Remerciements

Table des matières

1 Introduction **3**
1.1 Contexte du projet 3
1.2 LoRaWAN et la problématique de sécurité 3

2 Organisation **4**
2.1 Méthode de travail 4
2.2 Logiciels utilisés 4
2.3 Répartition des tâches 5

3 Réalisation et Tests **6**
3.1 Démonstrateur 6
3.2 Nœud 7
3.2.1 Surface d'attaque du nœud 7
3.2.2 Nœud LoRaWAN avec une carte B-L072Z-LRWAN1 9
3.2.3 Sécurisation carte STM32 10
3.2.4 Problèmes rencontré 13
3.2.5 Tests Unitaire 14
3.2.6 Tests Intégration 15
3.2.7 Conclusion 15
3.3 Passerelle 15
3.3.1 Tests Unitaire 15
3.3.2 Tests Intégration 16
3.3.3 Problèmes rencontré 16
3.3.4 Conclusion 16

4 Conclusion **17**

A Première Annexe **19**

Chapitre 1

Introduction

L'Internet des objets (IoT, en anglais) est un paradigme dont les premiers déploiements ont quelques années (voire plus, si l'on parle de réseau de capteurs). D'un point de vue sécurité, l'IoT a une surface d'attaque très importante, du fait du nombre de technologies, de protocoles, du type de déploiement et du nombre d'acteurs différents. Ce projet s'applique aux réseaux d'objets connectés longue porté du type LoRaWAN (Long Range Wide Area Network).

1.1 Contexte du projet

Dans le cadre de notre première année de Master 1 (Cyber-sécurité des Systèmes Embarqués) nous avons eu l'occasion de réaliser un projet d'une durée de 2 mois en parallèle de nos cours. Nous avons choisit le projet "Création d'un réseau LoRaWAN sécurisé" car il correspond à des technologies mis en œuvre pour l'IoT, qui nous intéresses.

1.2 LoRaWAN et la problématique de sécurité

Chapitre 2

Organisation

2.1 Méthode de travail

Pour gérer les modifications du projet nous avons utilisé un outil de versionning appelé *Github*, celui-ci nous permet de stocker les programmes ainsi que les différents documents nécessaire au projet. Pour nous organiser tout au long de la période du projet nous avons établi un diagramme de *GANTT*. Nous avons gardé à jour pendant toute la durée du projet. Pour avoir une gestion de projet plus précise (tâches à effectuer chaque semaines), nous avons utilisé l'onglet *Project* de notre *repository Github*, qui utilise un tableau de *Kanban*. Dans cet onglet nous indiquions pour chaque semaine, les différentes tâches à faire. Les tâches ont 3 états *À faire*, *En cours* et *Fini* nous déplacions les taches d'une colonne à l'autre en fonction de leurs statu, nous ajoutions aussi de nouvelles tâches au cour de la semaine si besoin.

Nous avons choisi une approche en spirale (méthode *Agile*) pour notre organisation vis à vis du développement. En effet, sur les conseils de notre encadrant, ce modèle nous permet de d'appliquer les différentes couches de sécurisation une à une et de revenir aux étapes précédentes si besoin pour modifier et compléter le dispositif, ou de passé à une autre après avoir effectué et validé des tests.

2.2 Logiciels utilisés

Pour gérer le versionning de notre projet nous avons utilisé *Git* car c'est un des outils que l'on nous a présenté lors de nos cours et dont nous avions déjà connaissance. Pour le développement du programme du nœud, nous avons utilisé *Visual Studio Code* qui est un éditeur de texte puissant et possédant plusieurs extensions facilitant la programmation dont *Pycom* qui nous a permis de développer un premier nœud. Nous avons aussi utilisé *STM32CubeIDE 1.0.2* pour le développement

sur carte car il nous a permis d'écrire un programme simple pour implémenter des fonctions de sécurité. Pour le debugage et le tests des fonctions de sécurité nous avons utilisé *St-Link* et *St-Link Utility* qui permettent de lire et de voir les *bytes* d'option d'un microcontrôleur STM32, de plus *ST-Link Utility* est disponibles uniquement sur Windows que nous avons du utilisé œuvre via une machine virtuelle supervisé par le logiciel *VirtualBox*. Pour effectuer le reverse engineering des binaires extrait de la mémoire par le biais de *St-Link* nous avons utilisé le logiciel *Ghidra* car c'est un logiciel que les Master 2 ont utilisé et c'est le seul logiciel de reverse engineering que nous connaissions.

Pour mettre en place la *Box LoRa* (passerelle, network server et application server) nous utilisons l'OS *ChirpStack* car il permet d'utiliser et de de configurer facilement ces 3 services du LoRaWAN de plus il peut être utilisé sur la Raspberry Pi qui nous sert de support pour cette partie du projet.

Pour finir avons utilisé comme OS Linux et plus précisement les distributions *Manjaro* qui est basé sur *Arch Linux* et *Xubuntu* qui est basé sur *Ubuntu* qui est lui même basé sur *Debian*

2.3 Répartition des tâches

Pour ce projet nous sommes deux étudiants, dans la première partie du projet (premier mois) nous avons tout deux étudié le LoRaWAN car nous n'avions pas de connaissances sur ce sujet auparavant. Pour valider nos connaissances acquises durant cette étape de recherche nous avons mis en place un démonstrateur à présenter à notre deuxième jalon .

A la reprise du projet en décembre, nous avons chacun travaillé sur une partie différente du projet. François à travaillé sur la sécurisation de la *Box LoRa* et Arthur sur la sécurisation du nœud.

La Box LoRa est appareil permettant d'utiliser plusieurs services, un service LoRaWAN mais peut aussi contenir des services de mail par exemple. L'objectif de cette partie est d'empêcher un utilisateur d'avoir accès aux ressources d'un autre. Par exemple, le service mail ne doit pas interférer avec le service LoRaWAN. Pour cela il a fallu apporté des modifications à l'OS *ChirpStack*.

Le nœud LoRaWAN à besoins de clés pour ce connecter au réseau, qu'il doit dans notre cas stocker dans sa mémoire. Si un attaquant parvint à récupérer ces clés, il pourra espionner le trafic LoRaWAN ou connecter sont propre nœud à la place du nôtre. Pour sécurisé ces clés il a fallu utiliser des solutions de sécurité déjà présentes dans le microcontrôleur ou utiliser un composant de sécurité pour stocker les clés.

Chapitre 3

Réalisation et Tests

Au cours de ce projet nous devions effectué 3 réalisations, un démonstrateur, un nœud sécurisé et une *Box LoRa* sécurisé.

3.1 Démonstrateur

Comme nous l'avons dis dans le chapitre précédant nous avons dans un premier temps réalisé un démonstrateur non sécurisé d'un réseau LoRaWAN. Le but était d'utiliser nos connaissances du LoRaWAN et de les démontrer lors d'une présentation. Pour créer ce démonstrateur nous avons utiliser l'OS *ChirpStack*[1] sur la *Raspberry Pi* équipé de son *shield* antenne *iC880A* pour le nœud nous avons utilisé la carte *Fipy*[3] avec sont *shield PySence*.

L'OS ChirpStack permet de mettre très facilement un réseau LoRaWAN en fonctionnement car il intègre déjà les différents services de passerelle, serveur réseau et serveur d'application. Ces différents service sont aussi facilement configurable grâce à une interface web.

La carte *Fipy*, est une carte de développement orienté pour l'utilisation de réseau sans fils comme WiFi, Bluetooth ou LoRaWAN. Cette carte ce programme en *Python* avec l'extension *Pymakr* pour les éditeur de text *Atom* et *Visual Studio Code*. Elle nous a permis de créer le noeud LoRaWAN facilement car la documentation donne des exemples pour ce type de programme.

Lors du développement de ce démonstrateur, nous n'avons pas eu de problèmes particulier autre que des problèmes de configuration de l'extension *Pymakr*.

En fonctionnement le nœud envoyait une chaine de caractère à la passerelle puis on lisait cette chaine caractère grâce au serveur d'application.

3.2 Nœud

Pour le rendu final du projet le nœud doit fonctionner sur une carte B-L072Z-LRWAN1[4] qui est une carte de découverte produite par STmicroelectronics orienté pour le développement de solutions basées sur des réseaux *LoRaWAN* ou *SigFox*. Cette carte est équipé d'une MCU *STM32L072CZ* qui est basé sur une architecture Arm Cortex M0+.

Pour rappelle, dans l'architecture LoRaWAN, le rôle du nœud est de transmettre une information qui serra traité par l'application server. Dans le cadre de notre scénario le campus connecté, nous transmettons une donnée publique qui est la température. Nous nous sommes donc concentré, sur l'authentification du nœud LoRaWAN et l'intégrité de la donnée.

Pour cela nous devions éviter que notre nœud soit remplacé par un autre nœud lequel pourrait envoyer des informations erronées.

3.2.1 Surface d'attaque du nœud

Sur la figure 3.1 vous pouvez voir les différents composant autour de notre programme.

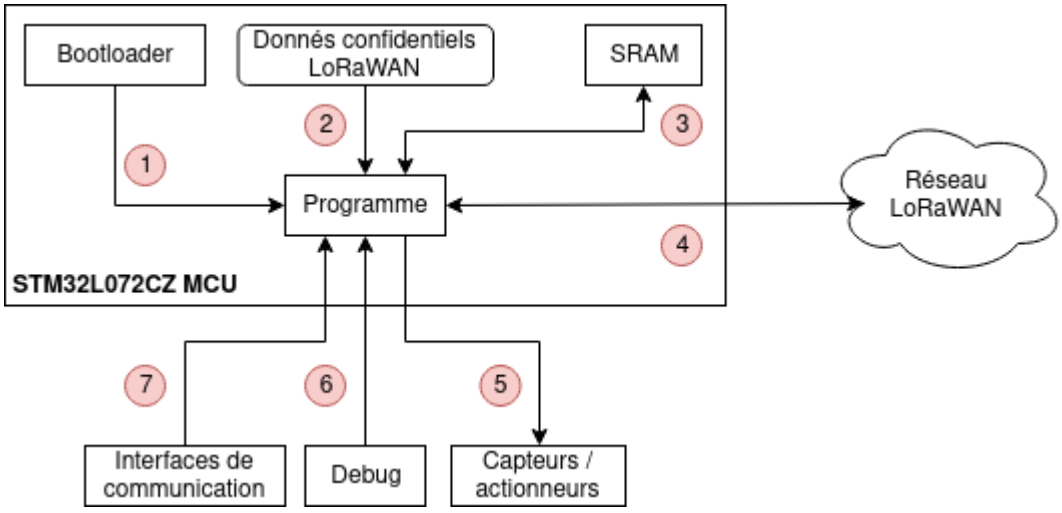

FIGURE 3.1 – Schéma présentant les différents éléments autour de la carte B-L072Z-LRWAN1

Il y a plusieurs vecteur d'attaque possible dans nôtre cas, chaque points de la figure 3.1 est un cible potentielle :

1. Changer le mode de démarrage pour utiliser un autre programme que celui contenu dans la mémoire Flash et avoir accès à tout les contenu du micro

contrôleur.

2. Accès à des parties du programme sensibles pour les copier ou les voler
3. Utilisation de faille dans la SRAM comme un buffer overflow pour voler des information de dans ou provoquer un dénie de service sur la carte.
4. L'observation des communications et/ou usurpation des appareils du réseau
5. Un attaquant pourrait remplacer le *shield* contenant les capteurs pour envoyer des fausses valeurs et mettre dans un état non déterminé.
6. Utilisation des ports de debugages pour avoir un accès complet au micro contrôleur.
7. Utilisation des Interfaces de communication pour avoirs accès au micro contrôleur

Pour chacune des attaques ci-dessus on peut trouver une ou plusieurs contremesures :

1. Autoriser qu'un seul mode de démarrage. Désactiver le démarrage depuis le port de débugage
2. Mettre la mémoire en mode execution only. Utilisé une MPU(Memory protection unit). Créer des zones mémoires sécurisé
3. Utilisé une MPU(Memory protection unit). Créer des zones mémoires sécurisé
4. Chiffrer et signer les communications.
5. Utilisation d'un système pouvant détecter une intrusion au niveau de la carte.
6. Désactiver les fonctionnalités de débugage
7. Rendre les interfaces de communications difficiles d'accès. Desactivé les interfaces si on ne les utilises pas.

Pour ce projet nous nous sommes concentré sur deux types d'attaques, la première **Une lecture de la mémoire (Dump mémoire)** et la deuxième **Accès à la mémoire par une partie du programme non autorisé**

Nous avons commencé par faire des recherches sur quel méthode de connexion était la plus intéressante à sécuriser par rapport à nôtres scénario.

Le protocole LoRaWAN permet à un nœud de ce connecter de deux façons différentes, **OTAA** (Over The Air Activation), **ABP** (Activation by personalization).

Fonctionnement activation OTAA

L'activation par **OTAA** met en jeux *DevEUI*, *AppEUI*, *AppKey* pour ce connecter à la *gateway* puis utilise *AppSKey* et *NetSKey* pour chiffrés la communication jusqu'à l'*application server*. Le protocole LoRaWAN ne précise pas si les clés doivent être stocké en clair dans la mémoire ou doivent être chiffrés. Il faut *AppSKey* et *NetSKey* sont renouvelé à chaque Si un attaquant arrive à obtenir les trois premières clés alors, il pourra usurpé l'identité de notre nœud. S'il parvint à obtenir les deux autres clés, il pourra uniquement déchiffrer l'information.

Fonctionnement activation ABP

L'activation par personnalisation met en œuvre *DevAddr*, *NwkSKey* et *AppSKey* ces deux dernière sont les clés permettant de chiffrer la communication ainsi, si un attaquant les récupères, il peut tout aussi bien usurper l'identité de notre nœud que lire les informations envoyés.

Analyse des risques, pour la connexion d'un nœud à un réseau LoRaWAN

Dans un premier temps on remarque que le protocole LoRaWAN ne stipule pas si les clés doivent être stocké en clair dans la mémoire ou bien chiffré. Cela veut dire que si un attaquant fait un *dump* de la mémoire, il va pouvoir trouver facilement les clés. On remarque ensuite, que l'utilisation de l'activation par **OTAA** est plus sécurisé car elle utilise plus de clés et en renouvelle 2 à chaque connexions. La connexions par **ABP** est moins sécurisé car elle utilise uniquement trois clés, dont deux qui peuvent aussi permettre l'écoutes de la communication.

Dans le scénario que nous utilisons nous n'avons pas besoins de dissimuler l'information transmise, donc obtenir *NetSKey* et *AppSKey* lors d'une activation **OTAA** nous importe peut.

D'un point de vue technique le nombre de clés à récupérer par attaquant pour usurper l'identité de notre nœud est le même d'un mode d'activation à l'autre. Nous avons choisit d'utiliser et de protéger le mode d'activation **ABP**, car si un attaquant arrive à obtenir les clés il pourra lire les informations transmises et/ou usurpé l'identité de nôtres nœud, de plus nous trouvons intéressant de pouvoir observer les trames LoRaWAN ce propager dans l'air à l'aide d'un analyseur de spectre et de pouvoir les décodé avec un logiciel comme *GNURadio*.

3.2.2 Nœud LoRaWAN avec une carte B-L072Z-LRWAN1

Le premier objectif que nous nous sommes fixé était de créer un nœud LoRaWAN non sécurisé comme l'on avait fait avec le démonstrateur.

Pour cela nous avons trouvé le projet *LoRaMac-node*[2] sur *Github*, qui est un projet permettant de configurer facilement un nœud LoRaWAN avec des cartes de développement dont la carte B-L072Z-LRWAN1 que nous utilisons. De plus dans sa documentation il est indiqué comment utiliser un composant de sécurité.

Le projet *LoRaMac-node* permet d'utiliser les trois différentes class de nœud LoRaWAN et avec plusieurs versions du protocole 1.0 à 1.0.3.

Pour fonctionner sur différentes plateformes le projet utilise des fonctions utilisant d'autres fonctions propre aux cartes sur lesquels il peut être déployer. Pour compiler un programme sur les bonnes cartes il utilise, **Cmake** pour générer des *makefile* lesquels permettent ensuite d'inclure les *lirairie* correspondant à la cible.

3.2.3 Sécurisation carte STM32

Comme nous l'avons dit précédemment, le nœud doit pouvoir stocker de manière sécuriser des clés. Pour cela nous avons deux solutions, stocker les clés dans un élément sécurisé ou les stocker dans la mémoire du micro contrôleur de manière sécurisé.

Nous avons choisit dans un premier temps de stocker les clés dans la mémoire, afin d'éviter d'utiliser un élément de sécurisé qui pourrait augmenter le coûts de notre projet, et sa consommation d'énergie. Cependant si les mesures de sécurité offertes par cette première solution ne sont pas suffisantes nous pouvions dans un second temps utiliser un élément de sécurité.

Fonctions de sécurité dans les MCU STM32L0

Dans un premier temps de recherche nous avons trouvé que les MCU STM32L0 contenait déjà des contres mesures pour différents types d'attaques, prêtes a l'emploi à l'aide de fonctions *HAL*[1]

Vous pouvez trouver la liste des protection présentes dans notre MCU ci-dessous :

- **RDP** (Read Out Protection) : Protection de la mémoire Flash qui empêche la copie du code. Cette fonction prévient donc du reverse engineering fait à l'aide des outils de debugage. Empêche aussi de charger une nouveau programme sur la carte.
- **WRP** (Write Protection) : Sert à protéger une partie de la mémoire Flash d'un effacement ou d'une mise à jour.
- **PCROP** (Proprietary code read-out protection) : Permet de configurer certaines partie de la mémoire flash pour qu'elles soient uniquement accéssible par le bus d'instruction du CPU (Execution Only).

1. Hardware Abstraction Layer

— **Firewall** : Le par feu est un composant physique qui contrôle les accès à trois parties de la mémoire, la zone du code (mémoire flash), les donnés volatile (SRAM) et les donnés non volatile (Flash).
— **MPU** (Memory Protection Unit) : Mécanisme de protection qui permet de définir des droits d'accès à certaines zone de la mémoire.
— **Anti temper** : Détection d'une intrusion au niveau physique, permet de prendre les mesure adéquates comme effacer la mémoire par example.
— **IWDG** (Independant watchdog) : Watchdog Independant permettant de lever des *flags* si une tache prend plus de temps que celui qui lui est attribué.

Pour rappel les deux attaques contre les quelles nous allons protéger la carte sont : **Une lecture de la mémoire (Dump mémoire)** et **Accès à la mémoire par une partie du programme non autorisé**. Pour protéger le nœud de la première nous avons choisit de mettre d'utiliser les sécurité **RDP** et **PCROP**, pour la deuxième **PCROP** et **Firewall**.

Protection RDP La RDP offre différents niveaux de protection 0, 1, 2.
— **Le niveau 0**, équivaut à aucune protection la mémoire flash et complètement lisible, peut importe le mode de démarrage du contrôleur (par la RAM, par la flash, par le debuger ...). Ce mode doit être utilisé uniquement pour la phase de développement.
— **Le niveau 1**, empêche l'accès à la mémoire flash et à la mémoire SRAM2 par le debuger. Cependant lorsque le programme démarra à partir de la mémoire flash celui-ci à accès à la mémoire flash et la SRAM2.
— **Le niveau 2**, ce niveau empêche tout les accès aux mémoire depuis l'extérieur. **Attention après l'avoir activé on ne peut plus revenir en arrière**.

Il faut faire très attention lors de l'utilisation de la **RDP** car dés que l'on utilise une protection de niveau supérieure à 1, on ne peut plus mettre de programme dans la carte via le bootloader. Pour mettre à jour le programme il faut utiliser une *SFU*[2].

Protection PCROP La **PCROP** permet de sécuriser la mémoires par secteur[3]. Il sert principalement à protéger les propriétés intellectuelles d'un programme, mais il peut aussi, permettre de dissimuler des code sensible dans une mémoire comme un code permettant de générer des clés cryptographiques. Lorsque

2. Secure Firmware Update
3. La taille des secteur dépend du micro contrôleur, dans nôtres cas, 1 secteur = 32 pages = 4Ko

un secteur est sécurisé, son accès est uniquement possible via le bus de d'instruction. Si un secteur protégé est lu il retournera une erreur sur le bus de lecture.

Firewall Comme le **PCROP**, le **Firewall** protège des secteur de la mémoire définit à l'initialisation du programme. Le **Firewall** est un composant physique qui va contrôler les et filtrer les accès entre 3 parties : La zone du Code dans la mémoire Flash, une zone de mémoire volatile dans SRAM et une zone de mémoire non volatile dans la Flash. L'accès à un code sécurisé par un **Firewall** ce fais par une seule fonction, si une autre fonction essaie d'accéder à ce code, le **Firewall** générera un *reset* de la carte

Mass Erase Avant d'implanter ces contres mesures dans un programme nous avons chercher comment es désactiver pour éviter d'être bloqué avec une carte non utilisable et ainsi pouvoir effectuer plusieurs tests. Pour pour pouvoir passer du niveau **RDP** 1 à 0 et pour désactiver **PCROP**, il faut effectuer un *mass erase* sur la carte ce qui à pour conséquence de mettre toute les mémoire Flash et SRAM à 0. Pour effectuer une mass erase il y a deux possibilité, la première et d'écrire une fonction faisant un *mass erase* dans la RAM et la deuxième est d'utiliser l'utilitaire *ST-link utility* qui permet de faire en autre ce genre d'opération sans avoir à les programmer.

Avant d'implémenter les protections, nous avons réaliser des dumps de la mémoire lorsque un programme fonctionnait sur la carte pour vérifier que nous trouvions bien les clés écrites dans le code.

Pour cela nous effectuons un dump de la mémoire à l'aide du logiciel *St-Link* et nous ouvrions le fichier récupéré avec *Ghidra* qui nous permettait de revenir presque au code C de base. Vous pouvez voir en figure 3.2 une des clés présente dans la mémoire.

Pour être sur que nous trouvions bien la clés et pas des variables aléatoire formant la même suite nous avons essayé avec trois clés différente et nous cherchions à chaque fois les 3 clés, pour être sur que celle trouvé ne soit pas une coïncidence.

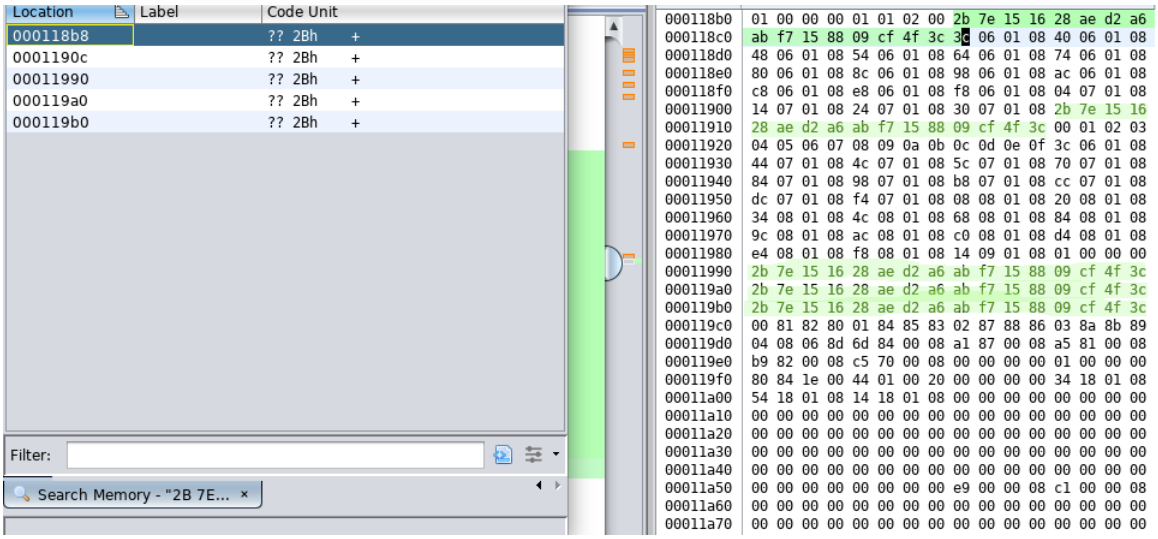

FIGURE 3.2 – Capture d'écran de *ghidra* sur laquelle on voit une des clés dans la mémoire

3.2.4 Problèmes rencontré

A ce jour, nous n'avons pas réussi à implémenter les contres mesures de sécurité présenté ci-dessus dans le programme LoRaMAC-node car, pour implémenter ces contres mesures il faut utiliser des fonctions *HAL* propres au STM32 qui sont définit dans des librairies différentes d'un micro contrôleur à l'autre.

Ces librairies ce trouves dans le projet *LoRaMAC-node*, mais lorsque nous utilisons les fonctions *HAL* pour par exemple activer le niveau 1 de **RDP** le compilateur ne parvient pas à faire le lien entre l'appelle des fonctions et leur définitions dans les *librairie*.

Hypothèse 1

Les librairies utilisées pour définir les fonctions ne sont pas incluses dans le programme.

Pour vérifier si les fonctions sont bien définit, nous avons essayé de lister tout les *include* qu'appelle chaque librairie du programme. Pour vérifier si le programme finit bien par appeler la librairie nécessaire à nos fonctions.

Au pour finir nous n'avons pas réussi à trouver avec certitude quelle librairies étaient utilisés.

Pour palier à ce problème nous avons décidés de programmer les différentes contres mesures dans un programme simple dans lequel nous avons dissimulés une

variable contenant des clés. Pour d'abords être sur que les contremesure fonctionne et que le code que nous implémenterons dans le programme *LoRaMAC-node* soit validé.

3.2.5 Tests Unitaire

Tests RDP

Pour tester le fonctionnement de la contre mesure **RDP** nous l'avons implanté dans notre programme simple. Puis nous avons essayés de lire la mémoire avec *St-Link Utility* s'il nous affichait un message d'erreur disant de désactiver *Read Out Protection* nous allions verifier les Octet d'option du programme et regardions le niveau de la *RDP*. Comme vous pouvez le voir sur la figure 3.3

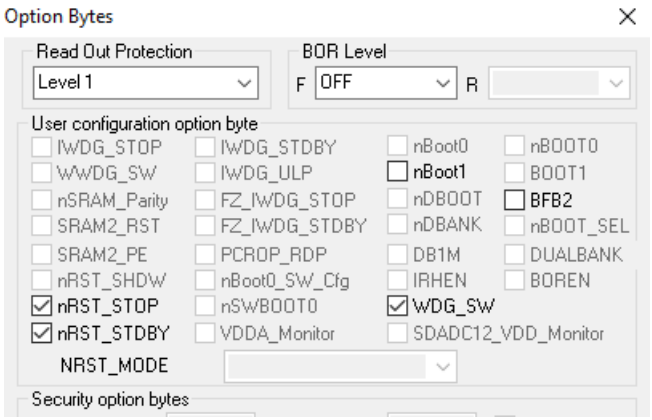

FIGURE 3.3 – Capture d'écran de *St-link utility* sur laquelle on voit le niveau RDP de la carte

Tests PCROP

Pour vérifier si *PCROP* a bien été implémenté il faut brancher la carte au programme *St-link Utility* et afficher les octets d'option pour voir les secteur sécuriser comme on peut le voir sur la figure 3.4

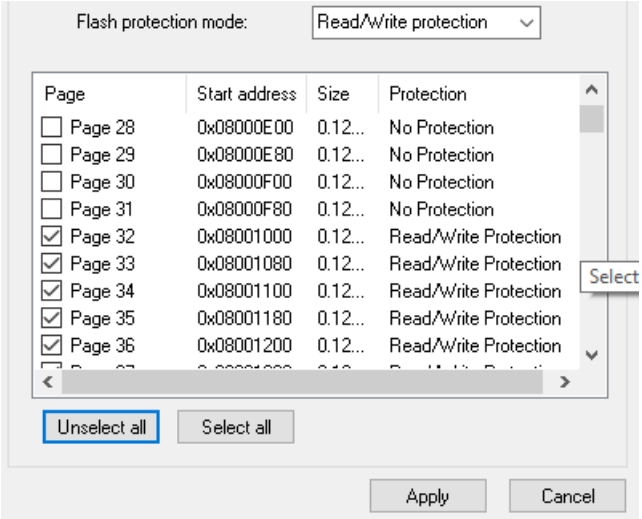

FIGURE 3.4 – Capture d'écran du logiciel *St-Link Utility* sur laquelle on peut voir Des secteur sécurisé et non sécurisé

3.2.6 Tests Intégration

3.2.7 Conclusion

Pour la suite du projet il nous restera à trouver comment stocker une fonction à un endroit précis de la mémoire pour pouvoir utiliser *PCROP* car actuellement nous savons sécuriser les secteurs souhaités mais sans savoir où se trouve dans la mémoire la fonction à sécuriser.

Il nous restera aussi à utiliser le *Firewall* si besoins car il faudra faire une étude de besoins pour vérifier si *PCROP* n'est pas suffisant dans nôtres cas.

3.3 Passerelle

3.3.1 Tests Unitaire

3.3.2 Tests Intégration

3.3.3 Problèmes rencontré

3.3.4 Conclusion

Chapitre 4

Conclusion

Table des figures

3.1 Schéma présentant les différents éléments autour de la carte B-L072Z-LRWAN1 7
3.2 Capture d'écran de *ghidra* sur laquelle on voit une des clés dans la mémoire 13
3.3 Capture d'écran de *St-link utility* sur laquelle on voit le niveau RDP de la carte 14
3.4 Capture d'écran du logiciel *St-Link Utility* sur laquelle on peut voir Des secteur sécurisé et non sécurisé 15

Annexe A

Première Annexe

Bibliographie

[1] Orne Brocaar. Chirpstack. https://www.chirpstack.io/.

[2] Lora net / Semtech. Loramac-node. https://github.com/Lora-net/LoRaMac-node.

[3] Pycom. Fipy. https://pycom.io/product/fipy/.

[4] STMicroelectronics. B-l072z-lrwan1. https://www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-mpu-eval-tools/stm32-mcu-mpu-eval-tools/stm32-discovery-kits/b-l072z-lrwan1.html.