition du temps global. Le temps de calcul du condensé est constant et relativement faible au regard du temps total sur les deux histogrammes. Ce qui, somme toute, confirme que les résultats ne sont pas biaisés par ce calcul. Sans surprise, le temps passé pour le déchiffrement dans l'atmel driver représente la majorité du temps global. Ce dernier est réduit de manière significative par l'usage de *extReq* comme vu précédemment. Il est à noter que le temps *d'E/S + traitements noyau* est aussi divisé par un facteur deux : ce qui constitue un avantage additionnel du mode extReq.

4.4.3 Comparaison de performance pour toutes les configurations

Jusqu'à maintenant l'étude portait sur la configuration avec le co-processeur ECB-AES. Nous allons dorénavant ajouter la configuration avec le co-processeur XTS-AES intégré de la carte SAMA5D2. Toutes les configurations sont décrites dans la section 4.4. Nous nous appuyons sur

l'outil IOZONE [NC03], un outil aidant à la mesure des débits de chiffrement et déchiffrement pour différentes tailles de fichier ($256MB$ et $512MB$ pour nos tests).

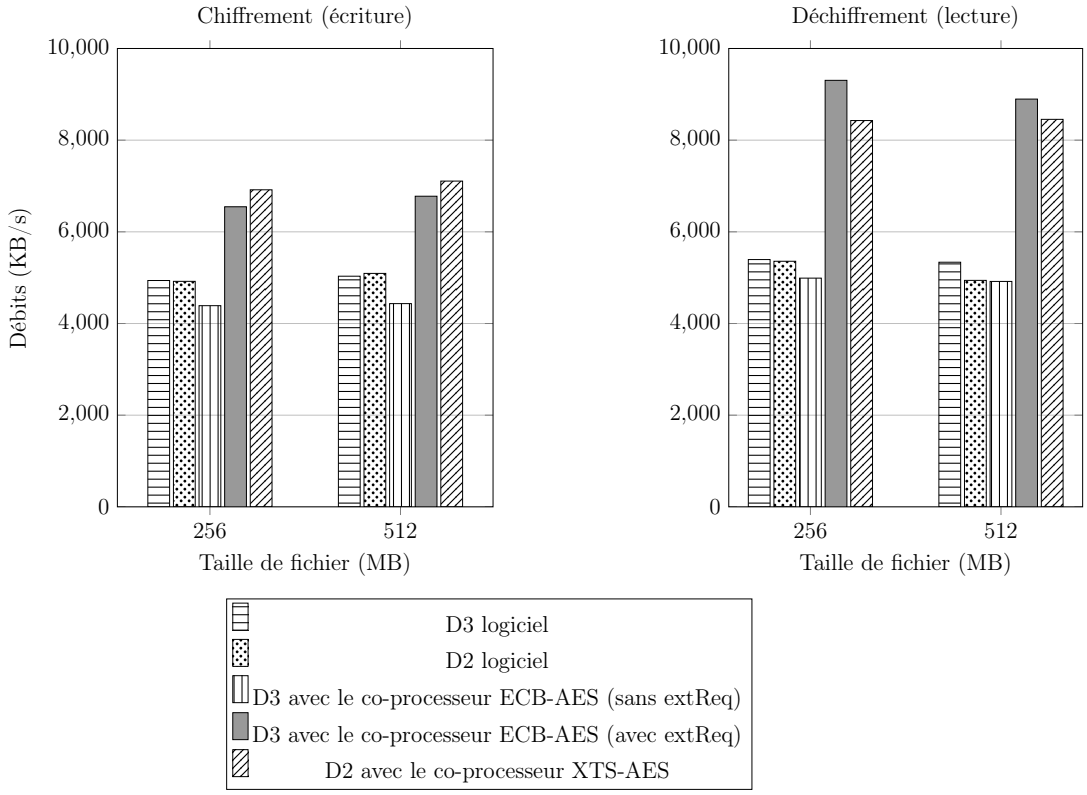


FIGURE 4.6 – IOZONE benchmark, débits de dé/chiffrement pour toutes les configurations.

La Figure 4.6 illustre les résultats obtenus avec l'outil IOZONE. Premièrement et pour rappel, les deux configurations 100 % logicielle affichent des résultats similaires, il est donc cohérent de comparer les deux cartes Atmel. Ces résultats révèlent que l'implémentation, utilisant le co-processeur uniquement pour les opérations de chiffrement / déchiffrement ECB-AES et l'optimisation extReq, permet d'obtenir des performances similaires à la carte Atmel SAMA5D2. Notre solution est légèrement plus lente pour le chiffrement, mais légèrement plus rapide pour le déchiffrement peu importe la taille du fichier. Cette approche combinée des requêtes étendues et d'une implémentation hybride avec le co-processeur ouvre des voies d'amélioration pour d'autres cartes embarquées équipées de co-processeur cryptographique ne supportant pas le mode XTS-AES.
Dès lors, pourquoi ne pas appliquer extReq à cette carte SAMA5D2 dotée d'un accélérateur matérielle supportant le mode XTS-AES ?

4.5 Peut-on appliquer *extReq* au co-processeur du SAMA5D2 pour le mode XTS-AES ?

La question qui se pose naturellement est celle de l'application d'*extReq* pour le driver atmel-aes de la carte SAMA5D2. Cependant, hormis le cas simple du mode ECB-AES, *nous voudrions bien préciser que cette discussion n'est pas supportée par une validation expérimentale en raison du choix de design d'Atmel*, comme expliqué dans les sections suivantes.

4.5.1 Le cas du mode ECB-AES

Nous allons d'abord considérer le mode ECB-AES avec le module SAMA5D2. Ce mode est le plus simple parmi tous les modes de l'algorithme AES. Appliquer notre optimisation *extReq* à ce mode est possible. L'expérience consiste à déchiffrer un fichier conservé dans le conteneur et calculer son condensé MD5. Le temps mesuré, comme vu dans la section 4.4.2, comprend le déchiffrement, les temps d'*E/S + traitements noyaux* et le calcul du condensé MD5.
Quant à la figure 4.7, elle met en évidence les résultats sous forme de débit plutôt que de temps. Ayant remarqué que les valeurs observées ne fluctuaient pas de manière significative avec la taille du fichier (nous avons testé entre 1MB et 500 MB), nous avons retenu uniquement la taille de 256MB. Afin de mettre en valeur l'impact de la taille de la requête, nous avons considéré des tailles de 512 octets jusqu'à 4096 octets. Nous remarquons à cet effet, que plus la taille de la requête augmente, plus le débit est élevé pour attendre un gain de 2.07 dans le cas d'une taille de requête de 4096 octets par rapport à une requête de 512 octets.

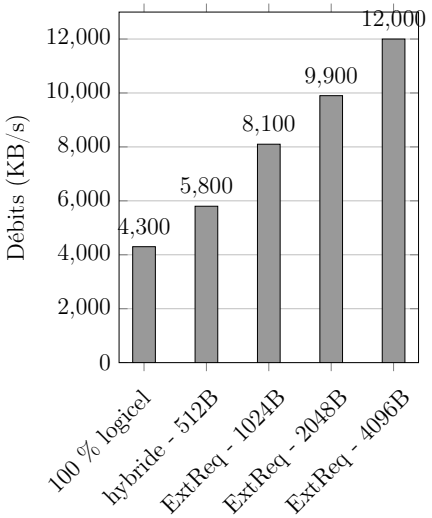


FIGURE 4.7 – Débit de déchiffrement avec le mode ECB-AES pour différentes configurations.

4.5.2 Le cas du mode XTS-AES

Nous considérons maintenant le mode XTS-AES et son implémentation matérielle avec le co-processeur cryptographique de la carte SAMA5D2 qui le supporte. L'algorithme est décrit dans l'alg. 5 pour le chiffrement d'un secteur de 512 octets. Contrairement à notre intuition, toutes les opérations du mode XTS-AES sont décomposées en deux parties :

- **Le chiffrement de l'IV :** est effectué avec le mode ECB-AES du co-processeur cryptographique à l'aide de la clé K_2 pour obtenir eIV. Cette opération doit être implémentée dans le driver.
- **Les autres opérations :** sont gérées par le mode XTS-AES du co-processeur cryptographique (`XTSEngine`) incluant le calcul des tweaks, les deux `XOR` et le second ECB-AES. Les paramètres en entrée sont eIV, P et K_1 et α.

Algorithme 5 : Implémentation du mode XTS-AES dans le driver atmel-aes pour le chiffrement d'un secteur de 512 octets.

input : clé partagée en $(K_1 \parallel K_2)$;
IV (16B) ;
texte à chiffrer P (512B) ;

output : texte chiffréC (512B)

1 eIV ← `AESEncECB` (IV, K_2)
2 C ← `XTSEngine` (eIV, P, K_1)

La question est de savoir si l'on peut mettre à jour le driver pour fonctionner non pas avec un secteur de 512 octets mais une page entière de 4KB en utilisant les fonctions `AESEncECB()` et `XTSEngine()`.

Le co-processeur `AESEncECB()` ne pose aucun soucis car elle peut avoir en entrée n'importe quelle taille de bloc (voir 4.5.1). Mais le co-processeur `XTSEngine()` n'a pas été conçu pour accepter des tailles de texte en entrée de plus de 512 octets, bien qu'il accepte des tailles inférieurs. Les raisons de cette limitation pratique sont triples :

- 512 octets est la taille commune de secteur pour la majorité des appareils ;
- De nombreux vecteurs de test sont fournis par le standard [IEE08] mais sont limités à une valeur maximum de 512 octets ;
- Le paramètre α doit être remis à 0 à la fin du chiffrement ou déchiffrement d'un secteur. Ce paramètre empêche le chiffrement d'une taille supérieur à 512 octets,i.e, sans intervention ou coupure.

Il n'est donc pas possible d'appliquer *extReq* avec le co-processeur de la carte Atmel SAMA5D2. Toutefois, cette limite n'est pas fondamentale, nous l'avons déjà contournée dans l'implémentation hybride avec le co-processeur supportant le mode ECB-AES et *extReq*(voir section 4.2.3). Afin de supporter *extReq*, une meilleure conception de ce co-processeur est possible à l'aide des opérations suivantes :

1. A partir de la taille du texte à chiffrer, **déterminer le nombre de secteur de 512 octets.** Avec une page complète de 4KB, pris comme taille de référence pour la suite, nous avons 8 secteurs de 512 octets ;
2. **Déterminer l'IV de chacun des 8 secteurs** (i.e., leur numéro de secteur). Avec une page complète de 4KB, les 8 secteurs sont nécessairement consécutifs et incrémenter le premier vecteur est suffisant pour déterminer les IVs suivants ;
3. **Initialiser un tampon** de 128 octets avec les 8 IVs consécutifs de 16 octets chacun ;
4. **Chiffrer le tampon** de 128 octets avec une seule opération ECB-AES et la clé K_2 ;
5. **Pré-calculer tous les tweaks** : chaque secteur de 512 octets est composé de 32 tweaks de 16 octets chacun. Dans ce cas, l'exposant α doit être réinitialisé à 0 pour chaque nouveau secteur de 512 octets ;
6. **Initialiser un tampon** de 4096 octets, avec les 256 tweaks de 16 octets ;
7. **Calculer les opérations [XOR]-[ECB-AES]-[XOR]** avec le texte à chiffrer P, le tampon de 4096 octets contenant tous les tweaks et la clé K_1.

En suivant ces lignes directrices, l'implémentation finale respecte le standard et devrait fournir des performances plus élevées. De façon générale, le principe d'*extReq* peut être appliqué à tous les modes de chiffrement dont l'IV est prédictible. Les tests de la section 4.5.1 montrant (sur le mode ECB-AES) les bénéfices de l'optimisation *extReq*, nous sommes en droit d'attendre des gains significatifs également avec le mode XTS-AES. Cette approche doit aussi pouvoir être adaptée dans d'autres drivers et co-processeurs cryptographiques.

4.6 Discussion

Le mode XTS-AES est un mode complexe et peut rapidement devenir un goulot d'étranglement lorsque de grands volumes de données sont traités dans le cadre du FDE. De plus, le mode XTS-AES n'est pas supporté par tous les co-processeurs en raison de son apparition récente : parmi les deux cartes Atmel, SAMA5D2 et SAMA5D3, seule la carte récente SAMA5D2 supporte le mode XTS-AES.
Durant nos travaux, nous avons étudié trois implémentations : du 100 % logicielle (**notre référence**) à celle reposant sur le co-processeur XTS-AES du SAMA5D2 (**meilleur cas**), et entre les deux, l'implémentation hybride reposant sur le co-processeur ECB-AES et le CPU pour les autres opérations du SAMA5D3. Ces tests ont montré d'une part que l'implémentation hybride atteignait des performances proches de l'implémentation logicielle, ce qui était en dessous des attentes. D'autre part, les débits de chiffrement obtenus avec l'implémentation 100 % matérielle correspondaient au double des débits mesurés avec l'implémentation logicielle.
Les performances limitées obtenues (avec l'implémentation hybride du SAMA5D3) nous ont conduit à trouver l'optimisation *extReq* consistant à étendre les requêtes cryptographiques en

créant une requête de la taille d'une page complète (4KB) au lieu de 8 requêtes de 512 octets (taille d'un secteur). Cette optimisation nous a permis de doubler les gains de chiffrement et déchiffrement, bénéficiant principalement à l'atmel-aes driver mais aussi aux E/S et traitements noyaux. Les résultats sont proches de ceux de l'implémentation matérielle du SAMA5D2. Ces études ouvrent, en somme, des perspectives intéressantes pour le FDE dans les systèmes Linux : les machines plus anciennes sans co-processeur XTS-AES pourront grandement accélérer les opérations de chiffrement/déchiffrement par ce biais.

Le dernier point de ce chapitre concerne la faisabilité d'appliquer *extReq* avec le co-processeur XTS-AES de la carte SAMA5D2. Bien que la conception actuelle du co-processeur, limitant les blocs à une taille de 512 octets, empêche cette optimisation (voir section 4.5.2), il n'existe pas de limite fondamentale. Ainsi, nous espérons que les futurs cartes (Atmel et autres constructeurs), intégreront cette optimisation afin d'accélérer les opérations du mode XTS-AES en particulier, et du FDE en général.

Ayant remarqué les limites des cartes Atmel en termes de puissance, nous avons voulu implémenter *extReq* sur une carte plus puissante. Nous avons ainsi testé la carte CL-SOM-AM57x du fabriquant Compulab. Dotée d'un processeur Sitara de Texas Instrument (Cortex A15 à double coeur @1.5 GHz), cette carte plus puissante intègre aussi un co-processeur cryptographique supportant les modes de base et XTS de l'algorithme AES.

Cependant nous n'avons pas pu mettre en oeuvre le chiffrement XTS-AES du fait d'un manque de documentation technique. Nous avons tenté, en vain, d'obtenir la documentation dédiée aux composants de sécurité auprès du fabriquant, du vendeur (intermédiaire) et directement auprès de l'entreprise Texas Instrument. Certains algorithmes de chiffrement ne sont pas implémentés dans les drivers Linux, et pour être en mesure d'effectuer les ajouts (modifications ou optimisations), des informations obligatoires comme les adresses des registres et leurs fonctionnements internes sont indispensables.

Nous voyons que les optimisations présentées dans ce chapitre ont un grand potentiel, que ce soit d'anciens matériels ne supportant pas nativement le mode XTS-AES, ou de nouveaux matériels afin d'aller plus loin en termes de vitesses de chiffrement / déchiffrement. En revanche ceci requiert (1) que les designs des co-processeurs soient suffisamment flexibles pour supporter ExtReq et (2) que les détails techniques soient accessibles afin de pouvoir effectuer les modifications requises dans les drivers Linux.

Chapitre 5

Application de l'architecture TTM à un Cloud d'entreprise

Sommaire

5.1 Introduction

La protection des données est un enjeu majeur pour les entreprises qui utilisent de plus en plus le Cloud comme service d'externalisation des données. La confiance limitée accordée aux services de stockage externe, notamment après les révélations de Snowden, renforce le besoin croissant de chiffrement côté client. A cette fin, nous proposons un Cloud pour entreprise basé sur les principes fondamentaux de notre architecture TTM : *"Trust The Module, Not The Cloud"*. Nous tirons ainsi avantage de la modularité de cette architecture pour offrir de nouvelles fonctionnalités.
Le Cloud pour entreprise pose de nouvelles contraintes qui seront abordées parmi lesquelles la distinction des espaces de stockage (personnel ou partagé), la protection des clés de chiffrement et la nécessité de performances supérieures.

5.2 Modèle d'attaque

Le modèle d'attaque est similaire au Cloud personnel. Nous retrouvons les deux adversaires extérieurs, nous ajoutons cependant un utilisateur malicieux au sein de l'entreprise.

- **un Cloud honnête mais curieux :** l'hébergeur de données est considéré comme honnête. Il exécute les différents services proposés et ne va pas tenter de corrompre les données. Il est cependant considéré comme curieux, susceptible de lire les données stockées à sa propre initiative ou par obligation vis-à-vis d'une autorité (gouvernementale par exemple).
- **un attaquant extérieur :** peut s'infiltrer au milieu de la connexion module-Cloud (en général Internet) et interagir avec le flux (écoute, insertion, supression de paquets). Il peut également s'introduire dans l'infrastructure du fournisseur de services Cloud et accéder à toutes les données chiffrées.
- **Un utilisateur interne :** dans l'entreprise, un utilisateur révoqué d'un groupe pourrait alors tenter d'accéder aux données partagées par la suite.

Dans le schéma 5.1 qui illustre le modèle d'attaque, nous distinguons deux liens principaux :

- **Le lien Client - Module :** ce lien est considéré comme lien de confiance.
- **Le lien Module - Cloud :** ce lien est ouvert à l'extérieur et donc sujet aux attaques.

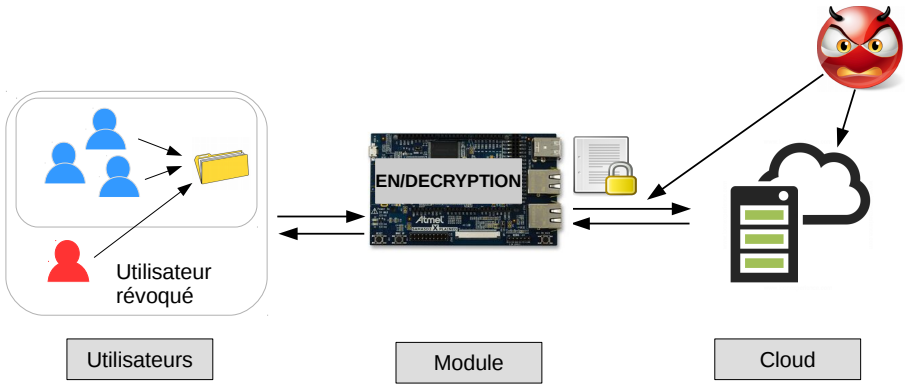


FIGURE 5.1 – Modèle d'attaque de la solution Cloud pour entreprise.

5.3 Énoncé du problème et des besoins

5.3.1 Problématique

Le principe de cette architecture repose sur le concept de TTM en profitant de sa modularité. L'objectif est de développer une solution évolutive répondant aux besoins d'une entreprise. Elle doit être capable de s'adapter aux changements,e.g., augmentation du nombre d'utilisateurs. A la différence du Cloud personnel, nous devons cette fois intégrer un système existant, celui de l'entreprise (par ex. pour l'authentification).

5.3.2 Les besoins

Notre solution TTM pour entreprise doit respecter certains critères de fonctionnalité et de sécurité. A ceux déjà identifiées pour le Cloud personnel (que nous ne rappelons pas, cf. section 3.3.2) s'ajoutent :

- **L'isolation :** la gestion des permissions et des droits d'accès (authentifications) doit être indépendante du chiffrement des fichiers pour le respect du critère de modularité.
- **La protection des clés de chiffrement par PKI :** les clés de chiffrement ne doivent pas dépendre de la robustesse d'un mot de passe à l'instar de l'algorithme utilisé par défaut dans le FDE sous Linux (i.e. PBKDF2). Elles doivent donc être protégées par chiffrement RSA (une PKI).
- **Les mécanismes d'urgence liés aux clés de chiffrement :** la gestion et le stockage des clés de chiffrement doivent d'intégrer des mécanismes d'urgence :
  - *L'arrêt d'urgence* permettant de supprimer les clés du module de sécurité, afin de bloquer le déchiffrement d'un conteneur.
  - *La destruction de conteneur* supprime l'accès aux données de manière définitive.

5.4 Notre architecture TTM pour entreprise : "Trust The Module, Not The Cloud"

5.4.1 Introduction à l'architecture TTM pour entreprise

Deux différences fondamentales existent par rapport au Cloud personnel. La première concerne la gestion de l'authentification et des permissions. En effet, au sein d'une entreprise les conteneurs peuvent être accédés soit sous forme d'espace de partage pour un groupe, soit sous forme d'espace de stockage individuel. Pour ces raisons, nous introduisons une nouvelle entité dans notre architecture qui est le *serveur d'authentification*. Il gère à la fois l'authentification et les droits d'accès. Cette entité peut déjà exister au sein de l'entreprise.
La deuxième différence concerne le choix du module de sécurité matériel. Il est déterminé par deux critères :

— **Les performances :** le nombre de conteneurs actifs en parallèle détermine le dimensionnement du module de sécurité en terme de puissance de calcul. Le module doit ainsi être capable de gérer en parallèle des dizaines de conteneurs. Si nécessaire plusieurs modules peuvent co-habiter afin de répondre à des besoins plus grand.

— **La gestion des clés de chiffrement** détermine le type de module choisi. Dans notre cas, nous souhaitons protéger les clés de chiffrement non pas à l'aide d'une dérivation de clés comme expliqué dans l'annexe A mais plutôt via un chiffrement RSA. Pour rappel, jusqu'à maintenant la clé de chiffrement des données d'un conteneur, appelée Master Key (MK), était chiffrée à l'aide de la clé dérivée puis conservée dans l'entête du conteneur. Nous ajoutons une nouvelle fonctionnalité qui est la protection de la MK à l'aide d'une paire de clés RSA. La clé est *wrappée* (emballée, chiffrée) avec la clé publique et *dewrappée* avec la clé privée.

Zoom sur la protection de la Master Key par PKI sous Linux

Pour rappel sous Linux, le FDE s'appuie sur le logiciel *cryptsetup* pour la configuration des conteneurs et la surcouche LUKS pour le format. Nous allons expliquer comment protéger la Master Key à l'aide d'une PKI à la place d'un chiffrement via une clé dérivée. Nous distinguons les deux étapes de la gestion des conteneurs :

— **L'initialisation :** durant cette étape, nous allons récupérer la clé publique de l'utilisateur/du groupe appelée $Pub_{utilisateur}$. Nous nous focalisons sur l'utilisateur mais ceci est pareil pour un groupe car dans tous les cas une seule paire de clés est associée à un conteneur. Puis, une fois le certificat vérifié à l'aide d'un Centre d'Autorité, une clé symétrique XTS-AES (MK) de 256 ou 512 octets est générée à l'aide du PRNG (sans suffixe la MK est considérée en clair). Le conteneur est alors initialisé via la commande *luksFormat*. Au final, la $MK_{protegee}$, wrappée à l'aide de la clé publique, est conservée dans l'en-tête du conteneur selon la formule

suivante :

$$MK_{protegee} = E(MK, Pub_{utilisateur}). \tag{5.1}$$

La notion de wrapping/unwrapping intervient dans le cadre de la gestion des clés dans un HSM. Le principe fondamentale est de ne pas extraire les clés privées à l'extérieur du HSM. Cependant, dans le cas où la clé doit être transmise ou conservée dans un autre périphérique, un mécanisme appelé wrapping de clé existe. Ce mécanisme est similaire au chiffrement et déchiffrement. La différence est que le texte à chiffrer / déchiffrer n'est pas une donnée quelconque mais une clé.

— **L'ouverture :** afin d'ouvrir le conteneur, la $MK_{protegee}$ est récupérée puis déchiffrée à l'aide de la clé privée de l'utilisateur/du groupe appelée $Priv_{utilisateur}$:

$$MK = D(MK_{protegee}, Priv_{utilisateur}). \tag{5.2}$$

Enfin, une preuve de concept de ce principe sur la carte Atmel SAMA5D3 nous a permis de valider l'utilisation d'une paire de clé RSA dans la gestion des conteneurs via LUKS/cryptsetup.

Pour répondre à ces besoins nous proposons de concevoir une solution basée sur un module de sécurité avec **une forte puissance de calcul** et **certifié** appelé Module de Sécurité Matériel (Hardware Security Module -HSM-). Le choix du HSM sera déterminé dans la partie preuve de concept (section 5.5).

5.4.2 Troisième principe architectural : isolation utilisateurs - infrastructure de stockage pour une flexibilité maximum

Nous introduisons, en dernier lieu, un nouveau principe architectural dans la section suivante consistant à isoler la partie authentification et permissions de la partie chiffrement des données. Une vue haut niveau de l'architecture est décrite dans la Figure 5.2. On remarque la présence de nouvelles entités par rapport au Cloud personnel :

— **Les utilisateurs :** sont plus nombreux et peuvent être regroupés dans des groupes.
— **Le serveur d'authentification :** gère l'authentification des utilisateurs et les droits d'accès.
— **Le serveur de chiffrement et déchiffrement (SCD) :** concentre les opérations cryptographiques liées au chiffrement et déchiffrement des fichiers.
— **Le HSM :** concentre les opérations cryptographiques liées à la gestion des clés de chiffrement. Il est utilisé pour protéger les Master Key.

Le principe d'isolation utilisateurs-infrastructure de stockage permet une modularité forte et repose sur des raisons pratiques. La solution que nous proposons doit s'intégrer dans des sys-

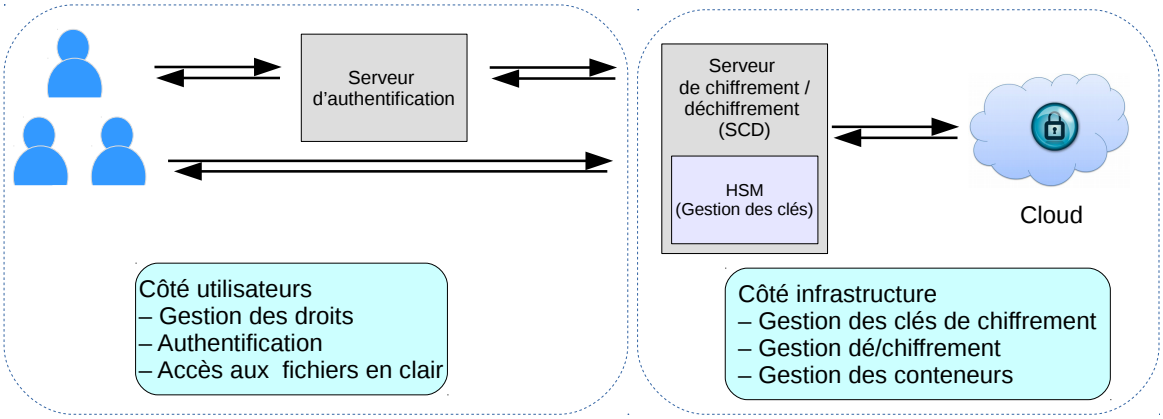


FIGURE 5.2 – Vue de haut niveau de l'architecture TTM pour un Cloud entreprise.

tèmes existants. Ainsi, quels que soient les outils utilisés dans le côté utilisateur (en particulier pour l'authentification), l'architecture TTM peut s'adapter soit en intégrant les outils existants soit en les modifiant afin de tirer profit du HSM.

5.5 Preuve de concept

5.5.1 Vue d'ensemble

Le cas d'usage dans le cadre de cette preuve de concept consistait à protéger des flux vidéos d'une entreprise. Aussi nous étaient imposés les contraintes sur le réseau et le serveur d'authentification. La flexibilité de notre solution a permis de s'adapter à ces différents besoins.
Nous créons un réseau avec des postes Clients Windows et un serveur d'authentification Windows intégrant une Active Directory (AD). Les autres serveurs de sécurité et de stockage des données contiennent une distribution Linux (Ubuntu 14.04.1 LTS, kernel 3.3). Certaines entités de l'architecture se regroupent dans un réseau privée (LAN) : les postes Clients, le serveur de chiffrement/déchiffrement (appelés SCD) et le serveur d'authentification. De surcroît, le serveur de stockage, Cloud, peut être en interne (privée) ou chez un hébergeur de données (publique). Dans notre plateforme de test, le Cloud et le HSM sont distants. Concernant le HSM, hébergé chez Bull, les requêtes sont envoyées via internet à travers un canal sécurisé. Ceci pour des raisons pratiques, car la possession d'un HSM physique réel engendrerait des coûts élevés dans le cadre d'un démonstrateur (cf . Zoom ci-dessous).
Chaque entité remplit un rôle défini (cf . Figure 5.3) :

— **Les postes Clients :** se connectent à leur session. Ils ont à disposition un raccourci sur l'écran d'accueil (bureau) permettant de connecter un lecteur réseau donnant accès à leur conteneur (espace de stockage) via SAMBA.

— **Le serveur Windows avec l'Active Directory (AD) :** toute l'architecture réseau est configurée via le serveur Windows. Les permissions et droits d'accès sont configurés

dans l'AD. Le serveur SAMBA discute avec l'AD afin d'authentifier les utilisateurs sans leur demander à nouveau leur mot de passe.

— **Le serveur de chiffrement/déchiffrement (SCD) :** ce serveur, doté d'un processeur puissant et intégrant les instructions Intel AES-NI, s'occupe de chiffrer et déchiffrer les données. Il est relié au Cloud, au HSM et aux postes clients.

— **Le HSM :** est certifié par l'ANSSI. Les requêtes cryptographiques de gestion de clés de chiffrement sont envoyées via le réseau à travers un canal sécurisé.

— **Le Cloud :** conserve les conteneurs chiffrés. Il est relié au SCD.
Remarque : dans le cadre d'une configuration réelle, le SCD et le HSM intégreront un même appareil (vendu ainsi par Bull). Le SCD sera le CPU principal (potentiellement avec plusieurs coeurs) alors que le HSM correspondra aux co-processeurs matériels. Dans la suite du document, la référence à HSM correspond aux co-processeurs cryptographiques alors que le SCD est une machine différente mais qui intègre le même CPU que sur l'appareil réel.

Zoom sur le module de sécurité matériel : HSM Proteccio

Le HSM Proteccio est adapté à notre architecture TTM pour entreprise. Sa puissance de calcul, sa certification et ses fonctionnalités cryptographiques répondent aux besoins de notre solution. Nous n'avons cependant pas eu accès à un HSM Proteccio physique mais distant. Ce dernier était hébergé chez Bull. Les commandes étaient alors envoyés via le réseau à travers un canal sécurisé. Cette contrainte n'était pas bloquante pour notre preuve de concept.
Les caractéristiques sont les suivantes :

— Un processeur Intel Core i7 avec la technologie AESNI (i.e., permettant d'accélerer le dé/chiffrement des modes de l'AES). *Dans le cadre de notre démonstrateur, ce CPU est sur une machine différente (le SCD).*
— De 2 à 16 Go de RAM.
— Les certifications : critère commun EAL4+c et ANSSI.
— Une interface graphique de gestion des clés stockées sur le HSM.
— 8 modules intégrés dédiés à des opérations cryptographiques pouvant être indépendant.
— Trois rôles prédéfinis : utilisateurs, administrateurs, responsables d'audit.

5.5.2 Vue détaillée

Nous allons nous attarder sur les différentes entités et détailler leurs rôles et interactions. Nous reprenons la Figure 5.3 afin de comprendre les différents services.
Un premier lien réside entre le poste Client et le serveur Windows. La connexion à la session Windows à partir du poste Client s'effectue via le couple identifiant/mot de passe. Cette au-

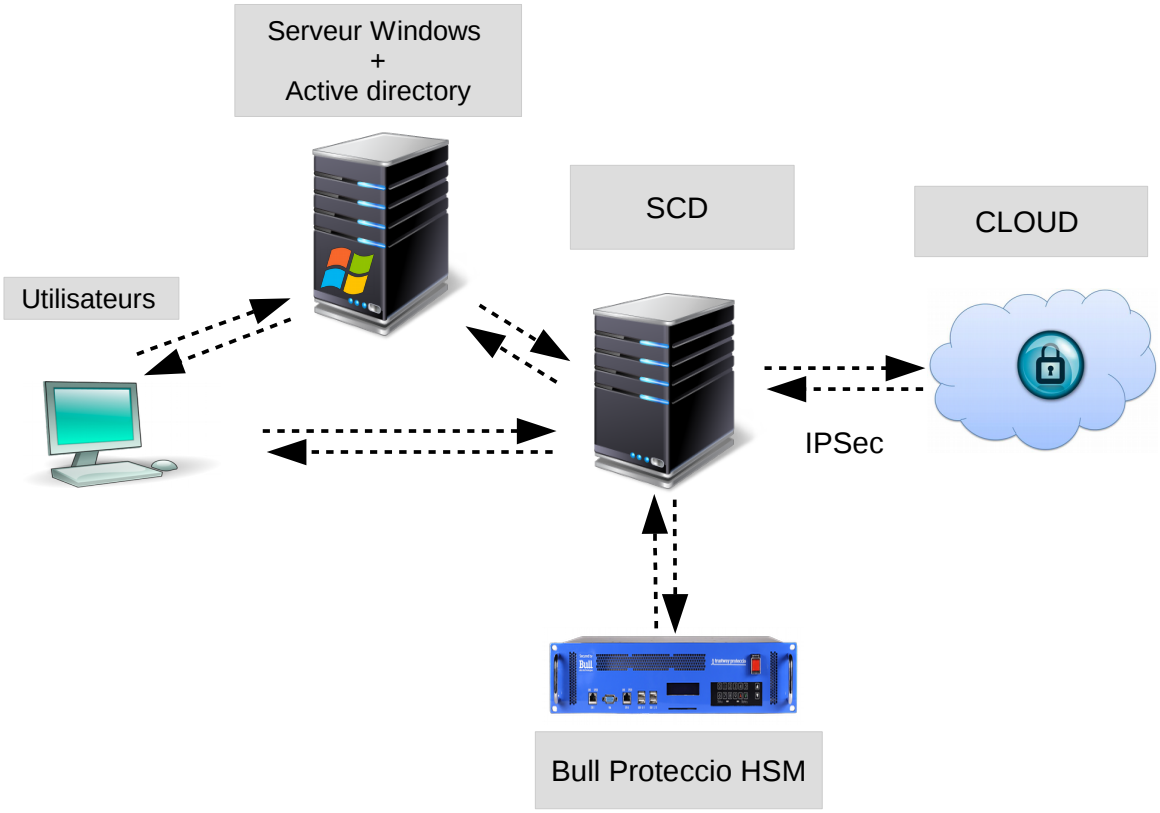


FIGURE 5.3 – Vue d'ensemble du démonstrateur pour un Cloud entreprise.

thentification et les droits d'accès sont gérés par l'AD. De plus, le Client se connecte à son espace de stockage via le protocole SMB (Server Message Block) sur le SCD. Cette connexion au serveur SAMBA s'effectue via l'adresse IP et le nom du partage. En outre, le poste client et le SCD étant sur le même réseau, les données entre ces deux entités sont envoyées en clair.
Un deuxième lien existe entre le SCD et le Cloud via le protocole iSCSI. Ce procédé est identique au Cloud personnel. Pareillement, l'accès au conteneur requiert des commandes SCSI via le réseau TCP/IP. Ces commandes permettent, entre autres, de lire/écrire à un certain décalage (offset) dans le conteneur.
Un troisième lien relie le SCD avec le poste Client. Au sein du SCD, un service SAMBA permet de partager des espaces de stockage à des groupes et utilisateurs. La particularité de SAMBA est de pouvoir partager des fichiers conservés sur un serveur Linux avec des postes Windows. Il permet de lancer des scripts à la connexion et déconnexion d'un utilisateur. Ainsi, une fois connecté, un script de connexion au conteneur se lance. Les détails sont disponibles dans la section 5.5.2.
Par ailleurs, les interactions sécurisées permettant de gérer et protéger les clés s'effectuent entre le SCD et le HSM. Les requêtes cryptographiques entre ces deux entités respectent le standard PKCS #11. Différentes opérations sont supportées telles que : C_GenerateKey pour la génération d'une clé symétrique, C_Encrypt pour chiffrer une donnée, C_Wrap pour protéger une clé et l'extraire de manière sûre. Les détails du standard PKCS #11 sont données dans la

partie *zoom* ci-après.

Zoom sur le standard de la cryptographie à clé publique

Le standard PKCS #11 (Public-Key Cryptography Standards) a été développé par RSA Security Inc. "*en partenariat avec les représentants du monde industriel, académique et des gouvernements*"[RSA01]. Le but est de fournir des standards pour les développeurs afin de permettre une inter-opérabilité entre les différents produits des fabriquants. Le standard couvre en réalité de nombreux aspects de la cryptographie à clé publique :

- **PKCS #1 :** Standard pour le chiffrement RSA.
- **PKCS #11 :** Interface de jeton (token) cryptographique standard.
- **PKCS #8 :** Standard de la syntaxe permettant de stocker les informations de la clé privée.

De nombreux produits supportent et utilisent le standard PKCS #11 comme le navigateur Mozilla Firefox, les modules matériels de Thalès ou Bull. Le but du standard est de fournir une interface entre les applications et les modules cryptographiques.

La notion de jeton (token) dans le standard PKCS #11 représente un appareil stockant des objets (e.g. clés, données ou certificats) et permettant d'effectuer des opérations cryptographiques. Lorsque l'on souhaite utiliser ou communiquer avec un jeton, il est nécessaire de se loguer afin de s'identifier à l'appareil. Par la suite, l'utilisateur est en mesure d'effectuer des requêtes cryptographiques à travers l'interface ou l'API.

Chaque objet possède des attributs, le premier attribut est le type de clé (publique, privée ou secrète). Différents attributs peuvent alors être paramétrés (liste non exhaustive) :

- **CKA_SENSITIVE :** si VRAI la clé ne peut pas être révélée en clair au sein du jeton.
- **CKA_EXTRACTABLE :** si VRAI la clé peut être extraite de l'appareil.
- **CKA_ENCRYPT :** si VRAI la clé peut être utilisée pour chiffrer une donnée.
- **CKA_PRIVATE :** si VRAI la clé ne sera visible qu'à l'utilisateur authentifié.
- **CKA_PUBLIC :** si VRAI la clé sera visible à tout le monde (sans authentification au préalable).
- **CKA_WRAP :** si VRAI la clé peut être utilisée pour "wrapper" (chiffrer) une autre clé.

Une attention particulière doit être accordée aux attributs. Le paramétrage de l'attribut CKA_EXTRACTABLE à VRAI d'une clé par exemple doit être étudié de manière approfondie, si ce dernier doit être envisagé. Généralement, le partage de la clé entre les différents jetons justifie un tel choix. D'autres risques liés à l'API existent. Citons l'attaque de séparation des clés : la clé secrète K_s est créée avec les attributs CKA_SENSITIVE et CKA_EXTRACTABLE à VRAI dans le token. Son handle (h_K_s) est récupérable

avec C_FindObjects. Les différentes étapes de l'attaque sont les suivantes :

1. **Générer** la clé de wrapping K_{wrap} :

   $h\_K_{wrap} = C\_GenerateKey(CKA\_DECRYPT = TRUE, CKA\_WRAP = TRUE)$;

2. **Wrapper** la clé secrète avec K_{wrap} :

   $wrapped = C\_WrapKey(h\_K_{wrap}, h\_K_s)$;

3. **Déchiffrer** la clé wrappée en utilisant la clé de wrapping K_wrap :

   $h\_K_s = C\_Decrypt(h\_K_{wrap}, wrapped)$;.

Inévitablement, l'attaquant récupère alors la valeur de K_s en clair. Cette attaque profite du rôle conflictuel de la clé de wrapping de sorte que cette dernière possède les attributs lui permettant à la fois de wrapper et de déchiffrer une clé. Toutefois, plusieurs solutions existent. D'une part, la création de clé peut être restreinte à un utilisateur spécifique. D'autre part,une autre solution serait d'ajouter des mécanismes de détection des rôles conflictuels lors de la création de clés.

Initialisation d'un conteneur

Les différentes étapes d'initialisation d'un conteneur sont (cf. Figure 5.4) :

— **L'étape 1 :** permet la création du conteneur au sein du Cloud. Il est nécessaire d'installer sur le Cloud un outil de gestion du protocole iSCSI coté serveur (appelé target) tel que iscsi-target. A titre de rappel, le conteneur peut prendre plusieurs formes : un conteneur fichier ou une partition disque. Nous avons choisi les conteneurs fichiers pour leur flexibilité. En revanche, un inconvénient majeur du FDE est la taille du disque qui n'est pas modifiable. En conséquence, le choix de la taille des conteneurs doit s'effectuer en amont avec une analyse précise des besoins. Une fois la taille du conteneur choisie, il faut le créer et le remplir de données aléatoires afin de ne pas distinguer le taux de remplissage du conteneur.

— **L'étape 2 :** permet de se connecter au conteneur via le protocole iSCSI. Une fois connecté, un identifiant unique généré par le client iSCSI pour ce conteneur est stocké dans $/dev/disk/by - id$ sous la forme $scsi - xxxxxx$. Nous l'appelons ID_c.

— **Les étapes 3-4-5 :** permettent d'initialiser le conteneur avec le format LUKS. Pour cela, une paire de clés est générée dans un premier temps avec ID_c. Puis, une clé de chiffrement XTS-AES est générée avec l'identifiant ID_k (nommé suivant une convention afin de repérer facilement le conteneur auquel il est associé). L'étape 5 chiffre la MK avec la clé publique. Au final, la MK chiffrée récupérée est conservée dans l'entête du conteneur.

— **Les étapes 6-7 :** permettent de créer un système de fichier. Il faut au préalable ouvrir le conteneur. La procédure complète d'ouverture est détaillée dans le paragraphe suivant.

Une fois ouvert, la commande $mkfs.ext4$ a pour charge de créer le système de fichier (*ext4fs ou autre*).

— **L'étape 8 :** permet de fermer le conteneur et de détruire la MK stockée dans le HSM. Pour y parvenir, la fonction DestroyKey est appelée avec comme paramètre l'identifiant ID_k.

Accès à un conteneur

Après avoir vu l'initialisation, nous allons maintenant détailler la phase d'accès au conteneur pour un utilisateur. Nous nous limiterons au cas d'un utilisateur unique souhaitant accéder à son conteneur personnel pour plus de simplicité. Les étapes présentées dans la Figure 5.5 sont les suivantes :

— **L'étape 1 :** permet à l'utilisateur de se connecter à sa session Windows via le Windows serveur. Les droits d'accès et l'authentification sont gérés par l'Active Directory (AD).

— **L'étape 2 :** permet à l'utilisateur d'accéder à un partage configuré dans le serveur SAMBA. L'authentification via kerberos s'effectue directement via l'AD. En ce sens, l'envoi de l'identifiant/mot de passe de l'utilisateur n'est pas nécessaire. Après vérification du ticket Kerberos, l'accès au partage est autorisé. Notons que par défaut le partage est vide, dans la mesure où pour l'instant le partage est un dossier local au SCD. Par la suite, nous allons effectuer un montage du conteneur vers ce dossier de partage.

— **L'étape 3 :** permet de se connecter au conteneur stocké sur le Cloud. Un script de pré-exécution intégré au serveur SAMBA permet d'automatiser la connexion et l'ouverture du conteneur. La connexion via le protocole iSCSI est réalisée soit à l'aide d'une authentification simple (client-serveur) soit mutuelle (client-serveur et serveur-client). Le tout est paramétrable dans les fichiers de configuration de l'initiateur et de la target[Dwi05]. Une fois la connexion établie, le SDC reçoit l'ID_c ainsi que la MK chiffrée.

— **L'étape 4 :** permet la récupération de la MK déchiffrée. La requête Unwrap du standard PKCS#11 permet de déchiffrer la Master Key. Cette opération nécessite trois paramètres. Le premier est l'identifiant de la paire de clés RSA (ID_c), le second est l'identifiant de la future clé AES qui va être déchiffrée (ID_k de MK). La Master Key chiffrée en est le troisième. L'identifiant de la MK ID_k,qui plus est, est utilisé dans l'interface d'administration dans le but de rendre possible à un officier de sécurité (ayant les droits) de visualiser les conteneurs en temps réel. D'ailleurs, cet aspect est aussi essentiel dans le cas où ce dernier souhaiterait fermer brutalement un conteneur ouvert.

— **L'étape 5 :** permet d'ouvrir le conteneur à l'aide de la commande *luksOpen*. Dans ce cas, la possession de la MK est nécessaire.

— **L'étape 6 :** permet de monter le conteneur au répertoire de partage SAMBA. L'utilisateur sera en mesure d'accéder à ses fichiers. Le déchiffrement est alors effectué à la demande, seuls les fichiers consultés sont déchiffrés.

— **L'étape 7 :** permet de fermer le conteneur à l'aide de la commande *luksClose* qui prend

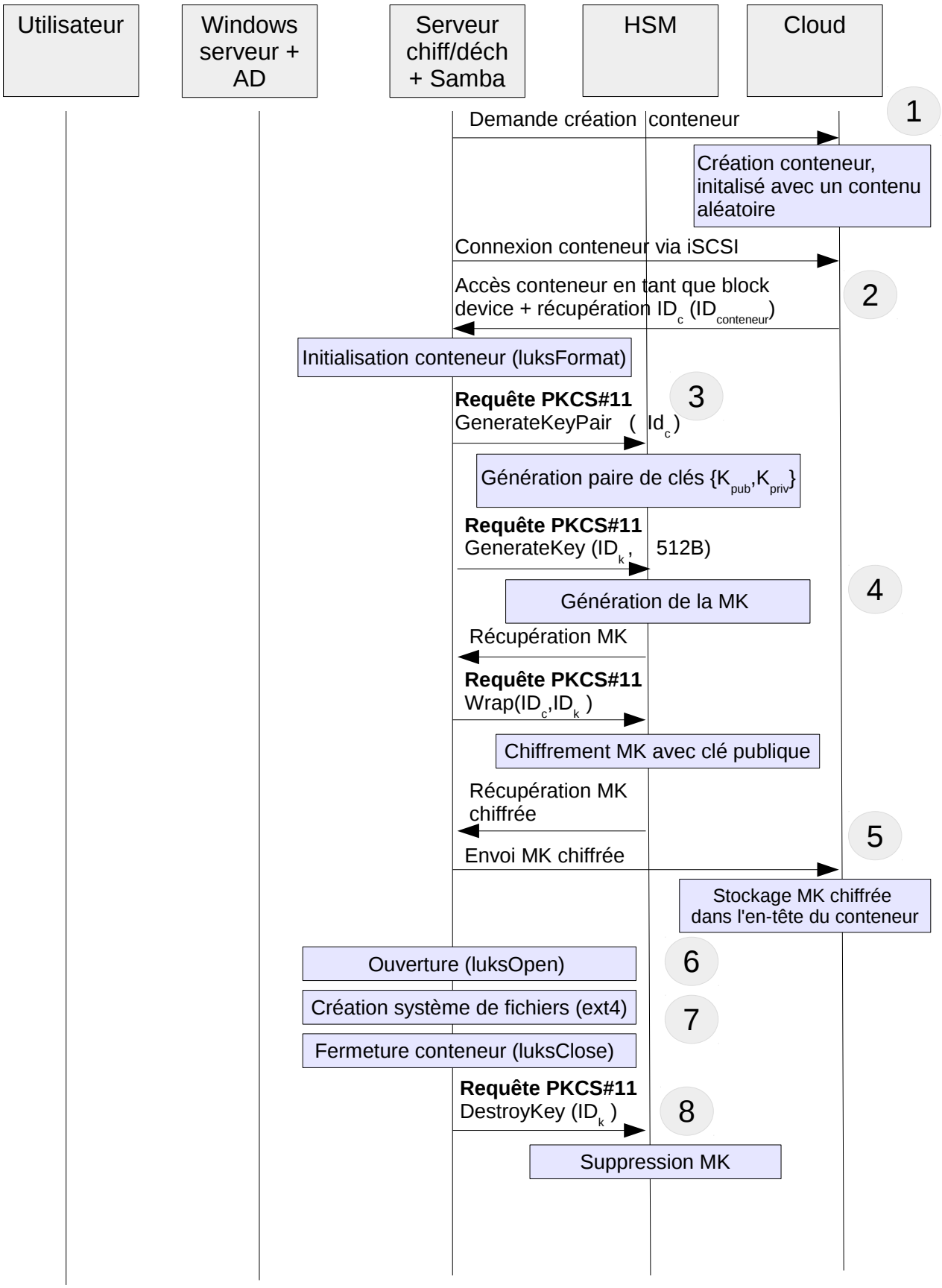


FIGURE 5.4 – Initialisation d'un conteneur.

comme paramètre ID_k. La MK est alors détruite de la mémoire du HSM.

Nous avons vu la procédure complète de l'initialisation jusqu'à l'ouverture et l'accès aux fichiers d'un utilisateur unique. La gestion d'un conteneur pour un groupe suit la même procédure, puisque la paire de clé RSA générée est liée au conteneur et non aux utilisateurs. La différence concerne l'authentification et les droits accordées au partage SAMBA. Il est important de comprendre dans cette architecture l'isolation entre :

- **Le côté client et l'accès aux fichiers :** cette partie repose sur des gestions de droits et de l'authentification. Nous avons utilisé le protocole Kerberos pour authentifier les utilisateurs au service SAMBA du SDC. Les droits d'accès aux conteneurs sont repris directement de l'AD. C'est pourquoi une nouvelle politique de droit d'accès dans le SDC et le service SAMBA ne sont pas nécessaires. Toutefois, les enjeux de ce côté sont des problématiques d'authentification. Dans le cas de l'utilisation d'une PKI à la place du protocole Kerberos, des requêtes supplémentaires pourraient être effectuées vers le HSM afin de vérifier les identités via des certificats.
- **Le côté SDC, HSM et Cloud :** cette partie concerne la gestion des clés et du chiffrement/déchiffrement liés aux conteneurs. Une fois le conteneur ouvert, ce dernier est monté dans le dossier de partage SAMBA.

En résumé, cette séparation permet d'isoler ces deux parties. En cas de problème de sécurité, les officiers peuvent intervenir sur un des deux côtés, soit en fermant le partage SAMBA, soit en fermant le conteneur.

5.5.3 Résultats et performances

L'architecture proposée ci-dessus a été implémentée avec des machines virtuelles chez un hébergeur. Le réseau était limité à une connexion 100 Mbit/s. Le SCD, le Cloud et le serveur Windows étaient chacun intégrés à une VM. De plus, le HSM était aussi virtuel. Ces requêtes passent nécessairement par le réseau. En ce sens les performances ne peuvent égaler un HSM physiquement présent sur le même sous-réseau. Cette limite n'a cependant que très peu d'incidence en raison du faible nombre de requête PKCS #11. Effectivement, après initialisation du conteneur, seulement quelques requêtes sont nécessaires pour l'ouverture de ce dernier. Les nouvelles requêtes par contre seront envoyées uniquement lors de la fermeture.

Dans le cadre des tests de conteneur la clé XTS-AES a une taille de 512 octets (2 sous clés de 256 octets), et la taille de fichier varie de 1MB à 1GB. Pour mesurer les performances, nous avons défini trois scénarios :

- **Scénario 1 :** mesure des performances réseau (MB/s) **sans chiffrement** afin d'identifier les limites du réseau. Les outils utilisés sont *iperf* et *netcat*.
- **Scénario 2 :** mesure des débits moyens de déchiffrement (MD5) du serveur de chiffrement /déchiffrement dans des conteneurs locaux.
- **Scénario 3 :** mesure des débits moyens de déchiffrement (MD5) sur le poste client dans

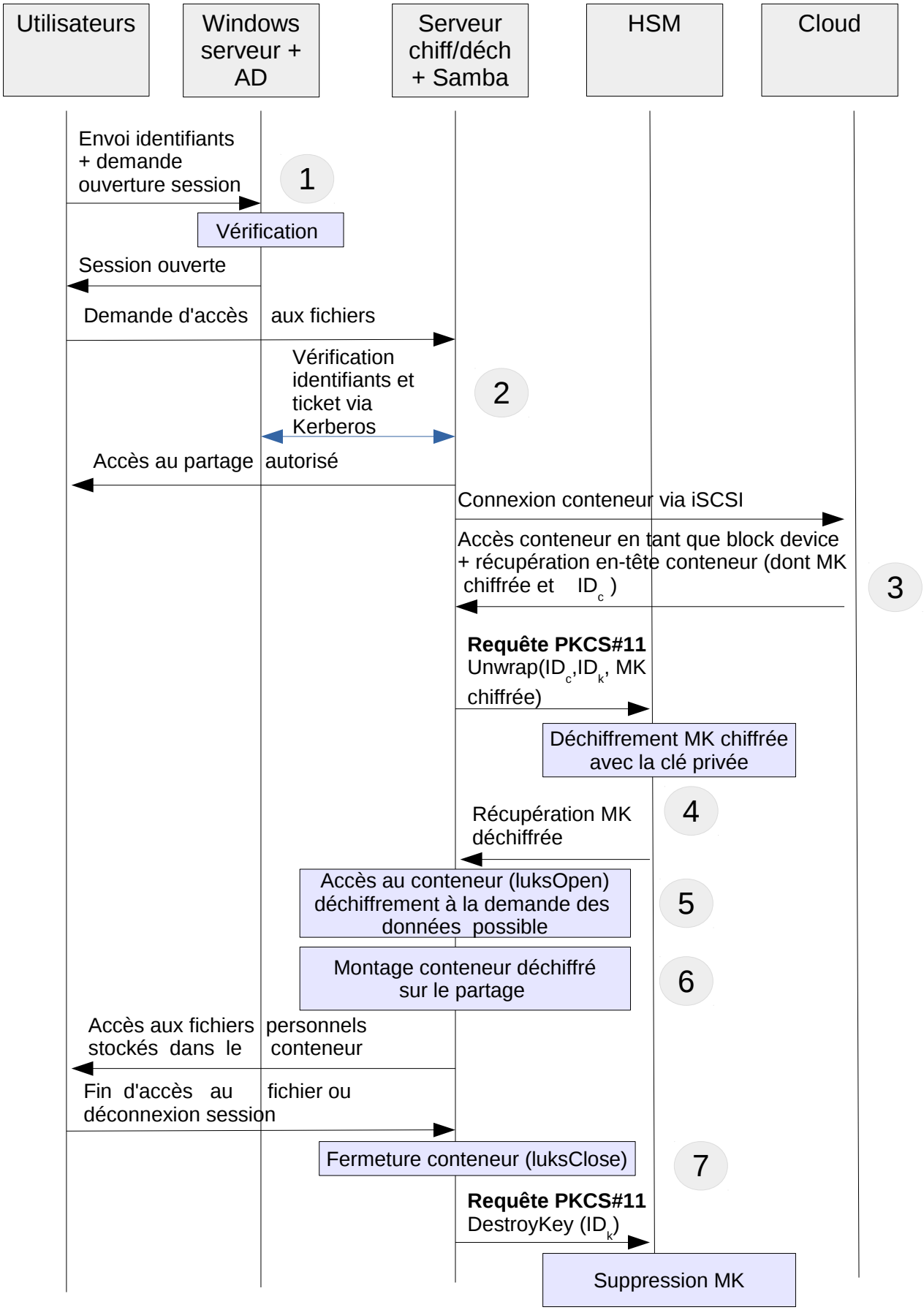


FIGURE 5.5 – Accès à un conteneur.

le cadre de l'architecture complète.

TABLE 5.1 – Mesures de différents débits pour les 3 scénarios.

Scénario 1	**Scénario 2**	**Scénario 3**
6.1 MB/s	501.5 MB/s	6.05 MB/s

Les mesures du tableau 5.1 montrent que les limites des débits proviennent du réseau (limité à 100 Mbit/s) et non pas des capacités de chiffrement et déchiffrement du SDC. Les débits de chiffrement et déchiffrement en local pour le scénario 2 révèlent une puissance de calcul importante. Ceci s'explique par l'optimisation pour les processeur Intel Core i7 avec les instructions CPU AES-NI. Dans le cadre d'une entreprise, le critère principale n'est pas forcément le nombre d'utilisateur mais le nombre de conteneurs ouverts en parallèle, et en cas de besoins, plusieurs SCD pourraient être utilisés conjointement pour faire face à la charge.

5.6 Analyse de sécurité

Nous allons analyser différents points de sécurité en se basant principalement sur les besoins que l'on avait listés dans la section 5.3.

5.6.1 Confidentialité des données

Avec l'utilisation du FDE, le Cloud n'est pas en mesure de déchiffrer les données stockées dans le conteneur. Par ailleurs, les clés de chiffrement sont présentes dans l'entête de chaque conteneur mais protégées par un chiffrement RSA à l'aide de la clé publique associée à l'identifiant du conteneur. De même, si un attaquant malicieux surveille le réseau en se plaçant entre l'entreprise et le Cloud, il ne verra qu'un trafic chiffré (les commandes iSCSI sont ainsi protégées).
Le fournisseur possède cependant un contrôle total sur les entrées sorties vers le serveur. Il est capable de connaître les zones du disque écrites ou lues. Ainsi, des informations statistiques sur l'accès à une certaine zone de la mémoire disque peuvent être inférées.

5.6.2 Contrôle d'accès

Le contrôle d'accès dans l'architecture proposée repose sur l'AD et le serveur SAMBA. L'accès aux données et le déchiffrement sont deux opérations distinctes comme expliqué dans la section 5.5.2. L'utilisateur final accède à ses données après l'authentification via l'AD. Les groupes existants au sein de l'AD sont récupérés par le serveur SAMBA. Si un utilisateur appartient à un groupe dans l'AD alors il sera répertorié de la même manière par le serveur SAMBA. En d'autres termes, les droits d'accès de l'AD sont directement utilisés de manière transparente. En revanche, au lieu d'utiliser le protocole réseau Kerberos, toute la partie authentification pour l'accès au conteneur pourrait être effectuée par une PKI gérée par le HSM.

5.6.3 Sauvegarde et restauration des clés

Le chiffrement des données permet une sécurité importante en terme de confidentialité. Cependant les risques associés demeurent aussi dangereux. Si les clés de chiffrement venaient à être perdues ou altérées, les données deviendraient inaccessibles. Il est donc nécessaire, au sein de toute infrastructure, d'avoir un système de sauvegarde et de récupération des clés. Ces deux problématiques peuvent être gérées grâce à LUKS et sa gestion multi-utilisateurs.
En effet, pour chaque conteneur associé à un groupe ou un utilisateur, une paire de clés supplémentaire d'un administrateur de sécurité pourrait être incluse dans l'entête du conteneur. De cette façon, la Master Key serait protégée avec cette paire de clés. En cas de perte de clés ou une erreur quelconque, les données seraient toujours récupérables avec la paire de clés de l'administrateur de sécurité. Le défi serait par contre de bien gérer cette paire de clés et faire une copie physique en lieu sûre par exemple. Il est certain qu'un accès à cette paire de clés permettrait d'accéder à **tous les conteneurs**. Une deuxième solution intermédiaire est de créer une paire de clés de secours par entité ou projet afin de limiter les risques.
Un autre aspect concerne l'intégrité de l'entête. En effet, la MK wrappée est conservée uniquement dans l'entête du conteneur. C'est pourquoi, en cas d'altération de l'entête et donc de la MK wrappée, la clé de chiffrement serait irrécupérable.
En résumé deux points sont à prendre en compte pour protéger les clés de chiffrement :

- Faire une sauvegarde des conteneurs, ou à minima, de l'entête des conteneurs ;
- Wrapper la MK d'un conteneur avec un autre utilisateur/administrateur pour chaque conteneur.

5.6.4 La question de l'intégrité

L'intégrité n'a pas été traitée en détail jusqu'ici. Il est essentiel de comprendre en détail le fonctionnement du FDE pour se rendre compte des difficultés de l'ajout d'un service de vérification de l'intégrité des données. Un des critère du FDE est que la taille des données à chiffrer correspond **exactement** à la taille du chiffré. Il n'existe pas d'espace de stockage au sein du conteneur pour des informations additionnelles. Or, tous les algorithmes de chiffrement authentifiés comme GCM ou CBC-MAC prennent en entrée un secteur et génèrent en sortie le texte chiffré et son tag associé, ce dernier étant utilisé comme preuve d'intégrité. Dans les solutions au niveau du système de fichiers, cette information est stockée, par exemple, dans l'entête du fichier. En revanche dans le cas du FDE, il est nécessaire de stocker le tag ailleurs. Ceci peut être dans un fichier annexe ou un autre conteneur.
Bien que la conception paraisse simple, les baisses de performances engendrées sont conséquentes. En effet, une telle implémentation signifie un test d'intégrité tous les secteurs de 512 octets ou 4KB (avec *extReq*). Le deuxième impact est la perte d'espace de stockage puisque les tags vont être stockés (e.g. pour un secteur de 512 octets , il faudra stocker en plus un tag de 16 octets).

Dans le cadre de cette thèse, nous n'avons pas considéré cet aspect. Cette recherche reste d'actualité puisque les mainteneurs de LUKS / cryptsetup ont proposé une discussion pour prendre en compte l'intégrité dans une version LUKS2[Bro16].

5.6.5 Conséquences de la séparation HSM / SCD

Bien que la clé de chiffrement des données (MK) soit déwrapée dans le HSM (et donc présente en mémoire) pendant l'ouverture d'un conteneur, cette dernière quitte forcément le périmètre du HSM afin d'atteindre le SCD. Cet envoi de la MK est problématique, puisque une fois envoyée au SDC, le HSM n'a plus de contrôle sur cette clé. Par conséquent, le lien fort entre le HSM et le flux de chiffrement/déchiffrement est brisé.
Ce mode de fonctionnement s'explique car le chiffrement des données n'est pas effectué par le HSM mais le SCD. Cette répartition des opérations cryptographiques se justifie par la différence de puissance entre le HSM et le SCD. Le débit pour les chiffrements symétriques du HSM est de l'ordre de 20 fois plus faible (25 MB/s contre 500 MB/s) que celui du CPU Intel Core i7 du SDC.
Nous pouvons relativiser ce point car la séparation du HSM et du SDC est dû à une implémentation de test. A terme, lorsqu'un réel HSM sera (physiquement) disponible dans une configuration réelle, le SDC et le HSM feront partie d'une seule machine. De cette manière, la clé sera présente dans la mémoire du noyau Linux du module sécurisé au sein de la même machine. Malgré tout, l'idéal aurait été de créer une dépendance réelle entre le texte chiffré et le HSM cryptographique.
Afin de permettre cette dépendance, nous proposons une évolution dans la section suivante et les avantages qui en découlent.

5.7 Ajout de mécanismes d'urgence à l'architecture TTM

5.7.1 Principes

Le principe de ces mécanismes est d'offrir des procédures d'urgence permettant d'arrêter l'exploitation des conteneurs de manière temporaire ou définitive. Ces procédures sont autorisées uniquement à l'officier de sécurité, capable de se loguer avec un rôle spécifique sur le HSM. Pour rappel, dans le fonctionnement initial de notre architecture TTM, la clé de chiffrement MK est envoyée au SDC une fois déwrappée. Or, après cet envoi le HSM n'a plus aucune responsabilité ni contrôle quant à l'exploitation des données. En d'autres termes, le SDC est capable de déchiffrer les données du conteneur de manière indépendante du HSM. Afin de redonner un contrôle au HSM, ce dernier doit être intégré dans les opérations de chiffrement et déchiffrement.
L'implication du HSM peut s'effectuer grâce au chiffrement à deux niveaux du mode XTS-AES. Le premier chiffrement de l'IV avec la clé K_2 peut être réalisé sur le HSM alors que le second (plus coûteux) s'effectue dans le SDC (avec la clé K_1). Dès lors, une suppression de la clé K_1

en cas d'urgence pourra interrompre le déchiffrement des données, le conteneur deviendra donc inexploitable. Cette solution est par contre temporaire. Une deuxième procédure, irréversible, serait de supprimer la paire de clé permettant de déwrapper la MK protégée. Dans ce cas, le conteneur deviendra inexploitable de manière définitive.
L'implémentation de cette preuve de concept n'est pas possible avec notre architecture pour deux raisons : nous n'avons pas de HSM physique, et nous n'avons pas accès aux drivers bas-niveau.
C'est pourquoi nous avons testé cette implémentation sur la carte Atmel SAMA5D3.

5.7.2 Implémentation et preuve de concept

La figure 5.6 montre le mode XTS-AES découpé en deux parties. La première partie correspond au chiffrement de l'IV avec la clé K_2 sur le co-processeur cryptographique (équivalent au HSM) alors que la deuxième représente le chiffrement des textes à chiffrer avec la clé K_1 sur le CPU principal (équivalent au SCD).

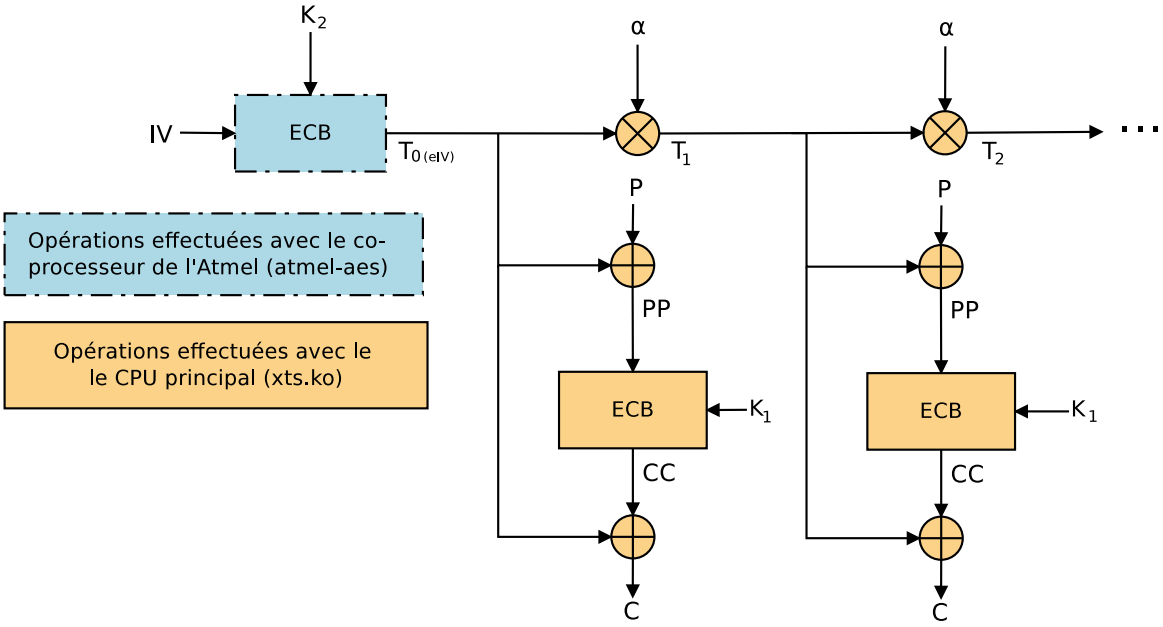


FIGURE 5.6 – Schéma de l'implémentation du mode XTS-AES sur deux niveaux.

Une vue plus détaillée (Figure 5.7) de l'implémentation met en évidence les trois modules impliquées :

— Le **module dm-crypt :** les requêtes cryptographiques sont mises en place dans ce module. Initialement, une seule requête était envoyée soit au module logiciel soit au module matériel. Dans notre cas, nous souhaitons créer deux sous-requêtes, une vers le co-processeur (étape 1) et une vers l'implémentation logicielle (étape 3).
— Le **module atmel-aes :** s'occupe du chiffrement de l'IV avec la clé K_2 (étape 2). La clé K_1 ne quitte jamais le module atmel-aes.

— Le **module xts :** s'occupe du reste des opérations du mode XTS-AES (étape 4). Le module doit être modifié afin de supprimer le chiffrement de l'IV (opération déjà effectuée dans le module atmel-aes -étape 2-).

Avec cette implémentation à deux niveaux, les clés sont présentes en partie dans le co-processeur cryptographique et en partie dans le module logiciel. Ainsi ces deux entités ont la capacité d'arrêter les opérations en supprimant une moitié de la clé de chiffrement seulement.

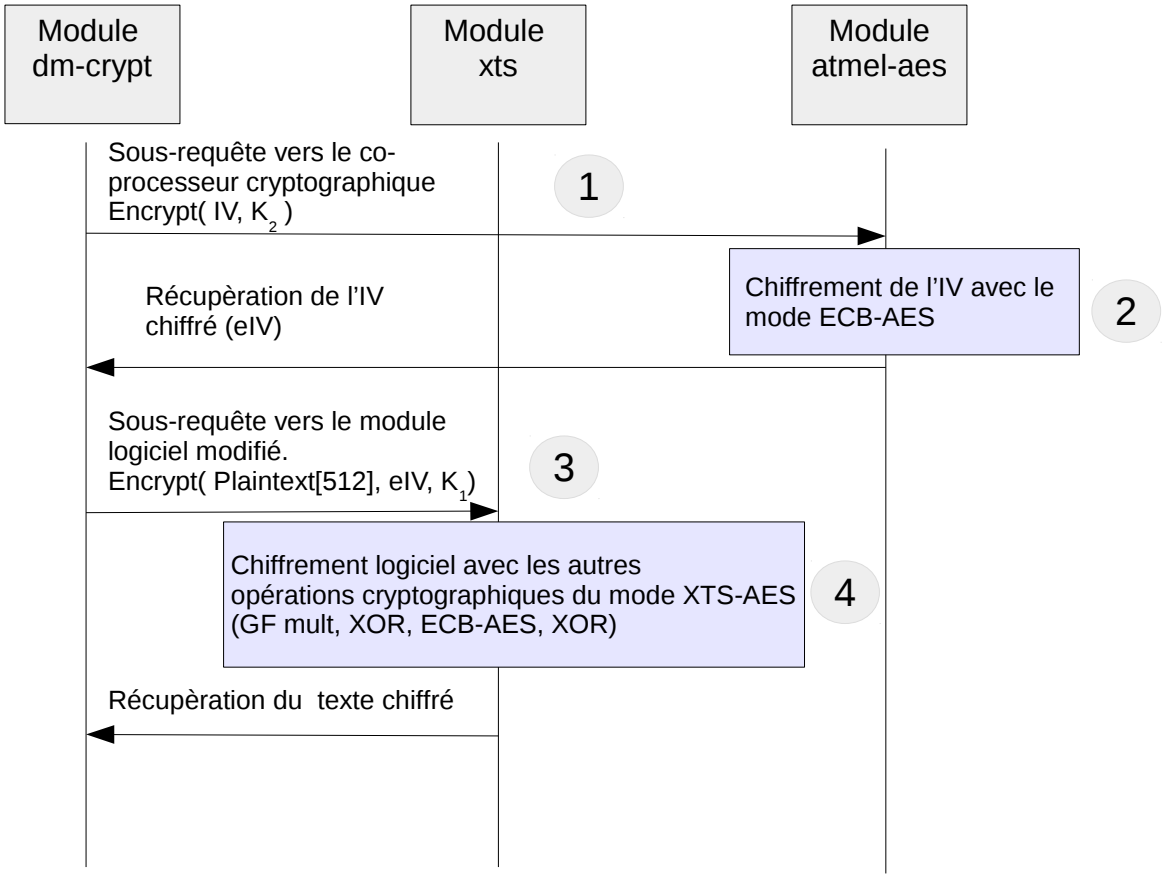


FIGURE 5.7 – Décomposition des requêtes du mode XTS-AES sur deux niveaux.

5.7.3 Résultats

Ces modifications ajoutent une dépendance voulue dans le chiffrement des données mais engendrent également un coût supplémentaire. En effet, envoyer une requête au co-processeur pour chiffrer seulement 16 octets correspondant à l'IV n'est pas avantageux en termes de performances. Nous avons établi deux mesures de débits de chiffrement : avec le mode XTS original et avec le chiffrement à deux niveaux. Les résultats du tableau 5.2 montrent une baisse des performances de 21,5 %, cette diminution peut être atténuée avec l'utilisation de *extReq*. En revanche ils attestent de la faisabilité de notre proposition. Un officier de sécurité peut arrêter le chiffrement/déchiffrement des données instantanément et à tout moment.

TABLE 5.2 – Débit de déchiffrement moyen pour des fichiers stockés dans le conteneur.

Implémentation original logicielle	**Implémentation à deux niveaux**
4.24 MB/s	3.33 MB/s

5.8 Discussion

Nous avons détaillé et démontré la faisabilité dans ce chapitre une architecture Cloud adaptée à une entreprise. Cette architecture tire partie de l'architecture du Cloud personnel en ajoutant des services évolués comme la gestion multi-utilisateurs, et la gestion des clés de chiffrement via une PKI. De plus, l'utilisation d'un HSM certifié dédié aux clés de chiffrement permet d'atteindre et de garantir un niveau de sécurité plus élevé. Nous avons décrit l'architecture complète et notamment la **séparation entre le SCD** pour les opérations coûteuses de chiffrement / déchiffrement (qui exploitent un processeur puissant) **et un HSM** certifié qui se limite aux opérations de gestion des clés. Si notre preuve de concept reposait sur un HSM virtuel distant, une solution opérationnelle pourra co-localiser ces deux entités dans un même appareil.
Enfin nous avons proposé une implémentation à deux niveaux pour donner la capacité à un officier de sécurité d'arrêter l'usage d'un conteneur en supprimant une clé sur deux du mode XTS-AES. Ceci est possible car nous avons créé une dépendance des opérations de chiffrement et déchiffrement des données avec le HSM. C'est, à notre avis, une évolution fonctionnelle importante pour les architectures de Cloud pour entreprise.

Chapitre 6

Conclusion

6.1 Conclusions

Tout au long de ce document, nous avons proposé des solutions protégeant les données des utilisateurs. Deux contraintes de départ nous étaient imposées. Premièrement, en raison d'une confiance limitée envers l'hébergeur, nous avons choisi **un chiffrement côté client.** Une deuxième contrainte était le choix de **concentrer toutes les opérations cryptographiques** dans un **Module en rupture placé entre le Client et le Cloud.**
Par la suite, nous avons analysé les solutions existantes (cf. section 3.4.1). Ces dernières sont très complexes car monolithiques. De ce fait, nous avons défini une **nouvelle architecture innovante**. Celle-ci repose sur deux principes fondateurs, **un chiffrement transparent** basé sur le **Full Disk Encryption (FDE)** avec le mode de chiffrement **XTS-AES** dédié au chiffrement bas-niveau (mode bloc). Le deuxième principe est **une distribution transparente** des données reposant sur le protocole iSCSI qui permet de voir l'espace de stockage du Cloud comme un disque local sur le Module. Les données sont, en somme, conservées de manière chiffrée sur le Cloud, déchiffrées sur le Module et accédées en clair sur le Client.
Ces deux principes sont fondamentaux car ils apportent une **modularité unique** à la solution facilitant la **ré-utilisation des briques existantes** telles que le chiffrement, la gestion des clés, l'authentification, le contrôle d'accès ou la gestion de version. Cette architecture et sa modularité ont été validées au travers de deux déclinaisons, le Cloud personnel et entreprise, le tout avec plusieurs preuves de concept.
Ces travaux ont également mis en évidence des problèmes de performances qui ont été abordés de différentes façons :

- **une implémentation hybride** du mode de chiffrement XTS-AES afin de tirer profit des co-processeurs des cartes supportant les modes de base de l'AES mais pas le mode XTS-AES ;
- **une extension des requêtes cryptographiques** afin d'exploiter pleinement les capacités des co-processeurs cryptographiques (cf. **extReq** dans la section 4.3.2) valable

pour différents modes de l'AES notamment le mode XTS-AES ;
- **une analyse des contraintes de design sur les co-processeurs.** Nous avons alors donné des lignes directrices afin de les rendre compatibles avec notre approche **extReq** ;
- **l'usage d'un équipement dédié intégrant** à la fois **un HSM certifié** pour la gestion des clés **et une unité de calcul puissante** pour les opérations de chiffrement et déchiffrement.

Enfin, nous avons proposé une approche innovante permettant d'ajouter des mécanismes d'arrêt d'urgence pour le Cloud Entreprise. Pour cela, nous avons tiré profit du chiffrement en deux étapes du mode XTS-AES et de la séparation HSM - SCD de l'architecture (cf. section 5.7).

6.2 Perspectives

Dans le cadre de nos études, nous n'avons pas pu exploiter pleinement la modularité offerte par l'architecture que nous proposons. Nous pouvons ajouter des services :

- **L'intégrité :** de façon général, les solutions FDE ne gère pas l'intégrité des données. Or, cette propriété est importante afin de protéger les fichiers contre une altération de la part du Cloud ou d'un attaquant. Il serait possible de mettre en place ce service par le biais d'un conteneur associé dédié au stockage des données d'authentifications des messages.
- **Le stockage redondant basé sur RAID :** l'architecture TTM repose sur du stockage bas-niveau. Il serait possible de tirer profit des dispositifs de stockage virtuel RAID afin d'ajouter de la redondance, de la sécurité ou de la tolérance aux pannes.
- **La gestion de version :** de nombreux outils existants peuvent se greffer à l'espace de stockage.

Le deuxième aspect à approfondir dans nos preuves de concept concerne les tests de performances. D'une part à propos du Cloud personnel et l'optimisation extReq, d'autres cartes comprenant des co-processeurs compatibles avec extReq pourraient être testés. D'autre part, dans le cadre du Cloud pour entreprise, les tests doivent s'affranchir des contraintes du réseau en terme de débit (pour rappel limité à 100 Mbit/s). En dernier lieu, des tests avec un vrai HSM physique doivent être envisagés.

Appendix A

Gestion des clés dans LUKS/dm-crypt

LUKS permet de structurer le disque en deux parties ; l'en-tête comporte les informations liées au conteneur telles que le condensé de la clé ou le type de chiffrement. La deuxième partie comporte système de fichier chiffré. LUKS permet aussi de décomposer la sécurité des clés en plusieurs niveaux par le biais de 3 clés différentes :

- **Master Key (MK)** : cette clé permet de chiffrer les données contenues dans le disque avec l'algorithme AES et un mode spécifié lors de la création.
- **Key Encryption Key (KEK)** : cette clé permet de protégé la MK, elle utilise le même mode de chiffrement que la MK.
- **passphrase/clé utilisateur** : cette clé est demandée à l'ouverture d'un conteneur, elle permet de générer avec un mécanisme de dérivation de clé KEK.

Ce système composé de plusieurs étages permet d'avoir jusqu'à 8 utilisateurs, chaque utilisateur à sa propre KEK mais la MK est inchangée. De plus un système anti-forensic permet de protéger la MK grâce à un principe d'étalement et de dépendance mutuelle.

Ce principe permet de supprimer définitivement la clé en cas de besoin. En effet la clé diffusée est répartie sur 125KB, la probabilité de supprimer efficacement quelques octets parmi ces 125KB est plus élevée que la probabilité de supprimer tous les octets d'une clé de seulement 32 octets. Ainsi grâce à la dépendance de chaque bloc, la suppression réelle et non réversible de quelques octets parmi des milliers est suffisante à empêcher la récupération de la MK. Cette protection est composée de 2 fonctions réciproques. La fonction AF_Split permet de répandre la clé sur 4000 blocs avec chaque bloc ayant pour taille la longueur de la clé. L'algorithme 9 illustre cette fonction. Les différents blocs vont de S_1 à S_{n-1} avec $n = 4000$ pour une clé de 32 octets. Les $n-1$ blocs sont générés aléatoirement, le vecteur d'initialisation est un vecteur nul. Le dernier bloc S_n dépend de la MK et du résultat de la fonction de hashage H.

$$S_n = H \oplus MK$$

Une fois le dernier bloc calculé, le tout est chiffré avec pour algorithme celui utilisé pour le

chiffrement des données et comme clé KEK. [MS13]

Algorithme 6 : AF_Split.

input : Un ensemble $S = \{s_1, s_2, \ldots, s_{n-1}\}$ de bloc
générés aléatoirement ;
en clair MK ;
KEK ;

Output : MK diffusée et chiffré Enc_AFKey

$\mathsf{IV} \leftarrow 0$
$h_1 \leftarrow \mathtt{Padding}(\mathsf{IV} \oplus s_1))$
$\mathsf{AFKey}_1 \leftarrow s_1$
for $i \leftarrow 2$ ***to*** $n-1$ **do**
  $h_i \leftarrow \mathtt{Padding}(\mathsf{hash}(h_{i-1} \oplus s_i))$
  $\mathsf{AFKey}_i \leftarrow s_i$
$\mathsf{AFKey}_n = h_{n-1} \oplus MK$
$\mathsf{Enc\_AFKey} = \mathtt{AESEnc}(\mathsf{AFKey}, \mathsf{KEK})$
return Enc_AFKey

Algorithme 7 : AF_Merge.

input : Un ensemble $\mathsf{E} = \{\mathsf{E}_1, \mathsf{E}_2, \ldots, \mathsf{E}_{n-1}\}$ de bloc chiffrés ;
KEK ;

Output : MK

$\mathsf{IV} \leftarrow 0$
$S = \mathtt{AESEnc}(\mathsf{E}, \mathsf{KEK})$
$h_1 \leftarrow \mathtt{Padding}(\mathsf{IV} \oplus s_1))$
for $i \leftarrow 2$ ***to*** $n-1$ **do**
  $h_i \leftarrow \mathtt{Padding}(\mathsf{hash}(h_{i-1} \oplus s_i))$
$MK = h_{n-1} \oplus S_n$
return MK

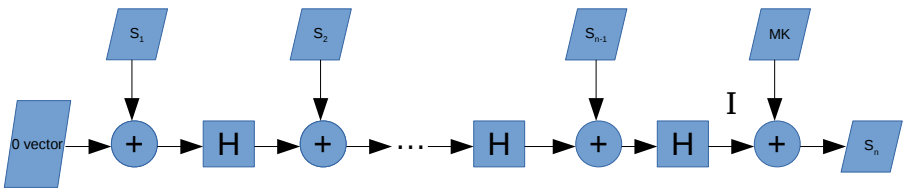


FIGURE A.1 – Protection de la MK lors de la création du conteneur.

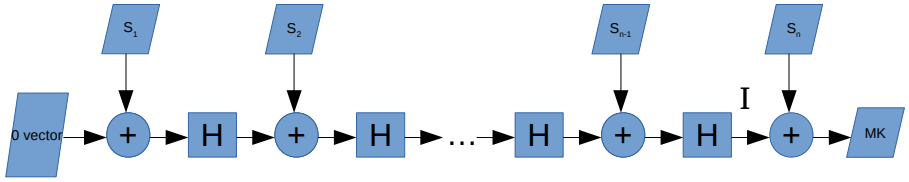


FIGURE A.2 – Récupération de la MK lors de l'ouverture du conteneur.

Traitement des données et fonctionnement global

La figure A.3 montre le fonctionnement global lors de la création du conteneur et lors du déchiffrement des données. Nous allons nous attarder sur ces deux points. La première étape est toujours la récupération de la KEK à partir de la passphrase/clé entrée par l'utilisateur. Un système de dérivation de clé appelé Password Based Key Derivation Function 2 (PBKDF2) permet d'appliquer C nombre de fois la fonction pseudo-aléatoire PRF. Le nombre d'itération est un paramètre déterminant dans la sécurité de la KEK car plus ce nombre est élevé et plus la dérivation sera chronophage ; ainsi une attaque brute-force serait difficile à appliquer. D'un autre côté le choix de l'itération dépend aussi de la puissance de l'appareil sur lequel est exécuté le chiffrement. Il n'est pas envisageable pour un utilisateur d'attendre un long moment avant de pouvoir ouvrir le conteneur. Ainsi un compromis entre la puissance de calcul de l'appareil et la sécurité attendue doit être trouvé. Une fois KEK récupérée, la MK est d́iffusée via la fonction AF_Split puis chiffrée avec KEK. La dernière étape consiste à stocker le texte chiffré dans l'entête du conteneur.
Pour l'ouverture et l'accès aux données déchiffrés, un fonctionnement réciproque a lieu. Une fois la KEK récupérée, on déchiffre la MK étalée afin d'extraire la MK seule. Ensuite nous sommes en mesure de déchiffrer les données stockées dans le conteneur.

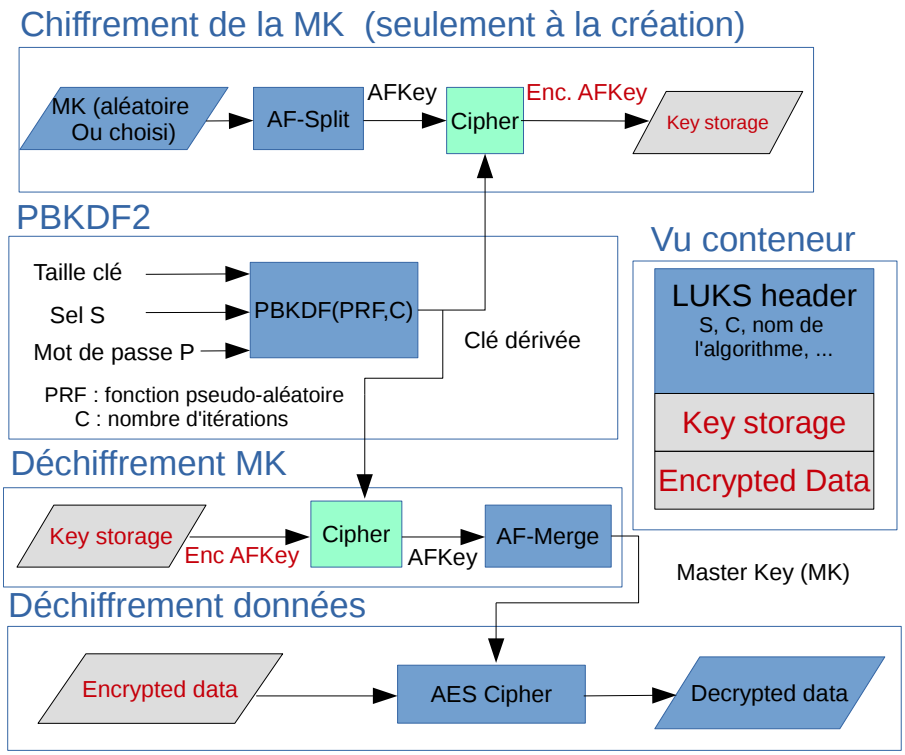


FIGURE A.3 – Vue globale sur la protection de la MK et le déchiffrement des données.

Bibliographie

[ADF+10] Kahraman Akdemir, Martin Dixon, Wajdi Feghali, Patrick Fay, Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, and Ronen Zohar. Breakthrough aes performance with intel aes new instructions. *White paper*, 2010.

[AHT13] Ian Betts Adrian Hoban, Pierre Laurent and Maryam Tahhan. Unleashing linux*-based secure storage performance with intel aes new instructions. Technical report, Intel corporation, 2013.

[AS99] Anne Adams and Martina Angela Sasse. Users are not the enemy. *ACM*, 42(12) :40–46, December 1999.

[BDK16] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2 : new generation of memory-hard functions for password hashing and other applications. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 292–302, 2016.

[BM15] Milan Brož and Vashek Matyáš. Selecting a new key derivation function for disk encryption. In *11th International Workshop on Security and Trust Management (STM)*, Sep 2015.

[Bro16] Milan Broz. The future of disk encryption with luks2. *DEVCONF*, 2016.

[CIC13] Ramaswamy Chandramouli, Michaela Iorga, and Santosh Chokhani. Cryptographic key management issues and challenges in cloud services. Technical report, 2013.

[cry16a] Cryptomator : new release, 2016. https://github.com/cryptomator/cryptomator/releases/tag/1.2.0.

[cry16b] Release new version, 2016. https://github.com/cryptomator/cryptomator/releases/tag/1.2.0.

[cry17] Keep your data safe in the cloud, 2017. https://www.cryfs.org.

[csa15] The top 10 cloud services in government that don't encrypt data at rest, 2015. https://blog.cloudsecurityalliance.org/2015/05/07/the-top-10-cloud-services-in-government-that-dont-encrypt-data-at-rest/.

[del17] Dell data protection, 2017. http://www.dell.com/en-us/work/shop/povw/ddpe-enterprise-edition.

[DH76] W. Diffie and M. Hellman. New directions in cryptography. 1976.

[Dwi05] Himanshu Dwivedi. iscsi security, 2005. Presentation - https://www.blackhat.com/presentations/bh-usa-05/bh-us-05-Dwivedi-update.pdf.

[Dwo01] M Dworkin. Recommendation for block cipher modes of operation. Technical report, National Institute of Standards and Technology, 2001. 800-38A.

[gcr17] Libgcrypt, cryptographic library originally based on code from gnupg, 2017. https://www.gnupg.org/related_software/libgcrypt/.

[gnu17] The gnutls transport layer security library, 2017. https://www.gnutls.org/.

[Gue12] Shay Gueron. Intel advanced encryption standard instructions set - rev 3.01. Technical report, 2012. https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf.

[Hec12] Philipp C Heckel. Minimizing remote storage usage and synchronization time using deduplication and multichunking : Syncany as an example. Technical report, University of Mannheim, 2012. http://www.syncany.org.

[Hor14] Taylor Hornby. Encfs security audit. Technical report, 2014. https://defuse.ca/audits/encfs.htm.Feb.

[HSH+09] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember : Cold-boot attacks on encryption keys. *ACM*, 52(5) :91–98, 2009.

[IEE08] IEEE. Ieee standard for cryptographic protection of data on block-oriented storage devices. *IEEE std 1619-2007*, 2008.

[JH10] Li Jun and Yu Huiping. Trusted full disk encryption model based on tpm. In *Information Science and Engineering (ICISE)*, 2010.

[Kal00] Burt Kaliski. Pkcs# 5 : Password-based cryptography specification version 2.0. 2000.

[Kam94] Poul-Henning Kamp. Md5 crypt, 1994.

[Kin06] Steven L Kinney. *Trusted platform module basics : using TPM in embedded systems*. Newnes, 2006.

[KL14] Nesrine Kaaniche and Maryline Laurent. A secure client side deduplication scheme in cloud storage environments. In *New Technologies, Mobility and Security (NTMS)*, 2014.

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177) :203–209, 1987.

[Mar10] Luther Martin. XTS : A mode of AES for encrypting hard disks. *IEEE Security & Privacy*, 2010.

[MCA17] Mcafee complete data protection, 2017. https://www.mcafee.com/us/products/complete-data-protection-advanced.aspx.

[MS13] Tilo Müller and Michael Spreitzenbarth. Frost. In *ACNS*. Springer, 2013.

[NC03] William D Norcott and Don Capps. Iozone filesystem benchmark, 2003. http://www.iozone.org/.

[off16] Microsoft office files encryption, 2016. http://www.makeuseof.com/tag/password-protect-encrypt-microsoft-office-files/.

[ope17] Cryptography and ssl/tls toolkit, 2017. https://www.openssl.org/.

[Per09] Colin Percival. Stronger key derivation via sequential memory-hard functions. *Document non publié*, 2009.

[Pes14] Alexander Peslyak. Yescrypt-a password hashing competition submission. *Password Hashing Competition. v0 edn 14*, 2014.

[PGB11] Zachary NJ Peterson, Mark Gondree, and Robert Beverly. A position paper on data sovereignty : The importance of geolocating data in the cloud. 2011.

[PM99] Niels Provos and David Mazieres. A future-adaptable password scheme. In *USENIX Annual Technical Conference*, pages 81–91, 1999.

[pon12] Encryption in the cloud - who is responsible for data protection in the cloud ?, 2012.

[RSA78] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2) :120–126, 1978.

[RSA01] Pkcs #11 v2.11 : Cryptographic token interface standard. Technical report, RSA Laboratories, 2001. Revision 1.

[RvH14] Ira Rubinstein and Joris van Hoboken. Privacy and security in the cloud : Some realism about technical solutions to transnational surveillance in the post-snowden era. Technical report, NYU School of Law, 2014. https://ssrn.com/abstract=2443604.

[Sac01] David Sacks. Demystifying storage networking das, san, nas, nas gateways, fibre channel, and iscsi. 2001.

[Sha49] C. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal, Vol 28, pp. 656–715*, 1949.

[SJAA+14] Marcos A Simplicio Jr, Leonardo C Almeida, Ewerton R Andrade, Paulo CF dos Santos, and Paulo SLM Barreto. The lyra2 reference guide. Technical report, 2014.

[SMG01] Huseyin Simitci, Chris Malakapalli, and Vamsi Gunturu. Evaluation of scsi over tcp/ip and scsi over fibre channel connections. In *Hot Interconnects 9*, pages 87–91. IEEE, 2001.

[SMS+04] J Satran, K Meth, C Sapuntzakis, M Chadalapaka, and Zeidner E. Internet small computer systems interface (iscsi). Technical report, IETF, 2004. https://www.ietf.org/rfc/rfc3720.txt.

[WA14] Duane C. Wilson and Giuseppe Ateniese. "to share or not to share" in client-side encrypted clouds. In *Information Security*, 2014.