Master 2
Réseaux Informatiques et Systèmes Embarqués

Présenté par

Philippe Pittoli
philippe.pittoli@etu.unistra.fr

Sécurité dans les réseaux de capteurs
Ajout de la sécurité au protocole CASAN

Encadré par

Pierre David
+33 (0) 3.68.85.45.86 TODO

Au sein de

ICube

Table des matières

Chapitre 1

Introduction

TODO
— permet au lecteur de situer rapidement le stage
— présentation de l'entreprise (synthétique) : secteur, structuration, position du stagiaire
— structuration du mémoire

Ce bilan à mi parcours du stage de fin d'études s'inscrit dans le cadre de la formation Master Informatique de l'Université de Strasbourg, spécialité Réseaux et Systèmes Embarqués (RISE). L'organisme d'accueil est le laboratoire Icube, et plus particulièrement l'équipe de recherche en réseaux, dirigée par le professeur Thomas NOËL. Cette équipe travaille aussi bien dans les réseaux de capteurs aussi communément appelé « Internet of Things », à différentes couches (lien, routage, et applicatif) ainsi que dans les réseaux d'infrastructure de l'Internet, à savoir des problématiques d'intégration de nouvelles techniques dans le routage intra et extra système autonome avec du matériel d'opérateurs réseaux.

Ce laboratoire, projet de l'Université de Strasbourg et du CNRS, est issu de la fusion de quatre anciens laboratoires : LSIIT, InESS, IMFS, IPB-LINC. Ceux-ci sont intégrés en tant que départements, formant les axes de recherche du domaine STIC-SPI : informatique, imagerie et robotique, électronique du solide et photonique ainsi que la mécanique.

Mon stage s'inscrit dans la continuité des travaux de mon encadrant, Pierre DAVID, maître de conférence. Le travail est axé autour des réseaux de capteurs et en particulier sur le protocole CASAN (Common Architecture for Sensor and Actuators Networks), développé dans l'équipe. Ce protocole de communication se base sur le travail effectué par l' Internet Engineering Task Force (IETF) sur le protocole Constrained Application Protocol (CoAP)[7].

Les réseaux de capteurs comportent plusieurs contraintes supplémentaires par rapport aux réseaux composés d'équipements plus traditionnels. Les capteurs sont composés de matériels particulièrement peu performants, que ce soit en terme de puissance de calcul, d'énergie (limitée car sur batterie ou intermittente dans le cas des panneaux solaires). Ces réseaux introduisent des contraintes à prendre en compte dans la conception, entre autre, des protocoles réseaux.

Le but de CASAN est de permettre un déploiement de capteurs qui auront une charge protocolaire minimale en terme de nombre de messages, de taille des données liées aux protocoles dans les messages, ainsi que le temps de réponse des requêtes.

L'objet de mon stage est de rajouter de la sécurité en complément du protocole CASAN.

Chapitre 2

Sécurité dans les réseaux classiques

Pour sécuriser ses échanges sur Internet, il y a un certain nombre de manières de faire. Deux protocoles sont utilisés pour créer une connexion sécurisée entre deux appareils, à savoir Transport Layer Security (TLS) qui permet de créer une couche de sécurité par dessus TCP et IPsec qui sécurise les échanges directement sur la couche IP.

On met à part les solutions de réseaux privés virtuels (VPN) et les surcouches à Internet (overlay networks) tels que Freenet, TOR ou encore GNUnet, car les buts de ces réseaux dépassent le cadre de ce mémoire (vie privée, anonymat, résistance à la censure).

2.1 Algorithmes de chiffrement, hachage et intégrité

Peu importe le protocole utilisé (TLS, IPsec) ou la surcouche logicielle au réseau internet (VPN, TOR, Freenet), tous sont fondés sur l'utilisation d'algorithmes de chiffrement, d'authentification et d'intégrité.

2.1.1 Le chiffrement

Notre but est que seul notre correspondant puisse comprendre nos messages. Pour cela nous allons utiliser du chiffrement, qui permet de rendre un texte clair incompréhensible par quiconque ne possédant pas la clé de déchiffrement.

Les algorithmes de chiffrement prennent en paramètre un texte en clair et une clé de chiffrement leur permettant de modifier le texte de façon réversible afin qu'une personne n'ayant pas la clé ne puisse pas comprendre le message. Certains algorithmes de chiffrement nécessitent également un vecteur d'initialisation, c'est à dire une série d'octets aléatoires. Ces algorithmes peuvent être classés en deux catégories.

Premièrement, les algorithmes **symétriques** où on partage un secret identique entre deux appareils. Le chiffrement et le déchiffrement se font avec la même clé.

Nous avons également les algorithmes de chiffrement **asymétriques** qui utilisent une paire de clés, publique et privée, pour chaque participant. Pour discuter avec notre destinataire nous utilisons sa clé publique pour chiffrer le message, et le destinataire utilisera ensuite sa clé privée pour déchiffrer le message.

2.1.2 L'authentification et l'intégrité

Le chiffrement seul ne suffit pas, il faut également s'assurer que le message n'a pas été modifié (intégrité), car nous pouvons modifier le message même sans connaître la clé, et de savoir à qui nous parlons (authentification). Ceci est apporté par les algorithmes de hachage, qui permettent de faire une sorte de « résumé » du contenu du message, en génèrant une suite d'octets à partir de celui-ci. Cette suite d'octets est appelée empreinte ou Message Authentication Code (MAC).

On utilise désormais des algorithmes de hachage avec une clé (HMAC), pour que seul celui qui a le droit de connaître le message en clair puisse lire le résumé, et seul celui qui possède la bonne clé puisse générer ce résumé. Cela empêche une personne au milieu de modifier le paquet et de recalculer le MAC du fichier pour indiquer qu'il n'a pas été corrompu.

Les algorithmes les plus connus sont MD5, SHA-1 et SHA-2 (famille d'algorithmes incluant SHA256 et SHA512). MD5 et SHA-1 sont considérés comme non sécurisés [**?**], [**?**] car des collisions peuvent survenir, c'est à dire qu'avec deux messages différents on arrive à la même suite d'octets.

2.1.3 Approches du chiffrement authentifié

Nous avons vu dans la partie 2.1.2 qu'il faut utiliser un algorithme de HMAC conjointement avec le chiffrement, et il y a plusieurs façons de calculer ce code et de l'intégrer à notre message.

Chiffrement puis MAC

Le texte en clair et premièrement chiffré, puis on produit le code d'authentification du message (MAC), à partir du texte chiffré. Le texte chiffré et son MAC sont envoyés concaténés. Cette méthode est utilisée par IPsec notamment. TODO REF

Cette méthode est considérée la plus sûre. Une extension pour (D)TLS a été présentée [**?**] pour y inclure cette méthode d'authentification.

Chiffrement et MAC

Le code d'authentification est ici généré à partir du texte en clair, et le texte en clair est ensuite chiffré sans le MAC. On envoie ensuite le MAC et le texte chiffré ensemble. Cette méthode est utilisée par Secure Shell (SSH). TODO REF

MAC puis chiffrement

Le MAC est généré à partir du texte en clair, puis le texte en clair et le MAC sont chiffré ensemble pour produire un texte chiffré contenant les deux parties. On envoie ensuite le texte chiffré (contenant le texte en clair et le MAC). Cette méthode est utilisée par TLS, bien qu'elle ne soit pas prouvée sécurisée (c'est à dire difficilement forgeable). TODO REF

2.2 Modes de chiffrement

Les algorithmes de chiffrement fonctionnent soit par flux, c'est à dire qu'ils prennent chaque octet du texte en clair au fil de l'eau et le chiffre, soit ils fonctionnent par blocs, c'est à dire qu'ils prennent en paramètre un certain nombre d'octets (on parle plutôt en bits) du message en clair pour le chiffrer.

Afin de chiffrer des messages longs (d'une taille plus grande qu'un seul bloc), nous avons plusieurs façons de procéder.

2.2.1 Mode Electronic Codeblock (ECB)

Ceci est le mode de chiffrement le plus simple. On prend chaque partie du texte clair que nous chiffrons avec l'algorithme de chiffrement symétrique choisi (exemple : AES). Le déchiffrement est identique, en remplaçant le texte en clair par le texte chiffré. Les figures **??** et **??** montrent le fonctionnement.

Le principal désavantage de cette méthode est que deux messages en clair identiques donneront le même texte chiffré. La conséquence à cela est qu'on ne cache pas les motifs qui peuvent être utilisés dans le texte clair dans son intégralité. La figure 2.1 est la pour illustrer le problème. La première image est l'originale, la seconde montre qu'à partir d'une image chiffrée en ECB, nous retrouvons un motif de l'image originale (nous retrouvons la silhouette du célèbre manchot). La troisième image représente ce dont nous attendons d'un chiffrement efficace, c'est à dire que nous obtenions une image sans indice sur le contenu original.

De plus, le mode ECB ne permet pas non plus de protéger contre les attaques par rejeu, puisque deux messages identiques donnent le même message chiffré.

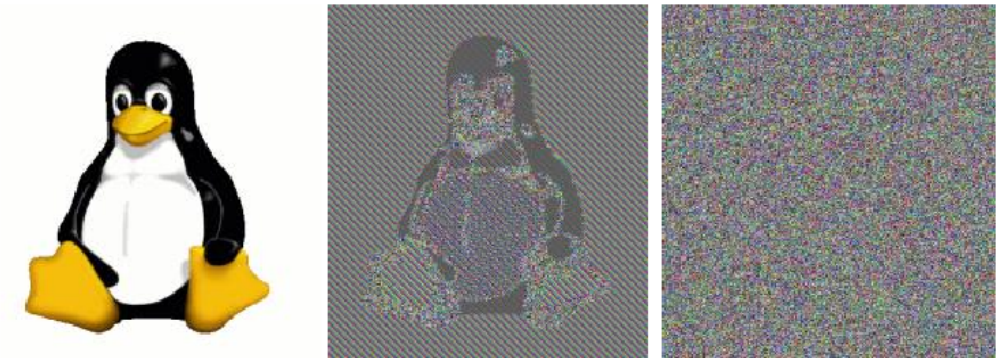
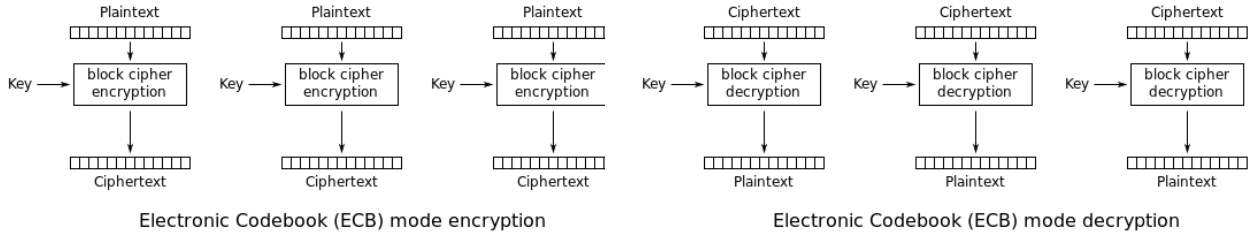
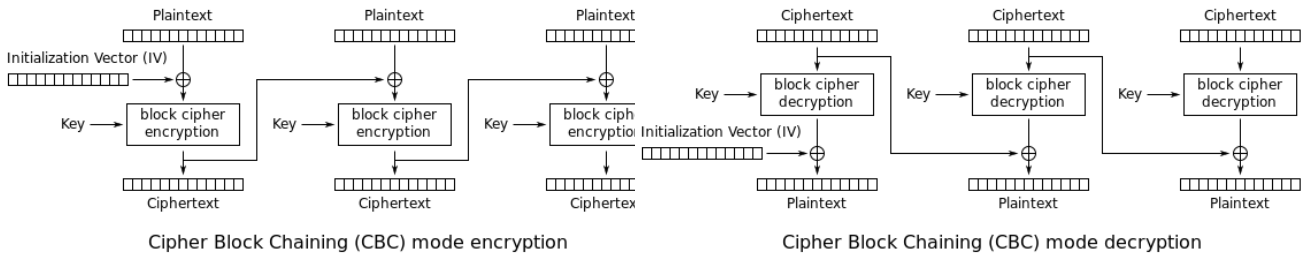


FIGURE 2.1 – Image originale, chiffrée avec ECB, chiffrée de façon sécurisée



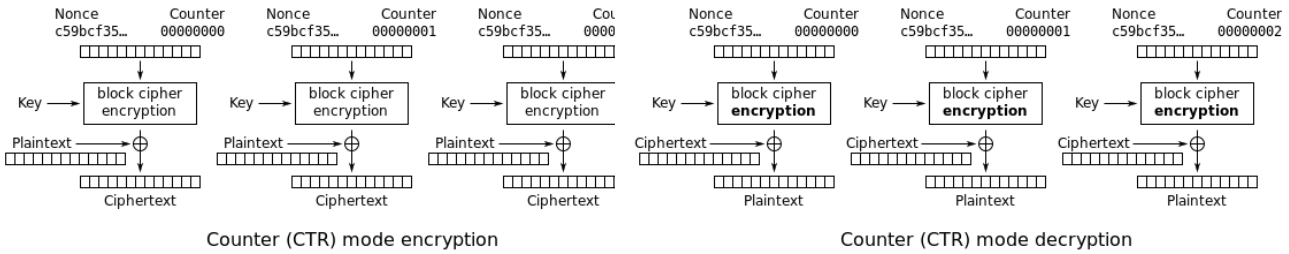
2.2.2 Mode Cipher Block Chaining (CBC)

Un autre mode possible est le chiffrement par chainage de blocs (Cipher Block Chaining, CBC). Cela veut dire que nous utilisons un vecteur d'initialisation (IV, une suite de bits arbitraires et uniques) et le premier bloc du message que nous souhaitons chiffrer, nous appliquons une opération XOR sur ces deux parties, puis nous chiffrons le bloc. Le bloc chiffré est ensuite réutilisé comme faisant partie du texte chiffré final mais aussi comme vecteur d'initialisation pour le bloc suivant et ainsi de suite jusqu'à arriver à un message chiffré complètement. Le déchiffrement est effectué en prenant un bloc du texte chiffré et en le faisant déchiffré par l'algorithme de chiffrement précédemment utilisé, puis on applique une opération XOR avec le résultat et le vecteur d'initialisation utilisé précédemment pour le chiffrement. Ce même bloc de texte chiffré est réutilisé comme le vecteur d'initialisation pour le bloc suivant. Le fonctionnement est expliqué sur les figures **??** et **??**.



2.2.3 Mode Counter (CTR ou CM)

Ce mode de chiffrement implique une nonce (une série d'octets unique, comme un vecteur d'initialisation) ainsi qu'un compteur de blocs (un simple nombre s'incrémentant). La nonce est concaténée au compteur et ceci est chiffré (algorithme symétrique) avec la clé de chiffrement, puis on applique une opération XOR avec le résultat et un bloc du message en clair. Le déchiffrement est identique, en remplaçant le texte en clair par le texte chiffré. Il est à noter que nous réutilisons la même fonction de chiffrement pour effectuer le déchiffrement. Le fonctionnement est expliqué sur les figures **??** et **??**.



2.3 Authenticated Encryption with Associated Data (AEAD)

Les algorithmes Authenticated Encryption with Associated Data (AEAD) permettent d'effectuer à la fois le chiffrement, l'authentification et l'intégrité. Il est possible également (suivant les algorithmes utilisés) d'avoir des données qui ne sont pas chiffrées, seulement authentifiées.

Un exemple est l'algorithme Counter with CBC-MAC (CCM), qui réutilise AES, le mode CBC-MAC et un compteur pour augmenter l'entropie, et qui génère une empreinte (en réutilisant AES) pour authentifier le message. L'empreinte générée et placée dans le paquet s'appelle une valeur de vérification d'intégrité (Integrity Check Value, ICV) [5].

2.3.1 Algorithme AEAD : CCM*

Le fonctionnement de CCM* est basé sur CBC-MAC en ajoutant un compteur tel que vu dans 2.2.3, qui peut se résumer de la manière suivante. TODO : mettre ces explications dans la partie « annexes » et simplifier.

Choix de départ et entrées

Tout d'abord il faut faire deux choix. La taille M du champ d'authentification, 4, 6, 8, 10, 12, 14 ou 16 octets, et la taille L du champ de taille du message, de 2 à 8 octets

Nous avons les entrées suivantes. La clé K de chiffrement, dont la taille est dépendante de celle du bloc de chiffrement. La nonce N, de taille $15 - L$, qu'il faut changer à chaque envoi de message. Le message m, de taille $0 <= l(m) < 2^{8L}$, l(m) est codé dans un champ de L octets. Enfin, nous avons les données supplémentaires authentifiées a, mais non chiffrées, de taille $0 <= l(a) < 2^{64}$.

Authentification

TODO explication plus simple.

La première étape est de créer une série de blocs de 16 octets. Suivant la taille de la partie du message qui ne sera pas chiffrée (« a »), on code sa taille différemment, voir la figure 2.4. Si $0 < l(a) \leq 2^{16} - 2^8$, la taille est codée dans 2 octets. On prend l'encodage de « l(a) » concaténé à « a », on le met sous la forme de blocs de 16 octets, en remplissant avec des 0 s'il le faut puis on met ça derrière B_0. On ajoute ensuite le message « m » avec un remplissage de 0 pour faire des blocs de 16 octets. Si $l(m) = 0$, pas de bloc. On peut tronquer le texte chiffré pour calculer T (la valeur MAC). E est la fonction de chiffrement par bloc (AES dans notre cas, aurait pu être n'importe-quelle fonction de chiffrement par bloc de 128 bits). Le dernier bloc B_n est XOR avec X_n et le résultat est chiffré avec le bloc cipher. La méthode de création du tag T à partir du tableau B est décrite dans la figure 2.7.

TODO PAGE 3

Chiffrement

$S_i := E(K, A_i)$ for i=0, 1, 2, ...

Flags = L' pour le moment.

Le message est maintenant chiffré en appliquant XOR entre le message et les l(m) premiers octets de la concaténation $S_1, S_2, S_3, \ldots$. On ne doit pas utiliser S_0 pour chiffrer le message.

On chiffre ensuite T avec le bloc S_0 et en tronquant à la taille désirée pour obtenir U, la valeur d'authentification. $U = T$ XOR first-M-bytes(S_0).

Déchiffrement

Pour déchiffrer un message, il faut avoir la clé K de chiffrement, la nonce N, les données uniquement authentifiées a, et le le message chiffré et authentifié c.

Le déchiffrement se fait en recalculant les blocs B, pour retrouver le message m et la valeur d'authentification T. Le message et les données supplémentaires authentifiées sont utilisées pour recalculer la valeur CBC-MAC et vérifier T. Par sécurité, si T n'est pas correct, on l'annonce, mais sans plus de précisions.

2.4 Partage de clés

TODO

Les échanges se font via des algorithmes de chiffrement symétriques, cependant il faut s'accorder sur une clé à utiliser pour ces échanges.

2.4.1 Préchargement de clés partagées (PSK)

Cette solution est simple, avant de déployer les appareils on donne le secret partagé à chaque participant via un canal hors bande.

Il y a plusieurs manières de les utiliser (RFC 4279) :

— échange PSK, qui n'utilise que des algorithmes à clé symétrique, ce qui est très léger et donc utile pour les environnements contraints

— échange DHE_PSK, qui utilise la PSK pour authentifier un échange Diffie-Hellman. Cela protège des attaques passives par dictionnaire mais pas des attaques actives. Cela permet le Perfect Forward Secrecy (PFS).

— échange RSA_PSK, qui combine l'authentification par clé publique du serveur (en utilisant RSA et les certificats) avec l'authentification mutuelle en utilisant la PSK.

2.4.2 Diffie-Hellman (DH)

Cette solution est fondé sur le principe qu'une multiplication de nombres premiers est difficile à factoriser.

Diffie-Hellman Ephemeral (DHE)

Cette méthode permet de créer une clé de session unique, que nous ne stockerons pas. Cela permet le perfect forward secrecy (**??**).

2.4.3 Courbes elliptiques (ECC)

Les courbes elliptiques sont fondées sur le principe qu'il est difficile pour nos ordinateurs actuels de retrouver les paramètres d'une courbe.

Les courbes elliptiques peuvent aussi êtres utilisées à d'autres fins, comme pour la gestion de certificats (signatures, vérifications).

2.5 Transport Layer Security (TLS)

TODO organiser les idées.

Le protocole TLS[4] est très utilisé sur Internet. Il sécurise le le web, mais aussi les échanges entre deux serveurs d'emails, la messagerie instantanée et bien d'autres applications. Il négocie un algorithme de chiffrement, de compression, d'échange de clés et encore un algorithme pour gérer l'intégrité des messages échangés.

Le protocole TLS permet de faire une authentification du serveur via un certificat X509, une clé PGP ou encore une simple clé publique, mais il permet également d'authentifier le client de la même manière.

Il y a plusieurs protocoles simples définis par et pour TLS. Tout d'abord il y a le Handshake Protocol (**??**), qui permet de négocier les paramètres de la session (quel algorithme et quelle clé de chiffrement utiliser), ainsi que d'effectuer l'authentification en présentant un certificat X.509 signé par un tiers de confiance. Ce protocole s'appuie sur le protocole d'enregistrement (2.5.4).

Ensuite nous avons le protocole d'enregistrement (2.5.3), qui permet de transmettre des messages en les chiffrant, assurant la confidentialité et l'intégrité. Il est utilisé par la suite pour chiffrer avec l'algorithme et la clé en question.

2.5.1 Les ensembles d'algorithmes (ciphersuites)

TODO références

Afin de décrire quels sont les algorithmes à utiliser, TLS défini des ensembles. Un ensemble est composé d'un algorithme de chiffrement, d'authentification et d'intégrité, mais également la manière dont la clé de chiffrement symétrique sera échangée et quel type de signature utilisé (si nécessaire).

Par exemple, « TLS_PSK_WITH_AES_128_CCM_8 » indique que nous souhaitons utiliser un chiffrement AES avec une clé de chiffrement de 128 bits (**??**), en mode CCM (2.3.1) avec 8 octets de HMAC (2.1.2).

Les algorithmes employés changent en fonction des avancées en matière de cryptanalyse. Par exemple, DES et IDEA ont été retirés des choix disponibles car ils ne sont plus considérés comme sécurisés. L'algorithme de chiffrement obligatoire à implémenter est RSA (partage de clés) avec AES (chiffrement). Cela permet d'avoir au moins un algorithme sur lequel s'accorder entre deux instances de TLS.

TLS gère des algorithmes de chiffrement par flux, par bloc, AEAD 2.3, comme CCM* et GCM, HMAC avec des protocoles comme SHA256 et un certain nombre de combinaisons de ces algorithmes est disponible. La liste des ensembles disponibles est sur le site de l'IANA [3].

2.5.2 Fonction pseudo aléatoire

TLS défini une fonction P_{HASH} qui crée une suite d'octets pseudo-aléatoire. Elle est basée sur un algorithme d'intégrité et d'authentification à clé (HMAC).

$$\begin{aligned} P_{hash}(secret, seed) = {} & HMAC_{hash}(secret, A(1) + seed) + \\ & HMAC_{hash}(secret, A(2) + seed) + \\ & HMAC_{hash}(secret, A(3) + seed) + \\ & ... \end{aligned}$$

Avec $A(0) = seed$

$$A(i) = HMAC_{hash}(secret, A(i-1))$$

$$PRF(secret, label, seed) = P_{hash}(secret, label + seed)$$

Cette fonction est utilisée pour générer les clés de chiffrement finales, à partir des nombres aléatoires échangés durant la négociation, ainsi que le secret partagé et un label (chaîne de caractères).

2.5.3 Poignée de mains

Nous pouvons voir une poignée de mains complète pour le protocole TLS. Les messages ClientHello et ServerHello négocient les algorithmes à utiliser. Les messages Certificate servent à envoyer les certificats pour l'authentification des deux parties. Les messages ClientKeyExchange et ServerKeyExchange servent à s'indiquer quelles clés de chiffrement à utiliser. CertificateRequest sert à indiquer comment envoyer le certificat du client, si l'envoie est nécessaire, et CertificateVerify sert à authentifier le client en lui faisant signer des données avec son certificat. Enfin, les messages ServerHelloDone et ChangeCipherSpec ne contiennent aucune information utile et servent respectivement à indiquer la fin de l'envoie des messages côté serveur et le début du chiffrement des messages. À noter que les messages ChangeCipherSpec seront supprimés dans TLS 1.3 [8].

Comme le chiffrement ne peut être activé qu'une fois que la connexion est établie, la négociation se fait sans chiffrement. Cela implique qu'un attaquant pourrait modifier les paquets et changer la

liste des protocoles disponibles par les deux parties pour en choisir un faible. Une solution simple pour contrer cela est de recommencer la négociation une fois le chiffrement en place.

Record Protocol

TODO : explications plus simples. Il manque des parties ou des parties sont trop explicites et devraient être abrégées.

Ce protocole est la base de TLS, il est utilisé par quatre autres protocoles, le Handshake Protocol (pour la poignée de mains), Alert Protocol (lorsque nous devons gérer des problèmes dans nos échanges), le Cipher Spec Protocol et Application Data Protocol pour s'échanger les données de l'application.

Ce protocole prend en paramètre les messages à transmettre, s'occupe de le fragmenter en blocs d'une certaine taille, compresse les données si demandé, gère le chiffrement et l'intégrité du message puis transmet les données.

À la réception, il vérifie, décompresse, réassemble et délivre le message à la couche suppérieure (l'application).

Pour qu'une connexion TLS puisse exister, il faut savoir si on est le client ou le serveur (connection end), une fonction pseudo aléatoire pour générer des clés à partir du secret maître. Il faut aussi un algorithme de chiffrement avec la taille des clés de l'algorithme, quel est le type de chiffrement (par blocs, par flux ou AEAD), la taille des blocs, la taille implicite et explicite des vecteurs d'initialisation (ou nonces). Il faut aussi l'algorithme MAC à utiliser pour l'authentification et l'intégrité des messages. Il faut savoir si on compresse le message, et quel est l'algorithme. Enfin, il faut un secret maître partagé entre les deux pairs (48 octets), et une valeur aléatoire de 32 octets fournie par le client, et une autre toujours de 32 octets fournie par le serveur.

Avec ces paramètres, la couche Record va pouvoir générer une clé MAC et de chiffrement ainsi qu'un vecteur d'initialisation pour le serveur et le client.

Chaque fois qu'un enregistrement est fait, on doit mettre à jour l'état, qui est composé de l'état de compression et de chiffrement (clé planifiée pour cette connexion, et toute information nécessaire pour continuer le chiffrement ou déchiffrement en cas d'algorithme de chiffrement par flux), la clé HMAC et du numéro de séquence. Ce dernier est contenu dans chaque état de connexion, il est différent en état de lecture ou d'écriture, doit être réinitialisé à 0 quand on passe en actif, ne doit pas dépasser $2^{64} - 1$ et doit être incrémenté après chaque enregistrement.

2.5.4 Record Layer

Cette couche reçoit les données non interprétées des couches supérieures dans des blocs non vides d'une taille arbitraire. Elle gère la fragmentation, qui permet de séparer des parties d'information de la couche record layer en blocs « TLSPlaintext » de 2^{14} octets ou plus petits. Cette couche gère également la compression qui permet de compresser les fragments. Elle s'occupe de chiffrer les données. Un bloc après chiffrement possède un IV (), le bloc chiffré (avec son contenu, sa MAC, son padding et la taille de padding).

2.5.5 Handshake Protocol

TODO : petit schéma avec les différents éléments cryptographiques.

Tout d'abord on échange des messages hello pour s'accorder sur les algorithmes à utiliser, on s'échange des données aléatoires et on vérifie que nous ne devons pas reprendre une session en cours. Ensuite on s'échange les paramètres cryptographiques pour permettre au client et au serveur de s'accorder sur la clé précédent la clé primaire (« premaster key »). On échange les certificats et les informations cryptographiques pour permettre au client et au serveur de s'authentifier. On génère une clé maîtresse à partir de la premaster key et des données aléatoires, voir 2.5.2 pour la génération des éléments cryptographiques. On fournit les paramètres de sécurité à la couche d'enregistrement (2.5.4). On permet le client et le serveur de vérifier que chaque pair a calculé les mêmes paramètres de sécurité, et que la poignée de mains s'est effectuée sans attaque (via la signature finale).

2.6 IPsec

TODO

K	clé de chiffrement	taille variable
N	nonce	$15 - L$
m	message	$0 <= l(m) < 2^{8L}$
a	données non chiffrées mais authentifiées	$0 <= l(a) < 2^{64}$

FIGURE 2.2 – Récapitulatif des entrées pour le chiffrement CCM*

octets : 1	$15 - L$	$15 - (15 - L)$
flags	Nonce N	l(m)

FIGURE 2.3 – B_0

octets : 1	1	4
0xff	0xfe	l(a)

FIGURE 2.4 – $2^{16} - 2^8 \leq l(a) < 2^{32}$

numéro du bit	description
7	réservé
6	Adata (1 si l(a) ¿ 0)
5 . . . 3	$M' = (M - 2)/2$
2 . . . 0	$L' = L - 1$

FIGURE 2.5 – Champ flags de B_0

octets : 1	1	8
0xff	0xff	l(a)

FIGURE 2.6 – $2^{32} \leq l(a) < 2^{64}$

$$X_1 = E(K, B_0)$$
$$X_{i+1} = E(K, X_i \text{ xor } B_i)$$
$$T = \text{first-M-bytes}(X_{n+1})$$

FIGURE 2.7 – Méthode de chiffrement, E est une fonction de chiffrement symétrique

octets : 1	$15 - L$	$15 - (15 - L)$
flags	Nonce N	Compteur i

FIGURE 2.8 – A_i

```
Client                                              Server

ClientHello                  -------->
                                                ServerHello
                                               Certificate*
                                         ServerKeyExchange*
                                        CertificateRequest*
                             <--------      ServerHelloDone
Certificate*
ClientKeyExchange
CertificateVerify*
ChangeCipherSpec
Finished                     -------->
                                           ChangeCipherSpec
                             <--------             Finished

Application Data             <------->     Application Data
```

FIGURE 2.9 – Poignée de mains complète TLS

Chapitre 3

Réseaux de capteurs

Lorsque l'on souhaite surveiller une certaine activité, on place un capteur. Par exemple la température dans une pièce, on place une sonde de température, on surveille les données et on régule notre chauffage manuellement. Une méthode pour automatiser ceci serait d'avoir une communication entre le chauffage et la sonde de température, l'un actionnant l'autre pour automatiser le procédé. On peut aller plus loin en généralisant le concept. Par exemple on peut avoir des capteurs qui envoient leurs données à un appareil qui centralisera les valeurs.

Comme dit en introduction, les réseaux de capteurs sont limités. On appelle ce type de réseaux Wireless Sensor Network (WSN) et on définit ces réseaux comme Low Power and Lossy Networks (LLNs). Ces réseaux sont de faible puissance, il y a des pertes de paquets et un certain nombre de contraintes de déploiement (portée radio, topologie, etc.). Dans notre cas, nous n'avons qu'un MTU de 127 octets.

La conception de ce genre de réseaux s'axe généralement sur la récupération des valeurs sondées, les capteurs étant dispersés de manière non linéaire, voire aléatoire dans un espace potentiellement grand avec des contraintes physiques fortes. Par exemple une forêt où on envoie des capteurs de fumée, le réseau devant remplir son rôle même par mauvais temps, même si un nœud tombe en panne.

3.1 Constrained Application Protocol (CoAP)

Ce protocole est générique, et ressemble grandement au fonctionnement bien connu de HTTP, il en est une adaptation pour des réseaux de capteurs. On identifie les ressources des nœuds (sonde ou activateur) par leur URI et on peut y accéder avec des requêtes semblables à HTTP (GET, PUT, POST, DELETE). Les réponses sont également similaires à HTTP.

CoAP utilise UDP avec un mécanisme d'identifiant de transaction pour assurer l'ordre des communications plutôt que de simplement utiliser TCP car il est trop coûteux dans un environnement contraint.

Les messages dans CoAP peuvent être suivis d'un acquittement seulement s'il est souhaité, ce qui permet de se passer d'acquittement notamment lors de broadcast.

Cependant, CoAP se base sur une couche réseau et de transport, ce qui limite la taille des données utiles (celle de l'application) dans un paquet.

3.2 Common Architecture for Sensor and Actuators Networks (CASAN)

Le protocole Common Architecture for Sensor and Actuators Networks (CASAN)[2] est axé sur une application généraliste des réseaux de capteurs. Le but est de faciliter le déploiement de réseaux de capteurs, en ayant un protocole simple, peu coûteux en terme de ressources nécessaires et en ayant un coût de maintenance faible. C'est pourquoi il intègre non seulement un protocole de communication très inspiré de Constrained Application Protocol (CoAP), mais aussi le système en définissant des rôles (maître et esclave), une manière de définir les ressources des capteurs et de les manipuler (récupérer des informations et activer des mécanismes).

CASAN se base directement sur la couche 2 du modèle OSI, contrairement à CoAP qui nécessite une couche de routage ainsi qu'une couche de transport. Cela permet de s'abstraire du mécanisme de communication physique (ethernet, radio), tout en se passant de l'adressage IP et de la couche de transport (TCP ou UDP) que nous retrouvons presque systématiquement dans les réseaux généralistes mais qui sont ici superflus (pour notre usage) et induisent une charge non négligeable dans les communications.

CASAN permet d'avoir pour un capteur plusieurs maîtres, eux-mêmes peuvent être connectés à un autre maître, qui peut faire la liaison avec le réseau externe.

3.2.1 Fonctionnement de CASAN

TODO : montrer l'échange de départ.

```
MASTER                                          SLAVE

Hello           - NON POST multicast --->
             <--- NON POST multicast -       Discover

Assoc           - CON      unicast   --->
             <--- NON ACK  unicast   -       Assoc
Hello           - NON      multicast --->

             <--- NON      unicast   -       Discover
Assoc           - CON POST unicast   --->
             <--- NON ACK  unicast   -       Assoc answer

             REVOIR ÇA
```

Premièrement il y a un apparaige, qui consiste à partager des informations entre un capteur et le maître. Cela se fait via Near Field Communication (NFC) et donc proche physiquement ce qui sera considéré comme un canal sécurisé, hors bande. Les informations échangées sont l'identité du capteur et son matériel cryptographique (clé de chiffrement). Nous ne partageons pas les informations du type algorithme de chiffrement utilisé lors de l'apparaige parce que ces informations peuvent changer au cours du temps. Notamment, il sera prévu à terme de pouvoir mettre à jour un appareil, firmware compris, et donc incluant aussi ses algorithmes de chiffrement, directement à travers une connexion sécurisée sans avoir à faire un nouvel apparaige.

Ensuite nous avons une association entre le maître et l'esclave, où l'esclave annonce les ressources qu'il possède (sondes, actionneurs). Le maître fait une liste de toutes les ressources disponibles sur les esclaves qu'il gère et un client pourra réclamer cette liste.

Enfin, un client pourra faire une requête, transmise au maître qui se chargera soit d'effectuer la requête sur le capteur, soit de récupérer la réponse dans un cache qu'il aura généré pour éviter d'épuiser les ressources. Par exemple, la température d'une pièce est une donnée qui pourrait être mis en cache pendant quelques dizaines de secondes (au moins) pour ne pas redemander inutilement cette information qui ne change que très peu aux capteurs.

Chapitre 4

Sécurité dans les réseaux de capteurs

La recherche dans la sécurité des réseaux de capteurs s'axe sur la mise en place de la sécurité malgré les contraintes fortes imposées par le matériel et sa source d'énergie limitée. La sécurité n'est pas considérée comme dans un réseau d'ordinateurs généralistes, ici le même appareil doit gérer son propre routage et s'assurer de la sécurité de celui-ci (ne pas passer ou même parler à un nœud compromis).

La sécurité dans les réseaux de capteur est très liée au déploiement. Par exemple, si on déploie une topologie en utilisant du multi sauts, alors la distribution des clés sera différente de si on a un seul saut. Il en est de même pour la diffusion d'un broadcast (même local) chiffré. Si on ajoute l'une ou l'autre de ces contraintes, il faut revoir notre mode de sécurisation du réseau.

TODO refaire cette partie. Le nombre de messages, les algorithmes à utiliser, la taille de charge utile dans chaque paquet.

4.1 Modèles d'attaque

Tout d'abord, voici les modèles d'attaque dont nous cherchons à nous protéger. Cela nous permet de différencier les manières dont les attaques sont effectuées. En les catégorisant on peut ensuite tenter de remédier aux problèmes, c'est la première étape.

En premier, nous allons nous protéger d'attaques venant de l'intérieur de notre réseau ou de l'extérieur. Peu importe si un attaquant place un nœud malicieux notre réseau ou s'il attaque depuis un ordinateur distant, il faut que notre réseau y résiste. Si on a un pare-feu qui empêche de créer un déni de service depuis l'extérieur vers notre réseau mais que nos nœuds se font attaquer en local (via une attaque de l'homme du milieu suivi d'un routage sélectif qui conduit à un déni de service), la protection n'est pas efficace.

Nous souhaitons nous protéger des attaques actives, c'est à dire que l'attaquant nous envoie des données ou en fait générer par un de nos éléments dans le réseau, lui permettant de créer un déni de service (épuisement de ressources). Nous souhaitons également nous protéger des attaques passives, où l'attaquant se contente d'écouter le réseau pour récupérer des informations sensibles.

Enfin, nous souhaitons avoir une protection contre les attaques se faisant à partir d'un nœud (contraint) et des attaques se faisant à partir d'un matériel plus conséquent (comme un ordinateur portable).

Il est exclut de se protéger contre des attaques physiques (Tampering), qui nécessiterait du matériel spécialisé ou de cacher les capteurs (et ceci est lié au déploiement, ce qui dépasse le cadre de mon travail). De même, il est exclut de se protéger contre les attaques intervenant sur la couche de lien (notamment les collisions, générés en envoyant des données alors qu'il y a déjà un émetteur sur le réseau).

4.2 Évaluation des solutions de sécurité

Différents critères sont pris en compte pour l'évaluation [10]. En sécurité, tout est affaire de compromis, la solution parfaite n'existe pas et il faut faire des choix en classant nos besoins par ordre de priorité.

Tout d'abord, on peut parler de la sécurité cryptographique des algorithmes de chiffrement

employés.

La résilience est l'assurance que le réseau continuera à fonctionner même en cas de compromission d'un certain nombre de nœuds.

L'efficacité énergétique est un critère qui s'applique tout particulièrement aux réseaux de capteurs. En effet, si chaque capteur est sous batterie, ses ressources sont limitées, donc il faut prendre en compte ce critère si on souhaite avoir un réseau qui tienne le plus longtemps possible en autonomie complète. Ce critère est corrélé au nombre d'instructions processeur, à la complexité du code et surtout au nombre de messages transmis par le capteur. La radio est très consommatrice d'énergie, et dans un réseau autonome et sous batterie, il faut veiller à ce qu'elle soit éteinte le plus longtemps possible.

La flexibilité indique si le réseau peut changer de topologie, s'il peut avoir différents déploiements. On peut prendre deux exemples concernant les algorithmes de gestion de clés de chiffrement, ce qui déterminera qui parlera à qui, et qui déterminent un certain niveau de flexibilité. Une première approche serait de laisser les nœuds tisser des liens de confiance de manière aléatoire entre eux (random node scattering), ce qui permet d'avoir une topologie quelconque (on passera outre les critères de disponibilité, certain nœuds ont une probabilité de ne pas être atteignables). Une seconde approche serait de connaître par avance le placement de tous les nœuds de notre réseau (predetermined node placement), et créer des algorithmes qui prennent en compte cette connaissance de la topologie, bien que ce soit une contrainte forte.

La tolérance de fautes (ou tolérance aux pannes) permet d'avoir les nœuds de notre réseau de moyenne qualité, fragiles ou placés à des endroits contraints avec des conditions difficiles (forte humidité, température) tout en gardant un certain niveau de service.

L'auto guérison est assez proche de la tolérance aux pannes. Cela permet d'avoir des nœuds qui tombent en panne ou qui sont compromis dans son réseau, même à des points centraux, mais de trouver un moyen de les rejeter et de retrouver une stabilité.

Le dernier critère est l'assurance, qui permet de laisser le choix à l'utilisateur de privilégier un critère sur son réseau, comme la flexibilité, la latence, le débit etc.

Dans notre cas, nous utiliserons comme critère privilégié la sécurité cryptographique, ainsi que la taille de la charge utile des messages.

4.3 Distribution de clés de chiffrement

La distribution de clés en Internet des Objets peut se faire de différentes manières. Tout d'abord, en chargeant le matériel cryptographique avant le déploiement (Pre Shared Key).

Ensuite nous pouvons utiliser du chiffrement asymétrique, en utilisant des courbes elliptiques pour améliorer l'efficacité (temps de calcul plus rapide, clés de chiffrement plus légers).

4.4 Algorithmes de chiffrement, hachage et intégrité

Les algorithmes de chiffrement pouvant s'exécuter dans un temps raisonnable sur les capteurs sont symétriques. Le chiffrement le plus répandu est AES 128 bits. Il est notamment inclut matériellement dans certains appareils, tel que le Zigduino**??**.

4.5 Datagram Transport Layer Security (DTLS)

Une déclinaison du protocole TLS a été faite pour fonctionner sur la couche de transport User Datagram Protocol (UDP) afin d'éviter la complexité de Transmission Control Protocol (TCP). Ce protocole est alors appelé Datagram Transport layer Security (DTLS)[6] [**?**]. Cela a été fait dans le but de fonctionner sur des ordinateurs plus contraints, comme un capteur.

La principale différence avec TLS est que DTLS n'a plus besoin d'une liaison fiable sur lequel s'appuyer. Cela implique quelques modifications dans le protocole, à savoir la gestion des pertes, la gestion de la fragmentation, la gestion de la retransmission et la modification de toutes les parties du protocole s'appuyant sur le numéro de séquence des paquets. Par exemple, pour le système anti rejeu on place un numéro de rang dans le payload du datagramme. Toujours pour l'anti-rejeu, la

gestion est modifiée car on peut avoir des paquets qui sont dupliqués en UDP sans que ce soit une attaque.

DTLS détermine la MTU du chemin emprunté et envoie que des enregistrements plus petits, pour éviter la fragmentation DTLS envoie des « records » entiers de taille maximale de Path Maximum Transmission Unit.

4.5.1 Messages supplémentaires

DTLS ajoute deux messages supplémentaires dans la poignée de mains : lors de la demande du client à établir une connexion, le serveur renvoie un message demandant au client de renvoyer ce même message (avec un cookie supplémentaire). Cela est utile pour éviter un déni de service impliquant un attaquant usurpant l'identité d'un client pour faire faire énormément de calculs cryptographiques au serveur ainsi que stocker des informations de connexion jusqu'à épuisement des ressources. Le message ClientHello obtient un champ supplémentaire pour le cookie. On modifie les en-têtes pour gérer les pertes ou ré-ordonnancements des paquets et on ajoute des minuteries pour détecter les pertes. Cela fait que le client doit envoyer deux messages ClientHello.

Afin de ne pas utiliser de chiffrement asymétrique nous pouvons utiliser un partage de clés PSK simple (**??**). C'est ce que nous utiliserons par la suite.

4.5.2 DTLS 1.2 en environnement contraint

Le groupe de travail DICE (DTLS In Constrained Environments) a été formé à l'IETF pour fournir des recommandations d'usage de DTLS notamment pour l'internet des objets.

Deux modèles (profiles), en fonction de l'environnement. Soit les clients sont contraints, soit ce sont les serveurs. Pour CASAN, ce sont les clients qui sont limités, le serveur (le maître) est un ordinateur classique fonctionnant avec un système d'exploitation classique.

clients TLS/DTLS contraints

On doit implémenter la reprise de session. Ça améliore les performances en environnement contraint.

Messages échangés en PSK

4.5.3 Explications sur le contenu des messages

TODO à partir de là, peu d'efforts d'écriture pour la partie DTLS, à refaire.

ClientHello , on indique au serveur quelle est la version de (D)TLS utilisée, les différents algorithmes de compression, de chiffrement et d'authentification que nous allons utiliser. TODO

ServerHello , ce message sert à échanger des informations sur les algorithmes à utiliser. TODO

ClientKeyExchange et ServerKeyExchange : c'est la partie où on s'échange les clés nécessaires au chiffrement. TODO

Les informations échangées sont :

- server_version : version du protocole SSL/TLS utilisé
- random : timestamp (4 octets) et des données aléatoires (28 octets)
- session_id : identifiant de session. Si dans le ClientHello le même champ n'était pas vide, alors on cherche une connexion précédente portant ce même identifiant dans notre table de connexions afin de la continuer, et on a un établissement de connexion abrégé. Sinon, ce champ contiendra le numéro d'une nouvelle session.
- cipher_suite : la « suite » d'algorithmes à utiliser. Cela correspond à un des éléments dans ClientHello.cipher_suites.
- compression_method : la méthode de compression choisie parmi ClientHello.compression_methods.
- extensions : liste des extensions demandées par le client et disponibles sur le serveur.

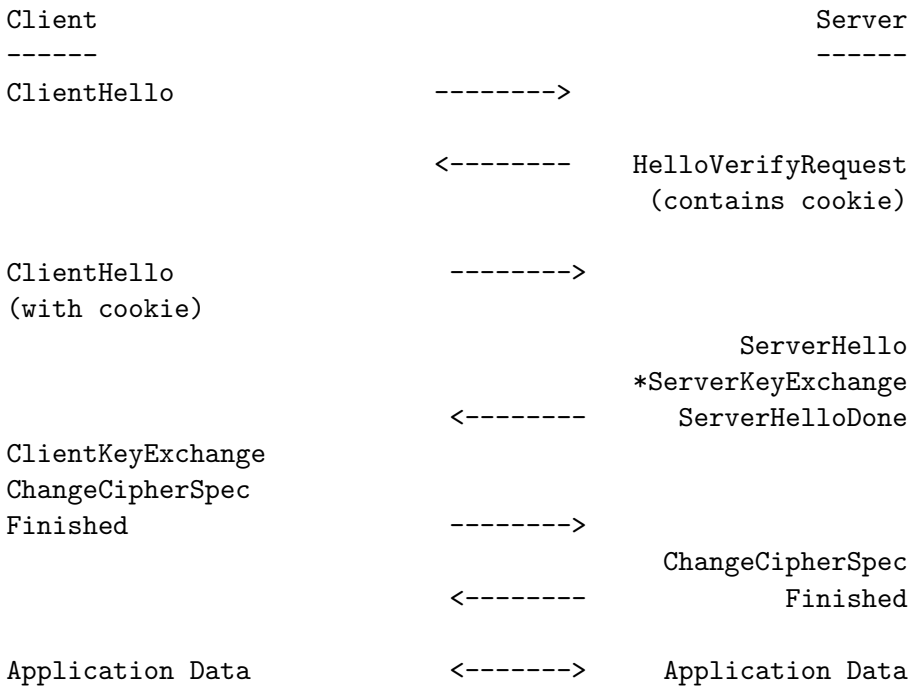


FIGURE 4.1 – DTLS PSK avec l'échange du Cookie.

ServerHelloDone , ce message sert à indiquer qu'on a fini d'envoyer notre clé (ServerKeyExchange).

ChangeCipherSpec : le message ChangeCipherSpec est envoyé par le client et le serveur pour notifier le receveur qu'on démarre le chiffrement des messages à partir de cet instant.

Finished , premier message à être chiffré et authentifié avec les algorithmes et clés choisis. À partir de ce moment on peut recevoir et envoyer des données.

$verify_data = PRF(master_{secret}, finished_label, Hash(handshake_messages))[0..verify_data_length-1]$;

Les informations échangées sont :

— finished_label : le message de fin est envoyé par le client ou le serveur, la chaîne « client finished » (respectivement « server finished »).

Recommendations du groupe de travail DICE

Page 19 : l'usage d'identités PSK courtes et PSK courtes est recommandé.

TLS_PSK_WITH_AES_128_CCM_8 à implémenter obligatoirement, HMAC avec SHA256.

TODO page 25 : on peut supprimer l'extension concernant les CA. L'extension est recommandée uniquement si on on souhaite découvrir de façon dynamique les serveurs.

Compression : il est recommandé de supprimer cette fonctionnalité car cela ajoute une certaine complexité dans le code. Autre raison est que cela permet des attaques comme CRIME. Et enfin, on a déjà des applications au dessus de DTLS qui sont déjà optimisées en terme de message pour limiter la taille des paquets.

Perfect Forward Secrecy : cela préserve la confidentialité des communications précédentes même si le secret partagé est compromis.

TODO incohérence : PSK et DHE possibles en même temps ?

On ne peut pas faire de PFS dans le cas de PSK car PFS se base sur DH. On pourrait éventuellement plus tard, quand ce sera standardisé, voir :

Schmertmann, L. and C. Bormann, "ECDHE-PSK AES-CCM Cipher Suites with Forward Secrecy for Transport Layer Security (TLS)", draft-schmertmann-dice-ccm-psk-pfs-01 (work in progress), August 2014.

RFC 7366 (chiffrer puis calculer MAC) n'est pas disponible pour les algorithmes de type AEAD.

Server Name Indication : (RFC 6066) exactement comme avec TLS sur un serveur web classique, si on souhaite avoir plusieurs serveurs sur la même machine physique possédant une seule adresse IP, cette extension est utile.

Maximum Fragment Length Negociation : (RFC 6066) offre une négociation entre le maître et l'esclave pour passer d'une taille de fragment de 2^{14} octets à 2^9 octets. Les clients **doivent** implémenter cette extension.

I-D.ietf-tls-session-hash défini une extension TLS qui lie la clé maître à un journal de la poignée de mains qui l'a créé. Les clients **devraient** implémenter cette extension.

Bhargavan, K., Delignat-Lavaud, A., Pironti, A., Langley, A., and M. Ray, "Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension", draft-ietf- tls-session-hash-03 (work in progress), November 2014.

Problèmes avec la renégociation : il est recommandé de ne pas implémenter la re-négociation. Ceci est décrit dans la RFC 5746. Si le serveur demande une re-négociation, le client doit renvoyer une alerte no_renegociation.

ChaCha20 pour le chiffrement et Poly1305 pour l'authentification et l'intégrité , bientôt une recommandation.

Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF protocols", draft-irtf-cfrg-chacha20-poly1305-07 (work in progress), January 2015.

Taille de clés.

Symmetric	ECC	DH/DSA/RSA
80	163	1024
112	233	2048
128	283	3072
192	409	7680
256	571	15360

Figure 11: Comparable Key Sizes (in bits).

Le chiffrement symétrique 128 bits est le minimum de nos jours.

Considérations sur la sécurité : il faut que nous puissions mettre à jour le logiciel. Il y a un protocole permettant de faire ceci, c'est « Lightweight Machine-to-Machine Protocol », publié par l'« Open Mobile Alliance » (OMA).

4.5.4 Pre Shared Keys, en pratique

Pre Shared Key simple

Lors qu'on utilise du PSK, les messages Certificate, CertificateRequest et CertificateVerify sont omis. Le message ServerKeyExchange peut être omis si on ne souhaite pas donner d'indice sur la PSK à utiliser.

Si on utilise que des 0 dans la partie other_secret, alors cela signifie que seule la partie HMAC-SHA1 (et non HMAC-MD5) de la PRF de TLS sera utilisée pour quand on créera le master secret. Ceci a été considéré plus élégant que d'utiliser la même clé pour HMAC-MD5 et HMAC-SHA1.

Structure des messages :

— ClientKeyExchange : psk_identity $(0..2^{16}-1)$
— ServerKeyExchange : psk_identity_hint $(0..2^{16}-1)$

Diffie-Hellman Exchange Pre Shared Key

La PSK est ici utilisée pour authentifier un échange Diffie-Hellman. Cela donne une protection contre les attaques par dictionnaire. Cela permet également d'offrir du Perfect Forward Secrecy.

Quand on utilise ce type de chiffrement, les messages ServerKeyExchange et ClientKeyExchange incluent les paramètres Diffie-Hellman. Les paramètres psk_identity et psk_identity_hint ont les mêmes significations que précédemment (ServerKeyExchange est toujours envoyé cette fois-ci puisqu'il comporte les informations sur Diffie-Hellman).

Structure des messages :

```
struct {
    opaque psk_identity_hint<0..2^16-1> : 2 o + x;
    ServerDHParams params;
} ServerKeyExchange;

struct {
    opaque psk_identity<0..2^16-1> : 2 o + x;
    ClientDiffieHellmanPublic public;
} ClientKeyExchange;
```

Cette fois-ci la premaster key est créer de la façon suivante. Premièrement on fait le calcul Diffie-Hellman, et on note le résultat Z. Maintenant on prend la taille de Z (uint16) qu'on concatène avec Z, puis la taille N de la PSK (un autre uint16) et enfin la PSK. La partie « other_secret » contient cette fois-ci Z.

RSA Pre Shared Key

Dans cette section on utilise la PSK ainsi que RSA et des certificats pour authentifier le serveur.

Le serveur ici envoie un certificat, on peut omettre le message ServerKeyExchange s'il n'y a pas de message psk_identity_hint.

Structure des messages :

```
struct {
    opaque psk_identity_hint<0..2^16-1> : 2 o + x;
} ServerKeyExchange;

struct {
    opaque psk_identity<0..2^16-1> : 2 o + x;
    EncryptedPreMasterSecret;
} ClientKeyExchange;
```

Le champ EncryptedPreMasterSecret envoyé par le client au serveur contient deux octets indiquant un numéro de version et 46 octets de données aléatoires, chiffrés en utilisant la clé publique RSA du serveur. La création de la premaster key se fait par les deux parties de la façon suivante. On concatène un uint16 avec la valeur 48 (2 octets pour le numéro de version, et 46 octets d'aléatoire), un uint16 contenant la taille de la PSK puis la PSK. Donc nous avons un premaster secret qui est de la taille de la PSK + 52 octets.

Pas de précision dans la RFC 4279 sur comment utiliser les certificats, on peut même utiliser RSA_PSK avec un certificat invalide pour uniquement avoir une protection contre les attaques passives par dictionnaire comme avec DHE_PSK.

Pas de précision non plus sur le stockage des clés.

RFC 4279 : à partir de la page 7 nous avons des recommendations sur l'implémentation et l'usage d'un PSK.

Le champ psk_identity peut être une adresse IP au format texte ou un nom de domaine par exemple (jusqu'à 128 octets, il est recommandé de supporter plus). Le champ psk_identity_hint est dépendant de l'application. La taille d'une PSK peut atteindre jusqu'à 64 octets.

4.6 Sécurité dans CASAN

Une solution pour intégrer la sécurité dans CASAN serait alors de porter DTLS sur nos capteurs. CASAN s'abstrait de la couche réseau et transport afin de maximiser la partie utile des paquets sur le réseau. On perdrait tout l'intérêt d'utiliser CASAN si on devait implémenter cette couche pour rajouter de la sécurité. Il faut donc faire en sorte d'utiliser DTLS directement sur la couche 2 du modèle OSI afin de s'abstraire de la couche réseau et transport.

Même en admettant réutiliser DTLS en l'adaptant sur la couche 2, une partie des messages induits par ce protocole reste inutile dans notre cas, car lié à des options trop gourmandes pour les réseaux de capteurs. Une implémentation simplifiée de DTLS pourrait donc être faite. Pour rappel, des messages obligatoires dans le protocole sont inutiles (ChangeCipherSpec), et seront supprimés dans la prochaine version du protocole, toujours en discussion au sein de l'IETF [8]. Aussi, certaines entêtes des messages contiennent des éléments inutiles à l'établissement pur et simple d'une connexion sécurisée entre deux appareils, mais sont là pour des raisons de compatibilité avec d'anciennes versions ou sont simplement redondants. On peut donc imaginer créer une version de DTLS bien plus légère, pour s'adapter aux réseaux de capteurs.

Une autre façon de faire de la sécurité pour CASAN serait d'intégrer les différents éléments utiles au chiffrement directement dans les messages de CASAN. Cela permettrait de se passer ainsi de l'utilisation d'une couche de sécurité impliquant des messages supplémentaire lors de l'établissement de la connexion. Un exemple simple est qu'à l'établissement de la connexion entre un esclave et un maître, on pourrait ajouter dans les messages des éléments que nous allons utiliser par la suite (comme les deux nombres aléatoire sur 32 bits utilisés dans DTLS échangés entre le client et le serveur). Cela économiserait de l'énergie, nous ferait gagner du temps à l'établissement de la connexion et nous permettrait également d'inclure éventuellement moins de code dans le binaire à mettre dans la mémoire flash des capteurs.

Il ne faut pas négliger la taille du binaire (et par conséquent la taille du code), car la flash est souvent petite sur un capteur. Un nœud comme ceux que nous avons ne supporte que 8 kB de mémoire. La simplicité du code doit être prise en compte. C'est un des critères de décision pour implémenter notre solution de sécurité, tout en reprenant globalement le fonctionnement de TLS qui est réputé pour être sécurisé. À titre d'exemple, voir le tableau qui récapitule les tailles d'implémentation de DTLS, mis en relief avec le protocole CASAN 4.2. Sans modification, l'implémentation la plus petite (et de loin la moins complète, qui ne supporte que les « suites » de chiffrement TLS_PSK_WITH_AES_128_CCM_8 et TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8 sur plusieurs dizaines) de DTLS représente déjà deux fois le nombre de lignes de code de CASAN. Cela prend de la place

Le protocole CASAN ne gère pas de multi sauts, et par conséquent son implémentation s'en retrouve simplifiée.

La figure 4.3 montre les éléments nécessaires à la sécurisation des échanges, ainsi que les besoins spécifiques à CASAN. DTLS représente ce qui est disponible dans DTLS 1.2, qui est le standard actuel. DICE est un groupe de travail de l'IETF, qui est en charge de définir des recommandations et bonnes pratiques concernant l'usage de DTLS dans un environnement contraint (quels algorithmes à choisir, éviter ou non la compression, etc). TinyDTLS est l'implémentation la plus légère de DTLS, faite pour tourner sur des réseaux de capteurs, et enfin CASANS est l'implémentation qui est prévue de la couche de sécurité pour CASAN. CASANS n'utilisera que deux messages supplémentaires pour avoir le même niveau de sécurité que DTLS car une partie des informations nécessaires à l'établissement de la connexion sécurisée seront échangés dans les messages déjà nécessaires à CASAN.

Bibliothèque	taille du code (lignes)
GnuTLS	242773
CyaSSL	112217
MatrixSSL	51470
TinyDTLS	11883
CASAN	5045

FIGURE 4.2 – Comparaison taille du code des implémentations de DTLS et CASAN

besoin	DTLS	DICE	TinyDTLS	CASANS
chiffrement	oui	oui	oui	oui
authentification	oui	oui	oui	oui
PSK	oui	oui	oui	oui
négociation algo de chiffrement	oui	oui	oui	oui
possibilité de ne pas tout chiffrer	oui	oui	oui	oui
nombre de messages pour la négociation (minimum)	6 (10)	6	6	2
taille du protocole dans les messages	29	29	29	19

FIGURE 4.3 – Besoins, protocoles, implémentations

Chapitre 5

Ma contribution

5.1 Choix du protocole de communication

5.1.1 Protocole de négociation

Pour que chaque connexion soit chiffrée de manière unique et ainsi limiter les attaques statistiques, il faut avoir une clé partagée de chiffrement unique. Pour cela, on peut s'échanger des nombres aléatoires qui participeront à créer un secret unique. Ceci est le même procédé que l'échange effectué par l'algorithme Diffie-Hellman, sauf que nous ne créons pas un grand nombre impossible à factoriser (les temps de calcul pour l'algorithme de Diffie-Hellman sont conséquents sur du matériel contraint). Ici on utilise ces nombres aléatoires sont utilisés par les deux parties conjointement avec leur secret partagé via l'appairage pour créer une clé unique de session.

Il serait possible de choisir quel seront les algorithmes à utiliser directement lors de l'apparairage de CASAN3.2.1. Nous pourrions convenir d'un algorithme de chiffrement, de compression, de vérification d'intégrité, tout comme il est possible avec TLS et IPsec. Dans ce cas, plus besoin d'une négociation se passant à chaque connexion, et donc d'un protocole complexe pour gérer nos connexions sécurisées. Cependant il faudrait toujours se partager ces nombres aléatoires, pour créer des sessions uniques, et effectuer à nouveau cet échange à chaque redémarrage de nos appareils. De même il faudrait intégrer un algorithme de PRF**??** pour créer la clé de session à partir de données aléatoires et de notre secret partagé. Tout ceci est déjà inclut dans des standards tel que TLS.

Il est souhaité à l'avenir de pouvoir mettre à jour le firmware de nos appareils à distance. Pour cela il sera possible à partir d'une connexion sécurisée de fournir le logiciel à écrire en ROM de l'appareil, lui donnant par la même occasion les informations telles que celles que nous avons lors de l'appairage CASAN.

5.1.2 TLS plutôt que IPsec

Voir [9] DTLS plus simple, plus léger
DTLS se base sur la couche UDP
IPsec
TODO expliquer pourquoi TLS plutôt que IPsec

5.2 Portage de DTLS sur Zigduino

5.2.1 Les bibliothèques et définitions inexistantes

Pour porter le programme TinyDTLS sur zigduino, il a fallut penser à émuler chaque fonctionnalité non disponible via avr-gcc.

J'ai commencé par **pthread.h** (gestion de processus légers) est une bibliothèque non existante pour le zigduino, où un seul binaire est exécuté, sans système d'exploitation (donc pas de noyau pour gérer les priorités entre processus, un seul processus existe). Cela a été assez simple à gérer, puisqu'une seule partie du code de TinyDTLS utilise les pthreads, et j'ai simplement commenté cette partie (via **#ifdef WITH_ARDUINO** et **#endif**).

Ensuite il y a une partie de gestion de fichiers, pour récupérer par exemple les clés de chiffrement.

Même technique, si on est sur arduino je n'utilise simplement pas les mêmes méthodes, on passe par un tableau contenant la clé au lieu de récupérer la clé sur le disque.

Il y a une gestion des sockets via **sys/socket.h**, **netinet/in.h** et **arpa/inet.h**, que j'ai retiré de la compilation de la même manière. Une partie de ce code gère les adresses IPv4 et IPv6, notamment dans les structures représentant les pairs auxquels on se connecte. Voici un exemple de code après l'adaptation à arduino.

```
#ifdef WITH_ARDUINO
// gestion d'adresses sur 2 octets
typedef struct {
  size_t size;
  u_int16_t addr;
  u_int8_t ifindex;
} session_t;

#else
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

// gestion des adresses IPv4 et IPv6
typedef struct {
  socklen_t size;               /**< size of addr */
  union {
    struct sockaddr     sa;
    struct sockaddr_storage st;
    struct sockaddr_in  sin;
    struct sockaddr_in6 sin6;
  } addr;
  uint8_t ifindex;
} session_t;
```

5.3 DTLS simplifié

TLS n'a pas été prévu pour des environnements contraints, et il comporte des messages échangés sans information utile, ainsi que des parties redondantes dans les messages. DTLS étant un simple portage de TLS sur UDP, il a aussi ces problèmes.

5.3.1 Messages inutiles

DTLS comporte un certain nombre de messages qui n'ont aucune information utile. Tout d'abord le message *ServerKeyExchange*, qui n'est pas utile dans notre cas, nous n'avons qu'un serveur. Le message *ServerHelloDone* indique que le serveur a fini d'envoyer les éléments nécessaires au client pour chiffrer la connexion. Si nous n'utilisons qu'un partage de clés PSK, que nous n'avons pas besoin de ServerKeyExchange, alors il ne reste que ServerHello et ce message, qui devient alors inutile (une fois la réception de ServerHello on a toutes les informations qu'il nous faut). Le message *ChangeCipherSpec* indique que les prochains messages seront chiffrés avec l'algorithme précédemment négocié, pas besoin d'un message pour cela, le reste est forcément chiffré puisque c'est la suite normale de la négociation. La figure 5.1 montre le récapitulatif des messages que nous pouvons supprimer.

5.3.2 Informations redondantes

L'entête de la couche d'enregistrement (Record Layer) fait 13 octets, la partie explicite de la Nonce fait 8 octets et l'ICV2.3 fait 8 autres octets. Cela fait 29 octets ajoutés par message pour sécuriser sa connexion.

Cependant l'entête comporte les informations suivantes :

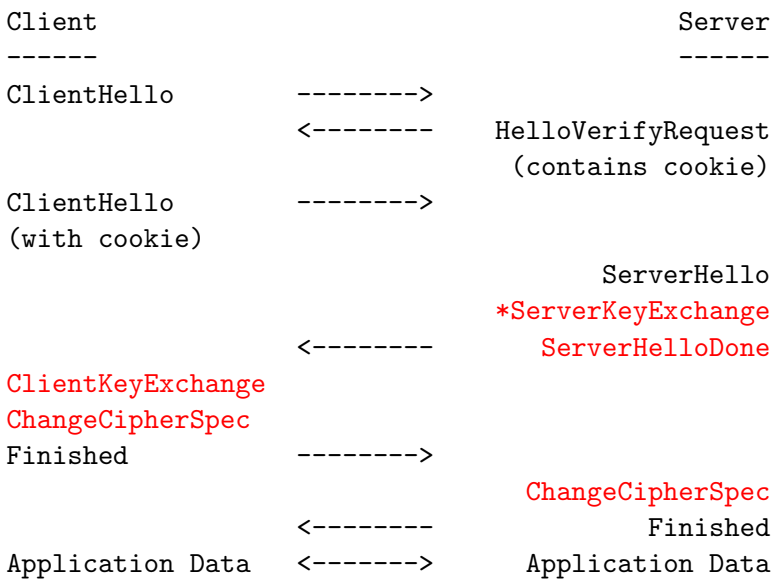


FIGURE 5.1 – Échange DTLS simplifié

— 1 octet pour le type de contenu
— 2 octets pour indiquer la version du protocole utilisé
— 2 octets pour « epoch » (temps)
— 6 octets pour le numéro de séquence
— 2 octets pour la taille du message

La nonce utilisée pour les algorithmes AEAD est de 12 octets, dont 4 octets de « sel » implicite généré via la PRF et les 8 octets contenus dans le paquet. Dans DTLS, la nonce explicite est composée des 2 octets du champ « epoch » et des 6 octets du numéro de séquence. DTLS a donc des informations redondantes dans les messages. Ceci est confirmé par le draft 10 du profile DICE annexe B, « DTLS Record Layer Per-Packet Overhead ». Le document rend également obligatoire (MUST) l'usage du champ epoch et du numéro de séquence pour créer la partie explicite de la nonce.

Dans une version simplifiée de DTLS nous pouvons enlever ces champs inutiles et ainsi gagner 8 octets. Le surplus utilisé dans les messages pour la sécurité une fois la simplification du protocole faite est de 21 octets, soit un gain d'un peu plus de 6% de place libre dans les messages pour l'application. Ceci casse la compatibilité avec DTLS, mais

5.4 CASANS

Une fois avoir simplifié DTLS, on a une connexion sécurisée qui demande une négociation en 6 messages et 21 octets sont pris dans chaque message pour assurer la sécurité. Ceci est indépendant de l'application qui s'exécute par dessus DTLS et est également indépendant des couches inférieures (physique, lien, réseau, transport).

5.4.1 Messages

TODO : vérifier cette partie (Assoc qui contiendra le random, identification du client déjà fait avec CASAN ?).

Il est possible de gagner quelques messages et quelques octets supplémentaires. Premièrement les informations partagées dans les messages *ClientHello*, *ServerHello*, *ClientKeyExchange* et *Finished* peuvent l'être dans des messages de CASAN. L'identification du client se fait via l'appairage (où on lui donne son identifiant) puis via le protocole CASAN on peut l'identifier, et on peut s'échanger des valeurs aléatoires dans le message ASSOC lors de l'association maître-esclave.

Cela permettrait de n'envoyer que deux messages supplémentaires par rapport à CASAN actuellement, pour avoir le même système de cookies que dans DTLS et ainsi éviter des attaques par déni de service sur le maître.

5.4.2 Contenu des messages

TODO réfléchir un peu plus à la troisième solution.

Pour concevoir la sécurité dans CASAN, trois solutions sont possibles. La première idée (5.2) serait d'avoir un type supplémentaire de message dans CASAN, sous la forme d'une option CoAP, avec le contenu d'un message DTLS. La partie applicative serait entièrement dans ce message. Ainsi, une fois l'association faite nous pourrions chiffrer la partie applicative.

La seconde idée serait d'avoir une option de CASAN toujours sous la forme d'une option CoAP pour le chiffrement mais au début de tous les messages. Le contenu de cette option serait identique à la première solution, mais cette fois-ci tout le message CASAN serait chiffré, et un attaquant ne saurait pas même le type de message émis.

Il est possible de gagner deux octets supplémentaires en ajoutant un octet avant la couche CASAN, ce que montre la figure 5.4. On ajoute un octet supplémentaire avant les messages CASAN indiquant la version du protocole utilisé sur 3 bits, le type de message DTLS sur 5 bits. Ainsi, 2 octets supplémentaires sont libérés pour les messages.

POST/GET/	CON	options CoAP avec paramètres DTLS	/temp

FIGURE 5.2 – Proposition 1 : DTLS est une simple option

option CoAP avec paramètres DTLS	POST/GET/	CON	/temp

FIGURE 5.3 – Proposition 2 : mettre une option CoAP au début pour chiffrer le reste

un octet indiquant le type de message et les paramètres DTLS	POST/GET/	CON	/temp

FIGURE 5.4 – Proposition 3 : tout chiffrer, un octet de signalement au début des messages

5.5 Évaluation

TODO mettre les temps de transmission et de calcul.

5.5.1 Évaluation théorique

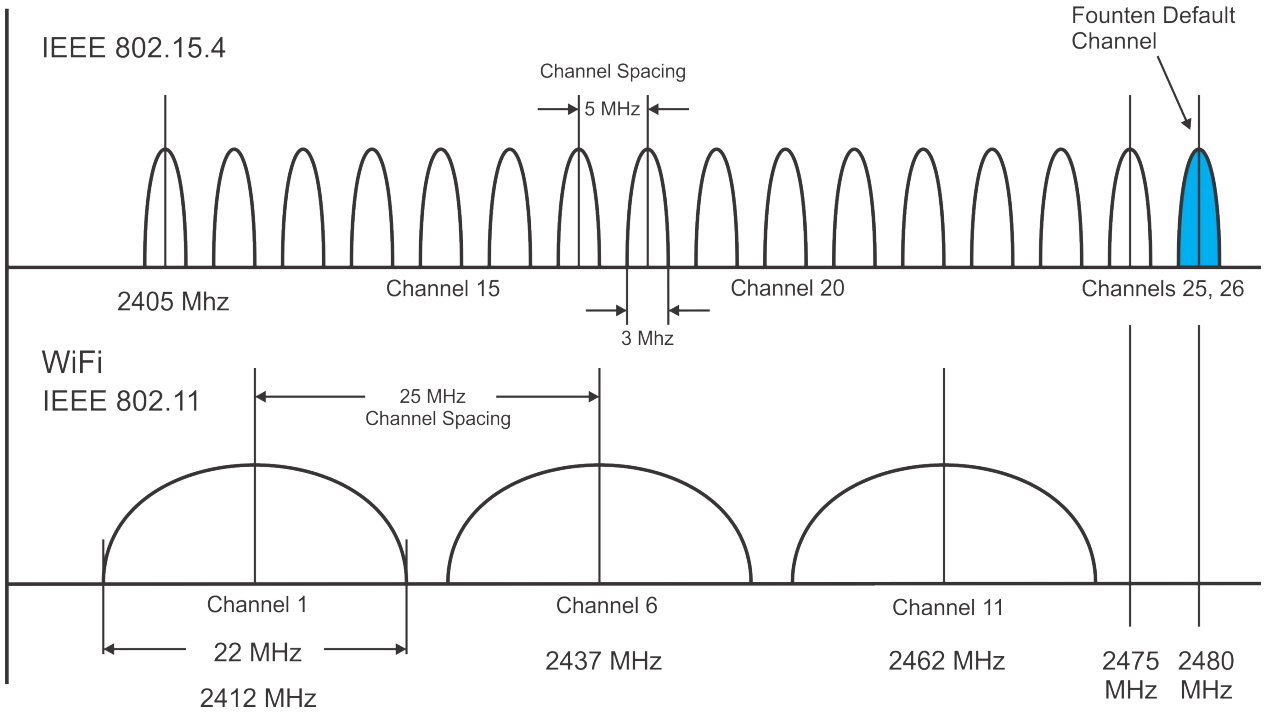
Temps de transmission

Nous travaillons via la couche de lien 802.15.4 de 2006 en mode pair à pair, et le lien physique Offset Quadrature Phase Shift Keying (O-QPSK) dans la bande de fréquences 2450 MHz. Cela peut créer des collisions avec la plage de fréquences utilisée par 802.11, tel que montré dans le graphe **??**.

Temps de calcul

Le chiffrement AES 128 matériel est 4 fois plus rapide qu'en passant par le processeur. Il atteint des vitesses de TODO Kbps matériellement contre TODO Kbps via le processeur.

En mode CCM (**??**), nous chiffrons deux fois chaque bloc du message à envoyer, donc nous avons un débit sur notre matériel qui devrait être de TODO Kbps.



5.6 Difficultés rencontrées

Le plus grand problème rencontré a été de détecter et de corriger une erreur pour le zigduino.

Le zigduino ne présente aucun système de détection et de correction d'erreurs autre que le programme *Atmel Studio* qui n'est pas disponible pour mon système d'exploitation.

Je venais d'adapter une partie de TinyDTLS pour le zigduino, et d'écrire un programme de test très basique (en terme arduino cela s'appelle un « sketch »). Le problème était le suivant : une fois la réception d'un message, l'appareil redémarrait. C'est une action qui se passe lorsqu'il y a une erreur de mémoire par exemple, et ce fut naturellement mon idée de départ. Sachant que la bibliothèque utilisée est conséquente en terme de lignes de code (12 000 lignes) et que des tampons sont utilisés, j'ai souhaité vérifier la mémoire disponible dès le démarrage de mon application.

Pour cela, pas de fonction toute prête de disponible, le site d'Arduino renvoie vers une bibliothèque à inclure à son application[1]. La fonction **freeMemory** indique que presque la moitié de la mémoire est encore disponible (environ 8000 octets). Lorsque je n'utilise pas les fonctions de la bibliothèque DTLS et que je reçois un message j'ai toujours un redémarrage qui se fait. Donc cela ne vient pas d'un manque de mémoire.

Autre piste, un problème avec un appel d'une fonction qui poserait problème, alors pour savoir où était cette fonction j'ai affiché des messages à certains endroits dans le code, notamment autour de la procédure utilisant la radio. Résultat non concluant : l'erreur survient semble-t-il de façon aléatoire. Parfois même au milieu de l'affichage d'un message.

Ensuite avec l'aide de mon maître de stage nous avons essayé de voir ce qui pouvait bloquer. Nous avons regardé le binaire généré, voir quelles étaient les interruptions utilisées, et ce qu'elles faisaient. Nous avons détecté l'interruption 60 qui est appelé lors de la réception d'un message.

Pour vérifier que le problème ne venait pas de la taille du code, et donc du binaire généré (environs 45k), j'ai créé une bibliothèque pouvant générer un binaire d'une taille similaire (un peu plus même, 60k). Pour cela il a fallut créer plus de mille fonctions, dont des fonctions de mille cinq cents lignes d'instructions, qui ne seraient pas supprimées par le compilateur via une option d'optimisation qui supprimerait le code inutile. Pour cela, les instructions faisaient appel à une fonction renvoyant un nombre aléatoire. Même en ayant un programme aussi lourd, le zigduino n'a pas d'erreur. Le problème semble donc venir de la bibliothèque TinyDTLS adaptée.

Afin de savoir quelle partie du code créait des erreurs, j'ai commencé par commenter le code de toutes les fonctions du fichier **dtls.c** qui est le cœur de la bibliothèque via **#if 0** et **#endif** ; puis j'ai activé chaque partie du code petit à petit. Il n'y avait pas de manière plus aisée à ma connaissance pour faire ceci. Une fois cela fait, la seule partie du code à commenter pour que le

programme semble fonctionnait se trouvait dans le fichier **aes.h**, lors de la définition d'une structure imbriquée (une structure est définie, puis une autre structure directement après la réutilise). J'ai souhaité vérifier que le problème venait bien de là, en réutilisant uniquement cet extrait de code ailleurs, sur un exemple qui fonctionne sans inclure la bibliothèque DTLS entière, et le problème ne se manifeste pas.

Enfin, avec mon maître de stage nous avons refait une réunion, en repartant sur la piste de la routine d'interruption, potentiellement erronée. Nous savons regardé les valeurs de la bibliothèque logicielle autour, qui se charge de transmettre et de stocker des messages, et nous avons trouvé que la bibliothèque se chargeait de garder en mémoire une centaine de messages, et donc se réservait un espace mémoire presque aussi grand que la mémoire de l'appareil utilisé pour les tests. Problème résolu. Il aurait été aisé de trouver cette erreur si la routine de test de mémoire donnait les valeurs attendus. Lors d'une corruption de la mémoire, tout devient difficile à corriger, on ne peut plus faire confiance aux valeurs affichées, et nous n'avons pas accès à des outils sophistiqués (tel que gdb) sur ce type de matériel pour nous aider à cette tâche.

Chapitre 6

Bilan critique

TODO
— Comparaison cahier des charges et objectifs atteints
— Difficultés importantes
— Critique de votre propre travail (ce n'est pas dévaloriser de dire qu'on a muri) : si je devais le refaire...
— Auto-évaluation : mais rester humble sur les compétences acquises (c'est du ressort du jury)

Le but de mon stage est de concevoir (et faire des choix) puis d'implémenter une solution de sécurité pour CASAN. J'ai repris une solution existante et je l'ai adapté. Il me reste un peu de temps pour terminer l'implémentation.

La plus grande difficulté a été d'assimiler autant de notions (sécurité dans les réseaux de capteurs, dans les réseaux classiques, 802.15.4) en peu de temps, et éviter me disperser par la même occasion. J'ai passé du temps à faire divers petits programmes (création d'un graphe de dépendances entre fichiers C par exemple), mais surtout du temps à comprendre certaines partie des algorithmes de chiffrement bien que ça ne soit pas utile pour faire mon travail. Si je devais refaire ce stage, je me concentrerais sur les parties les plus importantes pour arriver au résultat en premier.

Auto-évaluation ?

Chapitre 7

Conclusion

TODO
- Valorisation
  - de ce que ça a apporté à l'entreprise (réalisation)
  - de ce que ça vous a apporté (acquisition de compétences)
- Perspectives
- Bilan personnel : qu'est-ce qui vous a motivé ? (attention à ne pas être naïf)

Ce qui m'a motivé dans ce stage a été de découvrir un peu plus en profondeur le domaine de la sécurité. Nous avons eu un cours sur le sujet, mais nous n'avons pas vu les protocoles de communication comme TLS dans le détail.

Chapitre 8

Glossaire

TODO : des mots qui ne sont pas cités dans le texte, des mots dans le texte qui ne sont pas ici.

— KDF (Key Derivation Function) : permet à partir de créer plusieurs clés de chiffrement à partir d'une seule
— Key stretching : rend une clé faible plus sécurisée
  — voir : PBKDF2, bcrypt, scrypt
— LLN (Low Power Lossy Networks) :
— RPL (Routing Protocol for LLN) :
— MPL (Multicast Protocol for LLN) :
— positionnement node-centric : émetteurs publics avec location connue
— positionnement infrastructure-centric : on devine la location avec les communications inter nœuds
— DH (Diffie–Hellman) : méthode d'échange de secret partagé sur un canal non sécurisé
— ECDH (Elliptic curve Diffie–Hellman) : idem, mais avec des courbes elliptiques
— AES (rijndael, Advanced Encryption System) : algorithme de chiffrement symétrique
— MAC (Message Authentication Code) : est un code accompagnant des données pour assurer leur intégrité, en se basant sur le contenu du message et une clé secrête.
— CBC-MAC (cypher block chaining message authentication code) : technique pour construire un MAC à partir d'un bloc chiffré. Le message est chiffré avec un algorithme de chiffrement par bloc pour créer une chaîne de blocs tel que chaque bloc dépend du bloc chiffré précédent.
— CCM (Counter with CBC-MAC) : ajout d'un compteur de blocs à l'algorithme CBC-MAC.
— Galois/Counter Mode (GCM) : mode d'opération pour un chiffrement symétrique par blocs, répandu car efficace et rapide. Ce mode de chiffrement authentifié permet d'authentifier, de vérifier l'intégrité et d'assurer la confidentialité. GCM is defined for block ciphers with a block size of 128 bits. Galois Message Authentication Code (GMAC) is an authentication-only variant of the GCM which can be used as an incremental message authentication code. Both GCM and GMAC can accept initialization vectors of arbitrary length.
— Pre Shared Key (PSK) : lors de l'établissement d'une connexion sécurisée, il faut échanger ou créer une clé symétrique qui sera utilisée pour chiffrer la session. Le mode PSK permet d'avoir une clé déjà fournie dans les deux bouts de la connexion, qui servira soit à chiffrer directement la connexion soit à créer la clé de session.

Bibliographie

[1] Arduino. *"Available Memory on Arduino"*.

[2] P. "David and T." Noël. "casan : A new communication architecture for sensors based on coap". 2015.

[3] IANA. Paramètres tls.

[4] IETF. *"TLS 1.2"*, 2008.

[5] IETF. *"AES-CCM Cipher Suites for Transport Layer Security (TLS)"*, 2012.

[6] IETF. *"DTLS 1.2"*, 2012.

[7] IETF. *CoAP*, 2014.

[8] IETF. *"TLS 1.3"*, 2015.

[9] Nagendra Modadugu and Eric Rescorla. "the design and implementation of datagram tls". In *IN PROC. NDSS*, 2004.

[10] Yong Wang, G. Attebury, and B. Ramamurthy. A survey of security issues in wireless sensor networks. *Communications Surveys Tutorials, IEEE*, 8(2) :2–23, Second 2006.

8.1 Annexes : à venir