Sécurité et détection d'intrusion dans les réseaux sans fil

Mémoire pour l'obtention du

Doctorat de l'Université de Lille
Discipline : Informatique

par

Étienne Helluy-Lafont

présenté le
29 janvier 2021

Composition du jury :

Vincent Nicomette	*Professeur - INSA Toulouse*	Rapporteur
Valérie Viet Triem Tong	*Professeur - Centrale Supelec*	Rapportrice
Hervé Debar	*Professeur - Télécom SudParis*	Examinateur
Alexandre Boé	*Maître de conférence - Université de Lille*	Encadrant
Gilles Grimaud	*Professeur - Université de Lille*	Encadrant
Michaël Hauspie	*Maître de conférence - Université de Lille*	Directeur de thèse

Numéro d'ordre : 117369

Table des matières

1 Introduction **9**

2 Contexte et état de l'art **11**
2.1 Standards et équipements sans fil 12
2.1.1 Bluetooth Classic 12
2.1.2 Architecture d'un équipement Bluetooth 14
2.2 Détection d'intrusion 15
2.2.1 Détection de signature 16
2.2.2 Détection d'anomalie 16
2.2.3 Sondes pour l'IoT 17
2.3 Radio Logicielle 18
2.3.1 Généralités 18
2.3.2 Applications à la détection d'intrusion 19
2.4 Sécurité des *baseband* 20
2.4.1 Recherche de vulnérabilités 21
2.4.2 Instrumentation 22
2.5 Implémentations ouvertes de *baseband* 24
2.5.1 GSM 24
2.5.2 Bluetooth Low Energy 26
2.6 Conclusion 26

3 Problématique **27**
3.1 Des protocoles peu sécurisés 27
3.1.1 Exemple du GSM 27
3.1.2 Exemple du Bluetooth 29
3.2 L'opacité des modems 31
3.2.1 La sécurité par l'obstacle matériel 31
3.2.2 Du matériel très soft 32
3.2.3 Nous dépendons de code fermé et vulnérable 33
3.3 L'angle mort des communications réseau 34
3.3.1 Une perte de contrôle sur le code réseau du système ? 34
3.3.2 Les HIDS : une vision partielle 34
3.3.3 Les WIDS c'est pour le Wi-Fi. 36

3.4 Pistes étudiées . . . 37
3.4.1 Opportunités de sécurisation . . . 37
3.4.2 Prendre le problème par en-dessous : IDS sur couche physique . . . 39
3.4.3 Les micrologiciel . . . 42
3.5 Conclusion . . . 45

4 Sonde SDR pour la détection d'intrusion 47
4.1 Motivations . . . 48
4.1.1 *Software Defined Radio* . . . 49
4.1.2 Travaux reliés . . . 49
4.1.3 Approche générale . . . 51
4.1.4 Objectifs détaillés . . . 52
4.2 Architecture proposée . . . 54
4.2.1 Description générale . . . 55
4.2.2 Chaine de traitement. . . . 56
4.3 Implémentation : *Phyltre* . . . 60
4.3.1 Format du projet . . . 60
4.3.2 Détection d'activité des canaux . . . 62
4.3.3 Adaptation des canaux . . . 66
4.3.4 Détection d'énergie dans le domaine temporel. . . . 67
4.3.5 Démodulation . . . 68
4.4 Conclusion . . . 70

5 Fingerprinting sur la couche physique du Bluetooth Classic 71
5.1 Travaux reliés . . . 72
5.2 Expérience sur l'*inquiry* Bluetooth Classic . . . 73
5.2.1 Cadre expérimental . . . 73
5.2.2 Traits selectionnés . . . 74
5.3 Évaluation . . . 78
5.3.1 Observation des caractéristiques . . . 78
5.3.2 Effet de la température . . . 82
5.3.3 Analyse. . . . 83
5.4 Conclusion . . . 84

6 Attaques de contrôleurs Bluetooth 87
6.1 Introduction . . . 87
6.2 Travaux antérieurs . . . 88
6.3 Vulnérabilités des contrôleurs Bluetooth Classic . . . 89
6.3.1 Modèle d'attaque . . . 89
6.3.2 Choix de la couche LMP . . . 91
6.4 Méthodologie . . . 92
6.4.1 Première phase de rétro-ingénierie . . . 92
6.4.2 Identification du code pertinent . . . 92

6.4.3 Recherche de vulnérabilités dans la couche LMP 94
6.4.4 Développement de l'attaque 94
6.5 Cas pratiques 95
6.5.1 Première phase de rétro-ingénierie 95
6.5.2 Recherche de vulnérabilité 98
6.5.3 Développement d'une attaque 101
6.6 Analyse 105
6.6.1 Sources de vulnérabilités 106
6.6.2 Implications sur pour la sécurité 108
6.6.3 Contremesures 109
6.7 Conclusion 110

7 Une couche *baseband* Bluetooth Classic Open source 113
7.1 *Baseband* Bluetooth Classic 113
7.1.1 Paquets 114
7.1.2 Canaux physiques 116
7.1.3 Transports et canaux logiques 117
7.2 Implémentation d'UBTBR 119
7.2.1 Matériel 119
7.2.2 Architecture logicielle 119
7.2.3 Interface de contrôle 123
7.2.4 Code hôte 124
7.2.5 Approche du développement 124
7.2.6 Résultats 125
7.3 Cas d'usage : Surveillance d'une connexion 126
7.3.1 Contexte 126
7.3.2 Procédure moniteur 128
7.3.3 Évaluation 129
7.4 Conclusion 132

8 Conclusion 133
8.1 Synthèse 133
8.2 Limites et travaux futurs 135
8.2.1 Outillage 135
8.2.2 *Fingerprinting* 135
8.2.3 Détection d'intrusion 136

Remerciements

Je tiens d'abord à remercier Valérie Viet Triem Tong et Vincent Nicomette pour avoir accepté d'être les rapporteurs cette thèse. Vos relectures attentives et vos remarques m'ont permis d'améliorer ce manuscrit. Merci pour vos rapports que je conserve précieusement.

Le doctorat a été une opportunité unique de travailler sur des sujets qui me passionnent, avec une liberté à laquelle peu ont la chance de goûter.

Je remercie mes encadrants Michaël, Gilles et Alexandre. Vous m'avez permis de faire cette thèse, soutenu et fait confiance tout au long de cette thèse. Je vous doit énormément et j'espère avoir un jour l'occasion de vous en rendre un peu dans le futur.

Je n'oublie pas non plus les autres doctorants de l'équipe. Merci Narjes, Quentin, François, Nadir, Mahiedine pour m'avoir décidé à me lancer moi aussi. Merci Florian, partager un bureau avec toi m'a bien aidé à traverser ce confinement (et désolé pour les bavardages incontrôlables). Merci aux autres membres de l'équipe, permanents ou non, j'ai adoré refaire le monde avec vous à la pause du midi.

Enfin, j'adresse un remerciement spécial à ma famille, à mes parents qui m'ont soutenu jusqu'au bout de ces longues études, et à mes frères et soeurs qui n'ont jamais cessé de croire en moi.

J'ai sûrement oublié beaucoup de noms. Tant pis je vous devrais une bière dès que les terrasses rouvriront, bientôt j'espère. On va enfin pouvoir fêter dignement l'achèvement de cette thèse !

Chapitre 1

Introduction

Cette thèse a été financée par l'Université de Lille et l'équipe 2XS du laboratoire CRIStAL, avec le soutien de l'IRCICA [1] qui est une unité de service et de recherche du CNRS. L'IRCICA permet la collaboration d'équipes de différents domaines scientifiques afin de mener des projets interdisciplinaires. Ainsi, j'ai été encadré par des membres de l'équipe 2XS, mais aussi un membre de l'équipe CSAM du laboratoire IEMN [2], spécialisée dans les communications sans fil à basse consommation.

L'équipe 2XS (pour eXtra Small eXtra Safe) est spécialisée les systèmes embarqués et la sécurité informatique. Une partie de l'équipe 2XS travaille sur *pipcore* [Jom18], un noyau de système d'exploitation avec des fonctionnalités d'isolation mémoire formellement prouvées. L'équipe étudie également les techniques de détection d'intrusion. Damien Riquet a conçu Discus [Riq15], un système de détection d'intrusion (IDS) distribué destiné à sécuriser les centres de données. Une seconde thèse actuellement en cours porte sur la détection d'attaques DDoS, également dans les centres de données. Enfin, deux thèses dont celle-ci portent sur la sécurité des communications sans fil des objets connectés.

Les communications sans fil ont toujours posé des enjeux de sécurité spécifiques. D'abord car elles sont par nature directement exposées à des attaques distantes, pouvant mettre à mal leur confidentialité, leur disponibilité, mais aussi leur intégrité. Aujourd'hui, les interfaces sans fil représentent toujours des portes d'entrée intéressantes pour des intrusions, qui restent mal protégées au regard de leur exposition. Il est donc important d'étudier des solutions pour mieux assurer leur sécurité.

Avec le développement rapide des objets connectés, de nouvelles contraintes apparaissent. Les objets connectés s'appuient sur une multitude de protocoles de communication sans fil plus ou moins standardisés. Cela représente un défi pour les solutions de détection d'intrusion basées sur le réseau, qui font désormais face à des environnements radio hétérogènes. En outre, il est difficilement envisageable d'intégrer des IDS directement dans les objets connectés, qui utilisent des architectures propriétaires et souvent contraintes, qui se prêtent mal à l'ajout de logiciels extérieurs. Pour assurer la sécurité des communications

1. IRCICA : Institut de Recherche sur les Composants logiciels et matériels pour l'Information et la Communication Avancée (USR-3380)

2. Institut d'Électronique de Microélectronique et de Nanotechnologie (UMR 8520)

sans fil des objets connectés, les méthodes de détection d'intrusion réseau doivent être adaptées.

Enfin, les *modems* qui réalisent les communications sans fil peuvent eux-même comporter des vulnérabilités. Ils peuvent alors être ciblés par des attaques qui interviennent sur des couches de communication qui ne sont pas surveillées par les IDS. Il est donc important de bien comprendre ce type de menaces, afin de faire évoluer les IDS pour en tenir compte.

Cette thèse explore le sujet de la sécurité des réseaux sans fil, en s'intéressant en particulier aux problématiques survenant au niveau des couches basses des communications. Cela inclut l'analyse des transmissions au niveau de la couche physique, l'étude de la sécurité de modems, et pour finir l'implémentation d'une couche physique Bluetooth Classic Open Source.

Mon travail s'est focalisé sur le protocole Bluetooth Classic, car il est l'un des plus utilisés dans les objets connectés. J'ai ainsi mené de nombreuses expériences sur les différents aspects de ce protocole. Néanmoins, ces travaux peuvent être généralisés au domaine des communications sans fil liées aux objets connectés.

Ce document est organisé comme suit. Le chapitre 2 présente l'état de l'art et les principaux concepts liés à cette thèse. La chapitre 3 détaille notre problématique, et motive nos contributions. La chapitre 4 présente l'architecture et l'implémentation de *Phyltre*, une radio-logicielle qui a servi de base à nos expériences. La chapitre 5 propose des méthodes de fingerprinting sur la couche physique Bluetooth Classic. La chapitre 6 décrit une étude des vulnérabilités des modems Bluetooth. La chapitre 7 présente UBTBR, un *baseband* Bluetooth Classic Open Source pour faciliter la recherche sur ce protocole.

Chapitre 2

Contexte et état de l'art

Cette thèse porte sur la sécurité des communications sans fil. Plus spécifiquement, elle s'articule sur les problématiques de sécurité liés aux composants qui servent à réaliser les communications sans fil (*modems*).

En effet, les *modems* posent des défis supplémentaires aux techniques habituelles de sécurisation des échanges réseau. En particulier, certaines couches des protocoles sans-fil sont entièrement implémentées dans les *modems*, et ne sont pas exposées à l'hôte du modem.

Ces couches protocolaires enfouies dans les modems constituent un obstacle important pour les systèmes de détection d'intrusions (IDS), qui peuvent notamment chercher à détecter des attaques intervenant par le réseau. En effet, pour détecter des intrusions sur des échanges réseau, il faut d'abord avoir accès au contenu de ces échanges. Par ailleurs, trop peu de recherches sont menées sur la sécurité des modems sans fil, à cause de leur opacité, mais aussi du manque d'outils permettant d'interagir avec les couches basses des protocoles sans fil.

Plusieurs approches peuvent cependant être envisagées pour s'assurer de la sécurité de modems opaques. Cette thèse s'intéresse à deux de ces approches.

La première approche que nous explorons dans cette thèse est la détection d'intrusion sans fil et plus précisément les techniques d'IDS qui peuvent être utiles pour protéger des modems fermés. Dans ce cadre, nous avons développé un outil pour capturer rapidement les paquets de divers protocoles sans fil par *radio-logicielle*. Nous l'avons ensuite utilisé pour étudier des méthodes de *fingerprinting* sur la couche physique, qui peuvent améliorer la capacité des WIDS à détecter l'usurpation de l'identité d'un transmetteur légitime.

La seconde approche est d'identifier des vulnérabilités dans des modems sans fil, pour pouvoir les corriger, mais aussi pour mieux comprendre comment les IDS doivent faire face à ces menaces. Nous avons concentré nos recherches sur les couches *baseband* du Bluetooth Classic. Enfin, pour faciliter les futures recherches sur les vulnérabilités des composants Bluetooth Classic, nous avons développé une implémentation open-source d'un contrôleur Bluetooth, qui permet d'explorer une grande partie de la surface d'attaque des couches basses de ce protocole.

2.1 Standards et équipements sans fil

Avec le développement de la téléphonie mobile, nous utilisons désormais quotidiennement les communications sans fil, partout où nous allons. D'autres normes de communication sans fil ont ensuite été adoptées, comme la Wi-Fi ou le Bluetooth. Avec l'émergence de l'Internet des Objets (IdO), de plus en plus d'objets communiquent via tout un éventail de standards sans fil.

La standardisation des protocoles de communication permet à des appareils de constructeurs différents d'être compatibles les uns avec les autres. Aujourd'hui, les communications sans fil reposent beaucoup sur des composants intégrés (*modems*), qui fournissent le support clé en main d'une ou plusieurs normes de communication.

Ces modems présentent de nombreux avantages : ils sont performants, leur consommation d'énergie est maîtrisée, et ils peuvent être rapidement intégrés dans de nouveaux produits.

Mais comme tout équipement de télécommunication, ces modems font face à des problématiques de sécurité. Des intrus peuvent chercher à détourner leurs communications, ou à les compromettre pour obtenir un accès non autorisé dans un système d'information.

Cette section introduit le sujet des normes sans fil et des modems qui permettent de les utiliser. Pour cela la première partie présente l'exemple de la pile Bluetooth sur laquelle se sont basés la plupart de nos travaux. La seconde partie présente des exemples d'intégration du Bluetooth dans des objets communicants.

2.1.1 Bluetooth Classic

Bluetooth est une norme de communication sans fil, dont la première version fut publiée en 1999. Elle offre une courte portée allant de quelques mètres à une centaine de mètres selon les classes d'appareils.

Bluetooth a rapidement été intégré aux téléphones mobiles et aux ordinateurs, et est très utilisé pour y connecter sans fil des accessoires tels des claviers et souris, périphériques audio, etc.

Bluetooth partage la bande de fréquence ISM 2,4 GHz, également utilisée par la Wi-Fi. Sa première version (Bluetooth 1.0) ne définissait qu'un seul type de modulation : *Gaussian Frequency Shift Keying* (GFSK) avec un débit d'un mégabit par seconde (Mbps). La norme a ensuite évolué et offre désormais deux nouveaux types de modulation pour fournir des débits allant jusqu'à 3 Mbps. Avec le Bluetooth 3.0 + *High Speed*, les données peuvent désormais utiliser la couche physique Wi-Fi pour permettre un débit de 24 Mbps. En outre, la norme Bluetooth s'est enrichie d'une version *Low Energy*, avec une bande passante réduite pour une consommation plus faible, qui est mieux adaptée pour les objets connectés.

La norme Bluetooth comporte aujourd'hui de nombreux protocoles. La figure 2.1 présente certains d'entre eux, dans une version simplifiée de la pile Bluetooth Classic. Elle se compose d'une une partie **contrôleur** et d'une partie **hôte**. Le contrôleur est un composant dédié (qu'on appele généralement la puce Bluetooth), et l'hôte correspond au système

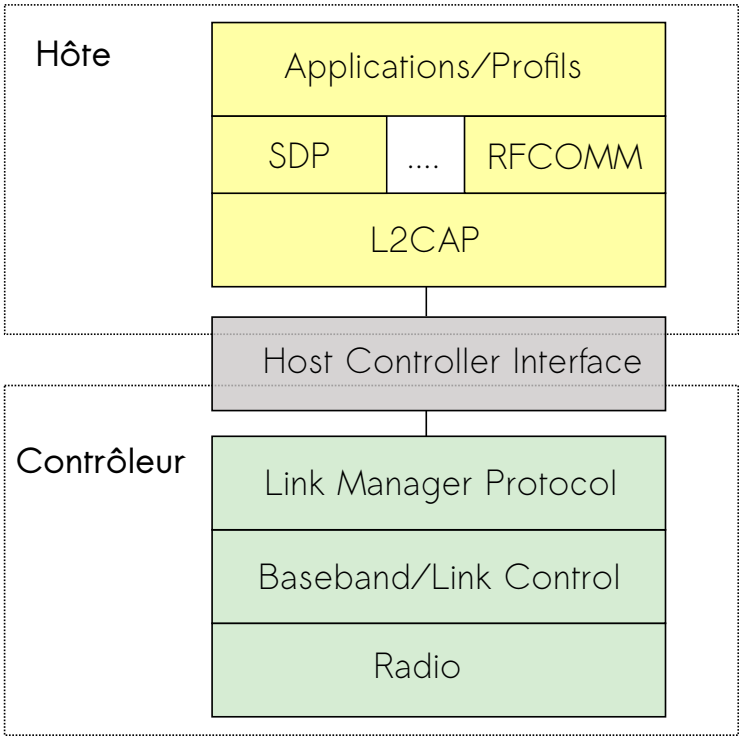


FIGURE 2.1: Vue d'ensemble de la pile Bluetooth Classic

d'exploitation, par exemple Linux. Les deux éléments sont interfacés via le protocole *Host Controller Interface* (HCI) qui transite le plus souvent sur un lien série ou USB.

Observons cette pile de bas en haut. La couche **radio** permet l'accès au canal physique sans fil. Elle effectue notamment la modulation et la démodulation des signaux radio. La couche **baseband** organise le fonctionnement des canaux physiques. Elle gère les différents états du contrôleur, l'encodage et le décodage des paquets, et intègre le contrôle du lien (*Link Control* (LC)) pour fournir des canaux de communication fiables. La couche **Link Manager Protocol** (LMP) est utilisée pour la négociation et la gestion des connexions, et le contrôle des canaux physiques. Elle réalise notamment les procédures d'appairage, ainsi que l'allocation des ressources.

La couche **HCI** est une interface standardisée pour connecter l'hôte et le contrôleur. Elle masque une partie de la complexité du contrôleur au système d'exploitation, et assure l'interopérabilité des différentes implémentations de piles hôte et contrôleur.

Le **L2CAP** (*Logical Link Control and Adaptation Protocol*) permet le multiplexage des transports logiques. Elle permet à de multiples applications d'accéder à différents canaux sur un même tranport logique. Elle permet également la fragmentation et le réassemblage des paquets échangés sur ces canaux.

Plusieurs **services** sont ensuite définis. Par exemple, le **SDP** (*Service Discovery Protocol* permet d'interroger un appareil sur les services et les profils qu'il supporte. Enfin un large ensemble de profils fournissent les fonctionnalités dédiées à différents usages, par

exemple la connexion d'un kit main-libres, l'échange de fichiers, le partage de connexion internet, etc.

Durant cette thèse, nous nous sommes concentrés sur les couches situées dans le contrôleur.

2.1.2 Architecture d'un équipement Bluetooth

Voyons maintenant comment sont implémentés les différents éléments d'une pile Bluetooth. Rappelons d'abord que le contrôleur et l'hôte ne sont pas nécessairement conçus par le même acteur. Le contrôleur est un composant dédié vendu par un constructeur spécialisé. L'hôte peut être un ordinateur, un smartphone ou un objet connecté, qui contient sa propre implémentation des couches supérieures Bluetooth.

Hôtes Bluetooth

Les piles Bluetooth hôtes les plus répandues sont celles qui sont intégrées aux les systèmes d'exploitation les plus utilisés. Plusieurs solutions ont été retenues. La pile Bluetooth de Linux (BlueZ) gère l'intégralité des couches hôtes Bluetooth dans le noyau. Dans Windows, la pile Bluetooth est également gérée par le noyau, via un driver principal, qui peut ensuite charger d'autres drivers pour gérer les profils nécessaires. Lorsque la pile Bluetooth est située dans le noyau du système d'exploitation, les vulnérabilités peuvent avoir des conséquences importantes. Par exemple, les failles CVE-2017-1000251 et CVE-2017-1000250 de BlueZ présentées dans BlueBorne [SV17], permettaient de prendre le contrôle du noyau Linux via un lien Bluetooth non authentifié.

Dans les OS mobiles Android et iOS, les piles Bluetooth sont exécutées par des démons en espace utilisateur, avec un ensemble de permissions réduit, ce qui limite quelque peu les risques posés par les vulnérabilités.

Contrôleur Bluetooth

L'architecture des contrôleurs Bluetooth est moins bien connu que celui des hôtes. En principe il n'est pas nécessaire de savoir comment marche un contrôleur Bluetooth pour pouvoir l'utiliser. Il faut seulement connaitre le protocole HCI. Le fonctionnement interne des contrôleurs est donc fermé et peu documenté.

Mais pour expliquer les enjeux de sécurité des contrôleurs Bluetooth, il nous faut décrire succinctement leur architecture, et en particulier distinguer les opérations qui sont réalisées par du logiciel de celles qui sont faites par du matériel. Reprenons la figure 2.1. Les couches les plus basses sont gérées par des périphériques matériels. Cela inclut la couche radio pour des raisons évidentes, mais aussi une bonne partie de la couche *baseband*. Cette dernière réalise des tâches simples qui doivent être réalisées en temps réel, comme l'encodage et le décodage des paquets, le chiffrement, et la couche de contrôle du lien. Il semble donc plus pratique de les implémenter avec une logique dédiée [1].

1. En fait, les implémentations pourraient varier, nous n'avons pas toute la documentation.

Les couches LMP et HCI sont pour leur part implémentées sous forme de logiciel (*firmware*). Ces tâches sont plus complexes car elles ont à gérer les nombreux messages LMP et HCI de la norme, mais elles n'ont pas besoin de fonctionner en temps réel. Les contrôleurs Bluetooth intègrent donc de petits processeurs qui peuvent être basés sur diverses architectures : ARM-Cortex, MIPS, 8051, Xtensa, etc. Ces processeurs ont une fréquence de fonctionnement faible de l'ordre d'une dizaine de mégahertz et une consommation d'énergie réduite. Le plus souvent le *firmware* est exécuté depuis une mémoire ROM. Néanmoins les contrôleurs prévoient des mécanismes permettant d'appliquer des correctifs sur le *firmware*, soit par l'ajout de patchs en RAM, soit avec des moyens de patcher temporairement la ROM.

2.2 Détection d'intrusion

Les communications et les réseaux sans fil sont au centre de nombreuses problématiques de sécurité. Ils peuvent être la cible d'attaques distantes, visant à manipuler les communications ou à obtenir un accès illégal à un système d'information. Ces attaques peuvent tirer partie de faiblesses dans les fonctions de sécurité des protocoles, ou de failles de conception dans les équipements.

Lorsque sont apparus les premiers réseaux WLAN, les mesures existantes de sécurité ont d'abord été adaptées. Ainsi, les pare-feu permettent de limiter l'exposition d'une interface réseau. Les systèmes de détection et de prévention d'intrusion (IDPS) peuvent identifier et bloquer des intrusions sur les services qui restent exposés. Cependant, les objets connectés posent de nouveaux défis aux solutions classiques de sécurité.

Une tendance actuelle consiste à greffer des composants de communication sans fil et des batteries à des objets du quotidien. On parle d'objets **connectés** ou d'objets **communicants**. Ces objets font face à différentes contraintes, et peuvent puiser dans un large choix de normes de communication pour y répondre. En outre, certains fabricants continuent d'utiliser leurs propres protocoles non standardisés.

Pour tenir compte de ces objets connectés, les techniques de détection d'intrusion doivent donc pouvoir s'adapter à des environnements hétérogènes qui voient cohabiter de multiples protocoles de communication. Par ailleurs ces objets ont de faibles ressources et peuvent difficilement être mis à jour.

Les systèmes de détection d'intrusion (IDS) servent à détecter des tentatives d'intrusion. Ils peuvent aussi chercher à les bloquer activement, ce sont alors des systèmes de prévention d'intrusion (IPS). On parle plus généralement de systèmes de détection et de prévention d'intrusion (IDPS).

Un IDS se compose d'au moins trois éléments :

— une **sonde** chargée de collecter des informations sur l'état du système ou du réseau ;
— un **analyseur** qui traite ces informations, recherche des tentatives d'attaques, et génère des alertes ;
— une **base de donnée** qui stocke les alertes.

Un IDPS comprendra également des contremesures pour lui permettre d'interrompre les tentatives d'intrusion détectées.

Il existe différents types d'IDS, qu'on distingue selon leur placement et le type d'information qu'ils analysent. Par exemple, les NIDS (Network-based IDS) sont placés au niveau du réseau et analysent le trafic. Les HIDS (Host-based IDS) sont placés sur l'hôte à protéger (par exemple, un anti-virus). Les WIDS analysent pour leur part les échanges Wi-Fi au niveau du canal sans fil.

Les méthodes d'analyse peuvent être résumées en deux grandes catégories : la détection de signatures et la détection d'anomalies. Des IDS hybrides peuvent mettre en œuvre ces deux types d'analyse.

2.2.1 Détection de signature

La détection de signatures exploite une **base de connaissance** qui recense des motifs d'attaques connues (ou **règles de détection**). La détection consiste à rechercher des motifs d'attaques connues dans les informations remontées par les sondes. Lorsqu'un motif correspondant à une tentative d'intrusion est détecté, une alerte est générée.

Snort[Mar99] est un NIDS Open Source qui utilise une détection par signature, et dispose d'une large base de connaissance régulièrement mise à jour [2]. Suricata[sur] utilise la même approche, et apporte le support du multi-threading et des GPU. Il est compatible avec les règles de détection écrites pour Snort.

La détection de signatures a l'avantage de provoquer peu de faux positifs, car les alertes générées reposent sur des motifs malveillants préalablement identifiés. Cependant, ses performances reposent beaucoup sur la qualité de sa base de connaissance, qui doit être exhaustive et régulièrement mise à jour. La taille de la base de connaissance peut également poser des problèmes de performances, puisqu'un grand nombre de signatures doivent être recherchées dans chaque échange réseau. Bien sûr, ce type de détection ne permet que de détecter des attaques identifiées à l'avance, et ne peut donc pas détecter l'exploitation d'une vulnérabilité *0-day* [3].

2.2.2 Détection d'anomalie

La détection d'anomalies exploite un modèle qui représente l'état normal d'un équipement ou du trafic réseau. La détection consiste à évaluer les informations récoltées pour mesurer leur niveau d'adéquation avec le modèle. Lorsqu'une anomalie dépasse le seuil de détection, un alerte est générée.

Le modèle peut être défini par un expert – ce qui peut représenter un travail considérable – ou construit par apprentissage automatique avec des méthodes d'apprentissage supervisées ou non supervisées.

La détection d'anomalie permet de créer des IDS très sensibles, capables de détecter des attaques jusqu'alors inconnues, ou tout évènement imprévu. Cependant, leurs performances

2. Pour les abonnés payants.
3. zero-day : vulnérabilité non corrigée n'ayant fait l'objet d'aucune publication

dépendent essentiellement de la qualité du modèle. Le choix des propriétés qui seront modélisées et la méthode utilisée pour construire le modèle sont donc déterminants. Enfin, les IDS basés sur la détection d'anomalie génèrent plus de faux-positifs que ceux qui sont basés sur des signatures. Il est donc important que les alertes générées par l'IDS soient interprétables par un expert humain, pour qu'il puisse rapidement identifier les alertes pertinentes.

Des solutions de sécurité se basent sur la détection d'anomalies dans des flux réseaux. La solution Netflow de Cisco[SF02] puis le standard IPFIX [4] donnent aux équipements réseau un rôle de sonde, qui remontent des statistiques sur les flux réseau, les ports et les interfaces utilisées, etc. Les informations remontées par NetFlow et IPFIX permettent de créer un modèle d'un système d'information du point de vue des flux réseau. Cela rend possible des approches de détection d'intrusion basées sur la détection d'anomalie.

Terzi et al.[TTS17] proposent un IDS basé sur des informations NetFlow, mettant en oeuvre une détection d'anomalie à base d'apprentissage non-supervisé. Le trafic réseau est représenté par quelques caractéristiques mesurées chaque minute : adresse IP, nombre de ports utilisés, taille des données et nombre de paquets. Ces traits sont ensuite normalisés et réduits par analyse en composantes principales (PCA). Un partitionnement en k-moyennes à deux classes est enfin utilisé pour distinguer les échanges normaux et anormaux.

Cette méthode a l'avantage de ne pas se reposer sur le contenu des échanges. Elle peut donc s'appliquer à tous types de services, même si leurs communications sont chiffrées, tout en respectant la vie privée des usagers. En revanche, elle reste tributaire de sa représentation schématique du réseau, et est donc mieux adaptée pour détecter des attaques en déni de service ou des connexions manifestement imprévues, que pour distinguer des intrusions visant la couche applicative.

2.2.3 Sondes pour l'IoT

Les objets connectés utilisent toute une variété de protocoles sans fil, ce qui crée un environnemnt radio hétérogène qui pose un défi aux solutions de sécurité classique. Effectivement, il est difficile d'adjoindre des HIDS à des objets connectés. Leurs capacité en mémoire et de calcul sont réduites, leur logiciels sont fermés et il n'est pas pratique d'y installer de nouveaux programmes. Pour ces objets, il semble donc préférable de placer un IDS au niveau du réseau. Cela peut se faire à différents niveaux.

IoT Sentinel [MMH+17] est un NIDS dédié aux objets connectés, placé au niveau de la passerelle internet qui observe le trafic ethernet. Il est capable d'identifier avec précision les appareils, par un *fingerprinting* des métadonnées de leurs paquets réseau. En fonction du type d'appareil détecté, il peut imposer des contraintes sur le trafic des appareils, à la manière d'un pare-feu, pour limiter les conséquences en cas de compromission de l'un d'entre eux.

L'approche d'IoT Sentinel permet de sécuriser des appareils hétérogènes, et l'utilisation du *fingerprinting* facilite l'interprétation du contexte réseau. Cependant, cette approche est

4. `https://tools.ietf.org/html/rfc7011`

liée à l'utilisation d'une passerelle internet, et ne permet pas de surveiller les communications sans fil entre des appareils qui n'en utilisent pas.

IoTScanner [SMT17] est justement pensé pour surveiller des réseaux sans fil hétérogènes. Pour cela, il utilise une collection de récepteurs radio dédiés, un pour chaque protocole, pour récupérer respectivement les trames Wi-Fi, 802.15.4, et Bluetooth Low-Energy. Le trafic capturé est représenté par quelques métadonnées : adresses présentes dans les paquets, type du paquet, taille, RSSI [5], etc. Les flux de données échangées par les appareils sans fil peuvent ensuite être quantifiés. Les auteurs remarquent que le modèle de certains appareils peut être identifié par des heuristiques simples. Le fait d'utiliser des récepteurs dédiés fait à la fois la force et la faiblesse d'IoTScanner. Ces récepteurs remontent directement des paquets démodulés et décodés, les informations sont donc précises, de faible volume, et leur traitement nécessite peu de puissance de calcul. En contrepartie, il faut ajouter du matériel à chaque fois que l'on souhaite supporter un nouveau protocole sans fil. De plus certains protocoles se prêtent moins bien à une écoute passive, et il n'est pas forcément possible de trouver un récepteur accessible et fonctionnel pour les surveiller.

La section suivante introduit le domaine des radio-logicielles, qui constituent une piste sérieuse pour la réalisation de sondes de détection d'intrusion sans fil.

2.3 Radio Logicielle

Une radio logicielle (SDR : *Software Defined Radio*) est un récepteur ou un transmetteur radio, dans lequel des fonctions habituellement réalisées par du matériel sont implémentées en logiciel. En particulier la modulation et la démodulation du signal ne sont plus faites par un matériel fixe, mais sont réalisées par un logiciel exécuté sur un CPU, un GPU, ou un FPGA. Il est donc possible d'implémenter différents protocoles sans fil, avec un même matériel, en ajoutant du logiciel.

2.3.1 Généralités

Le matériel utilisé par les SDR implémente un sous-ensemble des fonctionnalités d'un équipement radio traditionnel. Nous distinguons deux grandes catégories de matériel SDR, selon le type d'échantillonage qu'ils utilisent.

Les interfaces à base d'échantillonnage direct (*direct-sampling*) se constituent essentiellement d'un étage de filtrage et d'amplification du signal, avant la conversion analogique-numérique (CAN). Leur architecture est donc assez simple, et peut être réalisée à partir d'un oscilloscope, voire d'une carte son d'ordinateur. En revanche, ils ne peuvent couvrir que les fréquences couvertes par leur vitesse d'échantillonnage, et peuvent difficilement être utilisées pour travailler à des fréquences excédant la centaine de mégahertz.

Pour couvrir des fréquences supérieures à faible coût, on utilise généralement du matériel dérivé de l'architecture super-hétérodyne, dans laquelle le signal d'entrée est d'abord mélangé avec une fréquence générée par un oscillateur local (LO), pour être transposée

5. Received Signal Strength Indication : la puissance du signal

dans une fréquence intermédiaire (IF) nettement plus basse et qui ne dépend plus de la fréquence porteuse. Ces récepteurs utilisent un échantillonnage I/Q (*in-phase and quadrature sampling*), qui représente les échantillons sous formes de nombres complexes (I et Q). Cette méthode permet en fait de réduire les coûts, car elle permet de gérer une bande de fréquence d'une largeur équivalente à la fréquence d'échantillonnage, en utilisant deux CAN fonctionnant en parallèle, plutôt qu'un seul CAN qui devrait avoir une vitesse d'échantillonnage doublé pour reconstituer le signal en vertu du théorème d'échantillonnage de Shannon.

2.3.2 Applications à la détection d'intrusion

Les radio logicielles peuvent répondre à deux enjeux de la détection d'intrusion sans fil. D'abord, elles peuvent s'adapter à de nombreux standards de communication actuels et futurs, et donc offrir une solution aux environnements sans fil hétérogènes. Ensuite, les SDR rendent possible des méthodes de *fingerprinting* sur la couche physique qui permettent d'identifier des transmetteurs. Il est ainsi possible de mieux contextualiser des transmissions sans fil dont il serait autrement difficile d'attester la provenance.

Le *fingerprinting* consiste à observer de subtiles variations dans la façon dont des transmetteurs implémentent un protocole. Une empreinte radio (RFF : *RF Fingerprint*) peut ainsi être mesurée pour chaque appareil, pour ensuite pouvoir l'identifier. La couche physique des équipements sans fil fournit une riche source de traits caractéristiques.

Par exemple, le signal transitoire d'allumage (*transient*) est un indicateur très sensible, qui peut différer même sur des appareils du même modèle [HBK06, SLH11]. Ce signal est en revanche très court, puisqu'il correspond à la période d'allumage du transmetteur. Il faut donc mesurer l'amplitude du signal à une très haute vitesse d'échantillonnage, de l'ordre du gigahertz, pour pouvoir le mesurer avec une précision suffisante.

D'autres méthodes se basent sur des variations dans la modulation du signal. Une portion du signal démodulé est comparée avec un signal de référence modulé numériquement [BBGO08, ZJZ+18]. Les différences entre les deux signaux peuvent ensuite être exprimées par un petit nombre de traits caractéristiques, comme un décalage de fréquence, de phase, un biais entre les composantes I et Q, etc. Les méthodes basées sur la modulation ont l'avantage d'être plus simples à mettre en œuvre. Les propriétés mesurées sur le signal modulé sont plus résistantes au bruit de fond que celles mesurées sur la forme d'onde de l'amplitude. De plus, beaucoup de ces propriétés peuvent être mesurées à une vitesse d'échantillonnage plus raisonnable, proche de celle utilisée par un récepteur conventionnel. En revanche, ces méthodes sont moins précises. Elles permettent de distinguer des appareils de modèles différents, mais confondent généralement les appareils du même modèle. Enfin, les caractéristiques mesurées sur la modulation doivent être définies à l'avance, et ne peuvent donc pas être généralisées à tous les protocoles sans le travail d'un expert.

Plusieurs recherches récentes mettent en œuvre un apprentissage profond pour extraire directement des traits caractéristiques, et ainsi produire des techniques qui peuvent se généraliser à plusieurs protocoles sans nécessiter l'intervention d'un expert. Robyns et al. [RML+17] présentent une méthode de *fingerprinting* sur la couche physique LoRa, en

réalisant un apprentissage profond sur les échantillons qui constituent un symbole LoRa. Pour tenir compte du décalage de fréquence, une FFT est réalisée sur le signal pour le transposer dans le domaine fréquentiel. Cette méthode leur permet de distinguer avec précision des appareils de constructeurs différents, et dans une moindre mesure, des appareils du même modèle. Cependant, les empreintes ainsi crées sont fortement dépendantes des conditions du canal radio et du rapport signal sur bruit (SNR), et elles ne permettent plus d'identifier les appareils une fois qu'ils ont été déplacés.

Jafari et al. [JOA+19] proposent une approche similaire, dans laquelle ils évaluent les performances de plusieurs techniques de classification basées sur l'apprentissage profond, pour distinguer six transmetteurs 802.15.4 du même modèle. Leur réseau profond prend en entrée les échantillons bruts d'un signal. Les auteurs considèrent l'effet du SNR : dans le jeu de données utilisé, chaque appareil est configuré pour transmettre des messages à cinq niveaux de puissance différents. Ils peuvent ainsi distinguer avec précision les appareils de leur échantillon. En revanche, les empreintes ainsi constituées ne sont pas évaluées sur des transmissions extérieures au jeu de données.

Roux et al. proposent RadIoT [RAA+18] pour détecter des anomalies dans un environnement radio hétérogène. Il observe le spectrogramme de larges bandes de fréquences de l'ordre de 100 MHz. Le spectrogramme peut être découpé pour isoler des canaux de communication prédéfinis. Chaque mesure du spectrogramme est représentée par quelques propriétés statistiques. Les propriétés d'une suite de spectrogrammes sont fournies en entrée d'un autoencodeur qui est entrainé pour apprendre à compresser ses entrées et à les reconstruire. La détection d'anomalie se base sur l'erreur observée entre les données reconstruites et les données d'entrée. L'approche est testée en environnement réel, avec plusieurs attaques et des données capturées par un HackRF *en mode balayage*. L'approche montre des résultats satisfaisants, permettant notamment de détecter avec précision les attaques en déni de service, ou l'ajout d'un faux point d'accès Wi-Fi. Elle semble cependant moins efficace pour identifier des attaques qui respectent la sémantique du protocole, et qui peuvent difficilement être détectées sans démoduler les paquets.

L'apprentissage profond montre des résultats encourageants, mais nous pensons qu'il ne permet pas encore de se passer totalement du travail des experts. Une démodulation plus complète du signal peut permettre non seulement d'extraire des informations utiles des paquets, comme des adresses et des identifiants, mais aussi faciliter la collecte d'empreintes radiométriques. Nous pensons que ce type d'information de plus haut niveau pourrait fournir des entrées précieuses aux systèmes de détection d'anomalies. C'est ce qui a motivé les travaux que nous présentons dans les chapitres 4 et 5.

2.4 Sécurité des *baseband*

L'étude de la sécurité des modems *baseband* fait l'objet d'une littérature abondante depuis quelques années.

Effectivement, les modems posent des contraintes particulières. D'abord, ces composants sont propriétaires. Le code source et leur documentation sont donc confidentiels, ce

qui complexifie la recherche de vulnérabilités. Les mécanismes d'authentification des *firmwares* peuvent également empêcher leur débogage. En l'absence de composants ouverts, les chercheurs se trouvent parfois dépourvus des outils nécessaires pour dialoguer avec les couches protocolaires implémentées par les modems.

La première partie de cette section présente des recherches de vulnérabilités dans des modems sans fil. La seconde partie présente des travaux sur l'instrumentation des *firmwares*, leur émulation et leur débogage. La troisième partie présente des implémentations Open Source de *baseband* sans fil.

2.4.1 Recherche de vulnérabilités

En 2010, Weinmann [Wei10] expose les risques posés par les vulnérabilités des modems cellulaires. Dans leurs *firmwares*, il recherche des défauts inhérents aux langages de programmation bas niveau : vérifications insuffisantes sur des tailles, durée de vie des objets, dépassements d'entiers, fuites d'informations. Des vulnérabilités sont identifiées dans les modems de différents constructeurs. Weinmann peut démontrer des attaques pratiques, à l'aide d'une antenne relais en radio logicielle basée sur le logiciel OpenBTS [6] fonctionnant sur un USRP [7].

L'approche de Weinmann repose essentiellement sur l'ingénierie inverse et une revue manuelle des fonctions importantes. Ce même type d'approche a récemment conduit à la découverte et à la correction de plusieurs vulnérabilités dans des chipsets Wi-Fi.

En 2017 également, des études sur les chipsets Wi-Fi Broadcom ont montré qu'ils contenaient des vulnérabilités aux conséquences parfois importantes [Ben17, Art17, Ang19]. En observant les fonctions qui décodent le contenu des trames Wi-Fi, Beniamini identifia plusieurs dépassements de tampon et décrivit une attaque qui visait le chipset Wi-Fi d'un iPhone 7. Il pouvait ensuite exploiter une seconde vulnérabilité pour prendre le contrôle du noyau de l'iPhone 7 depuis la puce Wi-Fi [Ben17]. Artenstein poursuivit une approche similaire, puis utilisa le chipset Wi-Fi compromis pour rediriger le navigateur mobile vers un site web malveillant [Art17].

Par la suite des chercheurs se sont penché sur la sécurité des modems Bluetooth.

En 2017, l'entreprise Armis avait présenté [SV17] une collection de vulnérabilités critiques dans les couches hôte Bluetooth de Linux, Android, Windows et MacOS. Pour les découvrir, ils avaient notamment passé en revue l'implémentation des mêmes fonctionnalités dans plusieurs piles différentes. Ils avaient constaté des défauts similaires dans plusieurs implémentations. En 2019, la même équipe publia BLEEDINGBIT [SZV19] qui portait cette fois sur des contrôleurs Bluetooth Low Energy de la marque Texas Instrument. L'une des failles identifiées permettait l'exécution de code arbitraire dans le contexte du contrôleur Bluetooth.

En 2020, Kovah [Kov20] présenta les résultats d'une étude similaire, qui décrivait plusieurs vulnérabilités dans des contrôleurs Bluetooth Low Energy de Texas Instrument et

6. OpenBTS : `http://openbts.org/`
7. USRP : Universal Software Peripheral Radio

Silicon Labs.

Ainsi, dix ans après les travaux de Weinmann, la sécurité des modems sans fil demeure un sujet assez peu étudié, et de rapides revues de leurs *firmwares* suffisent encore à y identifier rapidement des vulnérabilités critiques.

La section suivante présente un autre aspect de la recherche sur la sécurité des modems sans fil : les techniques d'instrumentation et de débogage.

2.4.2 Instrumentation

Il est plus facile d'analyser la sécurité d'un appareil lorsqu'on dispose de capacités avancées de disgnostic. Mais les fabricants ne mettent généralement pas les outils de débogage du *baseband* à disposition des usagers. Cette section présente deux approches alternatives pour faciliter le débogage des processeurs *baseband*.

Instrumentation et Débogage. La première approche pour permettre une analyse dynamique d'un *firmware* est d'utiliser ou de développer des techniques pour déboguer le *firmware* directement sur sa plateforme matérielle originale.

En 2011, Guillaume Delugré a présenté ses travaux sur la rétro-ingénierie de *firmwares* de *baseband* Qualcomm, et l'implémentation d'un débogueur (*Qcommbbdbg*) pour des clés 3G USB basées sur le chipset MSM6280 [Del11]. Les modems Qualcomm exposent une interface (`DIAG`) pour diagnostiquer le modem. Dans les modems 3G utilisés par Delugré (Icon 225 & Evolution), l'interface `DIAG` comportait des commandes permettant de lire et d'écrire dans la mémoire vive du modem. Qcombbdbg utilise ces primitives pour s'installer dans le *firmware* du *baseband*. Cela lui permet d'ajouter de nouvelles commandes dans l'interface `DIAG`, pour intégrer son débogueur. Le *firmware* pouvait ensuite être débogué dans *gdb*[8].

En 2020, Berard et al. se sont inspirés de l'approche de Delugré, pour implémenter un débogueur pour le modem téléphonique des smartphones Samsung récents (Shannon) [BF20]. Sur ces téléphones modernes, les *firmwares* des processeurs *baseband* doivent être signés par le constructeur pour pouvoir être chargés en mémoire. La vérification de la signature est effectuée par le moniteur de *TrustZone* de l'appareil, qui s'exécute avec des droits supérieurs à ceux de l'OS Android, et qui normalement ne peut pas être modifié, pas même avec un noyau Android modifié.

Pour contourner la vérification de la signature sur les *firmwares* des modems, Berard et Fargues ont profité d'une chaine de vulnérabilité publiée précédemment par des chercheurs de Quarkslab, permettant l'exécution de code arbitraire avec les privilèges du moniteur de TrustZone (EL3) [Qua19].

L'équipe Seemo-lab a également travaillé sur l'instrumentation de modems sans fil, pour faciliter leur diagnostic, mais également pour leur ajouter de nouvelles fonctionnalités. NexMon [SWH15] est une suite logicielle pour modifier le *firmware* de puces Wi-Fi Broadcom, qui permet notamment de leur faire supporter le mode *moniteur*. InternalBlue [MCSH19]

8. GDB : GNU Debugger

s'inscrit dans la continuité de NexMon, et cible des contrôleurs Bluetooth Broadcom. Il étend notablement les capacités de diagnostic sur ces contrôleurs, en activant des interfaces de déboguage existantes, et en ajoutant des patches dans le *firmware*. Cela permet la récupération des rapports d'erreurs, l'inspection du trafic LMP, ainsi que l'injection de paquets LMP et LLCP.

Émulation. Une seconde approche plus radicale pour analyser dynamiquement un *firmware* est l'émulation. Le *firmware* est d'abord extrait de la mémoire du modem, et peut ensuite être exécuté directement sur un ordinateur.

Le fonctionnement d'un processeur *baseband* peut être émulé assez simplement grâce à des outils comme QEMU [qem] ou Unicorn Engine [uni]. C'est en fait l'émulation des périphériques matériels qui pose le plus de difficultés.

Plusieurs solutions peuvent être utilisées pour tenir compte des périphériques. D'abord, il est possible de les émuler en écrivant un logiciel spécifique, si les périphériques ne sont pas trop complexes. Autrement, les accès de l'émulateur aux périphériques peuvent simplement être reproduits sur le matériel d'origine, lorsque la latence le permet. Enfin, on peut contourner le problème en n'émulant que certaines parties du *firmware*, typiquement en partant d'un *snapshot* de la mémoire du modem déja initialisé, pour pouvoir émuler sélectivement des appels aux fonctions importantes.

L'émulation est souvent utilisée pour recherche des vulnérabilités dans les *firmwares* par *fuzzing*. Le *fuzzing* consiste à envoyer de nombreuses entrées malformées dans un programme, et à observer les *crashs* pour rapidement identifier des bogues.

L'émulation de *firmwares baseband* est un sujet activement recherché en 2020. Maier et al. présentent BaseSAFE [MSP20], un émulateur de *baseband* cellulaires Mediatek (basé sur Unicorn Engine [uni]) pour faciliter leur *fuzzing*. Ils testent les fonctions de traitement des messages de signalisation LTE avec le *fuzzer* AFL [afl]. Sur un seul coeur Intel i7, ils peuvent exécuter environ 1500 tests par seconde.

Hernandez et al. [HMT+] présentent une approche similaire, pour émuler le *baseband* cellulaire Shannon de Samsung et appliquer des méthodes de *fuzzing*. Leur environnement est basé sur le moteur d'analyse dynamique PANDA [DGHH+15] (lui même basé sur QEMU [qem]) et la plateforme d'orchestration Avatar^2 [MNFB18]. Ils soulignent que le *fuzzing* est une méthode incontournable pour analyser de larges *firmwares* sans leur code source. Selon eux, l'émulation est la seule méthode de *fuzzing* satisfaisante pour des modems cellulaires, car les tests en conditions réelles posent de multiples obstacles et il est difficile d'en analyser les résultats.

Rugue et al. proposent Frankenstein [RCGH20] dans la continuité d'InternalBlue. Ce projet basé sur QEMU [qem] leur permet d'émuler le *firmware* de contrôleurs Bluetooth Broadcom et Cypress, pour les analyser dynamiquement et les fuzzer. Leur émulation est basée sur un *snapshot* de la mémoire vive du contrôleur déjà initialisé. Le contrôleur ainsi émulé peut être attaché à la pile Bluetooth de Linux (BlueZ [blu]). Cela leur permet d'identifier des bogues dans le *firmware* du contrôleur, mais également dans des piles Bluetooth hôte.

L'émulation est donc une méthode utile pour découvrir automatiquement des vulnérabilités dans de larges bases de code fermées, qu'il serait difficile de passer manuellement en revue. Les audits manuels restent cependant primordiaux car ils permettent d'identifier des vulnérabilités critiques et de bien comprendre leurs implications.

Les *firmwares* des contrôleurs Bluetooth présentent une surface d'attaque intéressante, mais leur taille reste assez modeste comparée à celle des *firmwares* de modems cellulaires. Pour les contrôleurs Bluetooth, l'approche classique de rétro-ingénierie et d'analyse manuelle est à notre avis la première qui doit être menée. C'est celle que nous décrivons dans le chapitre 6.

2.5 Implémentations ouvertes de *baseband*

Pour comprendre et étudier les *baseband* fermés contenus dans nos modems sans fil, il est très utile de disposer d'implémentations ouvertes. Une implémentation ouverte d'une couche protocolaire fournit plusieurs outils précieux. D'abord elle permet d'acquérir rapidement une appréhension pratique du protocole, et constitue en cela une importante base éducative pour ensuite mieux comprendre les implémentations fermées. Ensuite, elle permet d'élaborer des outils de diagnostic du protocole, pour interagir avec d'autres appareils et déboguer leurs communications. Enfin, ces implémentations ouvertes peuvent remplacer entièrement des *firmwares* fermés. Ainsi, on peut communiquer en utilisant des piles de communication dont il est nettement plus simple de vérifier la sécurité.

2.5.1 GSM

Global System for Mobile communications (GSM) est la norme pour les réseaux mobiles de deuxième génération, adoptée au début des années 1990. Il existe des implémentations Open Source pour la plupart des composants d'un réseau GSM, ce qui facilite considérablement les recherches sur la sécurité de ce protocole.

Cette section présentons OsmocomBB, un *baseband* GSM Open Source qui nous a fortement inspiré.

OsmocomBB. Osmocom est une famille de projets d'implémentations libres de protocoles de communication mobiles. Il propose entre autre une implémentation complète de réseau GSM, comprenant un *Base Station Controler* (OsmoBSC) et un *Base Station Transceiver* (OsmoBTS).

En plus de la partie réseau du GSM, le projet Osmocom propose une implémentation libre d'une pile GSM pour téléphone mobile (OsmocomBB) [WM10]. OsmocomBB permet de remplacer intégralement le *firmware* GSM de plusieurs téléphones mobiles, tous basés sur le chipset Calypso de Texas Instrument (TI Calypso).

Les auteurs d'OsmocomBB ont choisi de baser leur *baseband* Open Source sur les chipset TI Calypso car une grande quantité de documentation était disponible pour ces composants. D'abord, la *datasheet* du TI Calypso est disponible sur internet, ce qui facilite gran-

dement sa compréhension. De plus, le code source d'un téléphone basé sur le TI Calypso (TSM30) avait été publié (par erreur ?) sur le site *sourceforge*, avec toute l'implémentation des couches *baseband* d'une station mobile GSM [9]. Cette documentation a permis aux auteurs d'OsmocomBB de réaliser un *firmware* entièrement libre qui fonctionne sur les TI Calypso.

Avant de débuter le développement du *firmware* OsmocomBB, Dieter Spaar avait déjà démontré une vulnérabilité dans le protocole GSM, qui permet à un téléphone mobile de causer un déni de service sur une antenne relais en envoyant un grand nombre de requêtes de connexions (*Rach-flooding*) dans un court laps de temps. Cette démonstration était basée sur un téléphone TSM30, dont Spaar avait légèrement modifié le *firmware* [Spa09]. C'est après cette première démonstration que Spaar et Welte ont entrepris le développement d'OsmocomBB.

OsmocomBB a permis de découvrir les possibilités étonnantes des téléphones GSM. Par exemple, une version modifiée d'OsmocomBB publiée en 2013 permet d'utiliser un téléphone comme une BTS, et ainsi d'émuler un réseau mobile [Mun13]. Nico Golde a utilisé OsmocomBB pour montrer comment des vulnérabilités dans les configurations de certains réseaux mobiles pouvaient être exploitées pour usurper l'identité d'un utilisateur mobile, intercepter ses communications, ou les rendre indisponibles [GRS13]. OsmocomBB est également utilisé par le projet *CatcherCatcher* de SRlabs [srl13], qui vise à la détection des IMSI-catchers [10]. Enfin, j'avais utilisé OsmocomBB dans mon projet de fin de master 1 pour implémenter des attaques en relais sur un opérateur téléphonique français, en profitant d'une vulnérabilité dans la configuration de leurs antennes relais.

La première version d'OsmocomBB a été publiée en 2010. Harald Welte, l'un des fondateurs d'Osmocom avait exprimé ses motivations ainsi [Wel10] :

« Il est temps vous ne trouvez pas ? Après 19 années de code propriétaire tournant dans les puces GSM de milliards de téléphones, il est plus que temps d'apporter la lumière de la liberté dans ce coin sombre de l'informatique. Pour moi, c'est le Saint Graal du logiciel libre : arriver à le répandre au-delà du PC, au-delà des systèmes d'exploitation et des programmes utilisateurs. L'implanter dans les milliards de périphériques embarqués où tout le monde est actuellement coincé avec du logiciel propriétaire sans alternative. Les gens tiennent pour acquis le fait de devoir faire tourner des mégaoctets de code propriétaire, sans aucune protection de la mémoire, sur un réseau non sécurisé (GSM). Qui ferait ça avec son PC sur Internet, sans un pare-feu et un flot constant de corrections de sécurité pour le logiciel ? Et pourtant des milliards de gens font ça avec leur téléphone en permanence. J'espère qu'avec notre travail viendra un temps où les gens qui ont payé pour leur téléphone seront à même de vraiment le posséder et contrôler ce qu'il fait. »

Je partage ces préoccupations, qui ont profondément influencé mes travaux durant cette thèse.

9. Le code du TSM30 est disponible ici :`https://cryptome.org/tsm30/tsm30.7z`

10. Les IMSI-Catchers sont des appareils permettant d'identifier les utilisateurs des téléphones mobiles proches en simulant une antenne relais.

2.5.2 Bluetooth Low Energy

OsmocomBB reste un projet expérimental qui n'est pas destiné à des utilisations grand public.

Il existe deux projets de *baseband* Open Source pour le Bluetooth Low Energy (BLE), qui supportent entièrement la spécification et peuvent donc être intégrés dans des produits finis. Ces projets fournissent des piles de communication BLE complètes, incluant l'hôte et le contrôleur.

Nimble [apa] est la pile de communication BLE du projet Apache-Mynewt, un OS temps réel dédiés aux objets connectés. Nimble comprend notamment une implémentation complète du contrôleur BLE, qui fonctionne sur les chipset NRF5x de Nordic Semiconductor. Les NRF5x sont des puces de communication 2,4 GHz qui sont conçues pour supporter le BLE. De plus la documentation détaillée des NRF5x est entièrement publique. Cela en fait une cible de choix pour le développement d'outils de communications ouverts.

Zephyr [zep] est un projet de la fondation Linux. C'est également un OS temps réel dédié aux objets connectés. Comme Nimble, il dispose de sa propre implémentation d'un contrôleur Bluetooth Low Energy (contribuée par Nordic Semiconductor) qui fonctionne également sur les chipsets NRF5x.

Ainsi, il existe déjà des implémentations parfaitement fonctionnelles de contrôleurs Bluetooth Low Energy. Elles peuvent facilement être étudiées, modifiées, et peuvent servir de base pour créer des outils de communication facilement auditables. Il apparait que ces deux projets n'auraient probablement pas vu le jour si Nordic Semiconductor n'avait pas publié la documentation détaillée du NRF5x. En effet, la plupart des constructeurs publient peu ou pas de communication pour leurs modems, ce qui rend très ardue l'écriture d'un *firmware* pour ces appareils.

Malheureusement il n'existait pas d'implémentations ouvertes de contrôleur Bluetooth Classic, ce qui limitait le champ des expérimentations sur ce protocole. Nous avons donc entrepris le développement d'une couche *baseband* Bluetooth Classic, présentée dans le chapitre 7.

À notre connaissance, il n'existe aucun composant Bluetooth Classic dont la documentation soit entièrement publique. Notre *baseband* est donc purement logiciel, et fonctionne sur du matériel qui à la base n'était pas prévu pour supporter le Bluetooth Classic.

2.6 Conclusion

Ce chapitre introduit les principaux concepts abordés dans ce document, et les principaux travaux existants qui s'y rapportent. Le chapitre suivant revient plus en détail sur notre problématique, et motive précisément les approches poursuivies dans cette thèse.

Chapitre 3

Problématique

Cette section expose plus en détail mon point de vue sur l'état de la sécurité des communications sans fil, pour en venir aux solutions que j'ai privilégiées.

La première partie du chapitre présente le problème de la sécurité des communications sans fil tel que je le conçois. La première section présente des défauts connus dans la sécurité de quelques protocoles que nous utilisons habituellement, et la difficulté qu'il y a à corriger ou atténuer ces défauts. Cela nous amène ensuite au sujet de la sécurité des composants qui servent à réaliser les communications sans fil (modems) — un sujet qui rejoint celui de la sécurité des objets communicants. Nous en venons enfin à discuter de l'usage des systèmes de détection d'intrusion (IDS) dans ce contexte, et du délicat problème du positionnement des IDS lorsqu'on considère des communications sans fil.

La seconde partie de ce chapitre présente les solutions que j'ai étudiées, qui constituent le corps de cette thèse.

3.1 Des protocoles peu sécurisés

Les protocoles de communication que l'on utilise couramment se conforment à des standards qui définissent leur fonctionnement dans les moindres détails. C'est grâce à ces efforts de standardisation que l'on peut communiquer au moyen de protocoles complexes en utilisant des appareils de la marque de notre choix.

Par contre, si une vulnérabilité se glisse dans la spécification, alors tous les appareils qui respectent cette spécification peuvent être attaqués.

C'est pourquoi la problématique de la sécurité des communications sans fil passe d'abord par la description de failles de sécurité dans la spécification des protocoles, et des contre-mesures existantes.

3.1.1 Exemple du GSM

Le Global System for Mobile Communication (GSM, 2G) a été mis en place au début des années 1990, et est depuis très largement déployé et supporté par tous les téléphones

mobiles.

Un standard vulnérable. Ce standard possède une faiblesse majeure. Dans le GSM, le mobile est obligé de s'authentifier auprès du réseau pour accéder au service. Malheureusement, le réseau n'est pas authentifié par le mobile. Une téléphone 2G n'a donc aucun moyen de s'assurer que le réseau auquel il est connecté est légitime.

L'autre gros problème est que la station de base peut forcer le mobile à utiliser un chiffrement faible, voire aucun chiffrement. Le standard GSM définis quatre modes de chiffrement (A5/1,A5/2,A5/3), plus l'A5/0 qui revient à communiquer en clair. Le type de chiffrement est négocié entre le mobile et la station de base. Une station malveillante pourra donc forcer une communication à être protégée avec le plus faible mode de chiffrement supporté par le mobile. Le chiffrement A5/2 était volontairement peu sécurisé, et n'est plus supporté par les nouveaux téléphones depuis 2006 [GSM06]. L'A5/1 est encore couramment utilisé, même s'il a été cryptanalysé et est désormais facilement décryptable avec des outils publiques [Noh09]. Des travaux ont montré des faiblesses sur A5/3, mais aucune attaque pratique n'a encore été publiée.

Pour résumer, n'importe qui disposant du matériel adéquat peut déployer une station de base malveillante, et forcer les téléphones qui s'y connectent à utiliser un mode de chiffrement défaillant.

Qui reste exploitable malgré les nouvelles normes. Cette situation est largement exploitée par les fabricants d'IMSI-catchers. Ces appareils, comme leur nom l'indique permettent de récuperer l'IMSI (International Mobile Subscriber Identity) des téléphones environnants, en émulant une station de base de leur opérateur. Les IMSI-catchers peuvent également intercepter les communications, ou interférer avec elles.

Même si les IMSI-catchers sont presque aussi vieux que le GSM, le problème qu'ils posent n'a jamais été réellement adressé.

Les nouvelles normes de téléphonie mobile (3G, 4G) ont introduit l'authentification du réseau, mais sans parvenir à déjouer les attaques existantes. En effet, cette authentification ne s'applique pas à tout le protocole. Ainsi, une fausse station de base 3G peut toujours récupérer l'IMSI d'un téléphone puis lui interdire d'utiliser la 3G, afin de le forcer à basculer sur un réseau 2G [BMP$^+$] (*downgrade attacks*). De même, une station de base 4G non authentifiée peut interdire à l'utilisateur de son choix d'utiliser la 4G et la 3G pour le forcer à basculer vers un faux réseau 2G [SBA$^+$15].

Sans contremesures standardisées. Intéressons nous aux contremesures existantes aux attaques sur la téléphonie mobile. Premièrement, la spécification GSM indique qu'un téléphone mobile doit afficher un avertissement lorsqu'il utilise un réseau non chiffré (A5/0). Cependant, cette fonctionnalité peut être explicitement désactivée par l'opérateur mobile, en activant une option dans la configuration de la carte SIM. Selon Paget [Pag10], tous les opérateurs activent cette option. En conséquence, les utilisateurs ne sont quasiment jamais alertés de l'absence de chiffrement de leurs communications.

Une seconde option est de configurer un téléphone mobile pour lui interdire d'accéder à un réseau 2G. Cela évite les *downgrade attacks* mais est difficilement applicable dans les zones rurales, encore largement dépendantes de la 2G.

Il existe aussi des applications pour smartphones, qui visent à détecter des attaques à la manière d'un système de détection d'intrusion. Elles peuvent par exemple rechercher des identifiants anormaux de stations de base, et certaines peuvent même observer tout le trafic entre le mobile et la station de base [1]. Borgaonkar et al. ont montré dans [BMP+] que ces applications peuvent facilement être bernées par un IMSI-catcher spécifiquement conçu pour échapper à la détection.

Enfin, des entreprises commercialisent des téléphones sécurisés dont la principale fonction est d'ajouter une couche de chiffrement sur tout le trafic réseau. Par exemple, en forçant toutes les connexions à passer par un VPN. Ce type de solution ne permet des communications sécurisées qu'avec des appareils de la même marque, car les sécurités ajoutées ne sont pas standardisées.

Finalement, la téléphonie mobile donne un exemple bien déplaisant en terme de sécurité. Les protocoles offrent peu de place à l'amélioration continue et des vulnérabilités introduites il y a près de trente ans continuent d'être exploitées silencieusement.

Les contremesures prennent la forme de rustines qu'on applique sur les boîtes noires que sont les *baseband* des téléphones mobiles.

3.1.2 Exemple du Bluetooth

Le Bluetooth est le standard le plus répandu pour les communication sans fil à courte portée. Il est en particulier très largement utilisé pour connecter les ordinateurs et les smartphones à divers accessoires et objets communicants. Les différentes versions de ce protocole définissent des fonctions d'authentification et de chiffrement.

Dans beaucoup de produits, le Bluetooth vient simplement remplacer une connexion auparavant filaire. Ces produits possèdent rarement une puissance de calcul suffisante pour ajouter une couche de chiffrement supplémentaire, et se reposent donc sur la sécurité intrinsèque au Bluetooth. Malheureusement, les protocoles Bluetooth souffrent de plusieurs faiblesses qui peuvent totalement mettre à mal les propriétés de sécurité que le protocole est censé assurer.

On peut citer plusieurs exemples sur le Bluetooth Classic et le Bluetooth Low Energy.

Bluetoooth Low Energy. Ryan [Rya13] a montré une attaque sur le chiffrement du Bluetooth Low Energy (BLE), qui cible le processus d'appairage. L'appairage BLE se repose sur une clé temporaire (TK). Cette clé peut être définie de trois façons différentes suivant le mode d'appairage choisi : on peut utiliser le mode *Just Work* dans lequel TK est fixé à zéro, un *pin à 6 chiffres*, ou encore une clé prépartagée *out-of-band* (OOB) de 128 bit. Ryan indique que l'écrasante majorité des applications se reposent sur l'une des deux premières méthodes, car le mode OOB est nettement moins pratique à utiliser.

1. Cela ne fonctionne qu'avec certaines marques de téléphone

Un attaquant passif peut écouter le processus d'appairage et - s'il n'utilise pas le mode OOB - la valeur de TK peut être retrouvée par une recherche exhaustive sur un espace d'un maximum d'un million de clés (pour un pin à 6 chiffres). Il est alors possible de dériver la clé à long-terme (LTK), qui sera réutilisée pour toutes les connexions suivantes. On pourrait croire que cette attaque est circonscrite au moment précis de l'appairage qui en principe ne se produit qu'une seule fois, mais Ryan précise que des attaques triviales permettent de désappairer des équipements Bluetooth, afin de forcer une nouvelle procédure d'appairage qui pourra alors être écoutée par l'attaquant. [2]

Bluetoooth Classic. En 2005, Shaked et Wool [SW05] ont montré que la sécurité de l'appairage (*Legacy Pairing*) reposait uniquement sur le code PIN. Après avoir observé passivent l'appairage, une attaque bruteforce permettait de récupérer quasiment instantanément la clé servant à chiffrer les communications. L'ajout du *Simple Secure Pairing* dans la spécification Bluetooth a corrigé cette vulnérabilité dans les générations suivantes d'appareils.

Antonioli et al. ont récemment présenté plusieurs attaques sur l'authentification et le chiffrement du Bluetooth Classic, antérieur au BLE. La première, qu'ils ont surnommée *Key Negociation Of Bluetooth* (KNOB) permet l'interception et la manipulation des échanges dans un scénario d'homme-du-milieu, contournant ainsi toutes les mesures de sécurité du Bluetooth Classic : authenticité, confidentialité et intégrité[ATR19]. L'attaque repose sur la procédure de négociation de la taille de clé. Elle permet lors de l'établissement de la connexion de choisir une taille de clé comprise entre un et seize octets. Un homme-du-milieu peut donc forcer la négociation d'une clé d'un octet seulement, qu'il lui est ensuite facile de retrouver par une recherche exhaustive. Suite à cette publication, la Bluetooth SIG a finalement publié un errata à la spécification Bluetooth [Blu19].

Les mêmes auteurs [ATR20] ont présenté un an plus tard une seconde attaque - *Bluetooth Impersonation AttackS* (BIAS) - qui permet cette fois d'usurper l'identité d'un appareil Bluetooth sans qu'il ne soit nécessaire d'observer l'initiation d'une connexion. Elle repose sur un défaut dans la spécification : il existe plusieurs modes d'appairage en Bluetooth Classic, et le *Legacy Pairing* ne requiert pas l'authentification du *maître* par l'*esclave*. Après un appairage, les appareils Bluetooth ne gardent pas en mémoire la méthode d'appairage qui avait été utilisée. Un attaquant peut donc réaliser un nouvel appairage en se faisant passer pour le maître, et il n'aura pas besoin de prouver son identité auprès de l'esclave. De plus, le Bluetooth Classic permet d'échanger les rôles du maître et de l'esclave via la procédure de *Role Switch*. Un attaquant qui souhaite usurper l'identité d'un esclave peut donc utiliser le *Role Switch* pour réaliser l'appairage en tant que maître, et exécuter la même attaque.

Il peut sembler étonnant que des attaques telles que KNOB et BIAS n'aient été dé-

2. En pratique, pour pouvoir écouter toutes les connexions, il faut disposer non seulement de LTK mais aussi du *nonce* (nombre aléatoire à usage unique) échangé en clair lors de l'initiation de la connexion. Il est possible de suivre une connexion courante et de brouiller les échanges pour forcer la réinitialisation de la connexion et récupérer le *nonce*.

montrées que si tard, près de 20 ans après la publication de la spécification vulnérable. Le risque posé par la procédure de négociation de la taille de clé était déjà connu depuis les années 2000 [Mav05]. Néanmoins il a fallu attendre 2019 et la publication de « KNOB attack » pour que le Bluetooth SIG se résolve à amender la spécification. Il semble que ce soit l'existence d'une preuve de concept — rendue possible par le projet InternalBlue publié en 2019 [MCSH19] — qui ait suscité cette réaction.

Cela montre comment l'existence d'outils modifie la perception des risques de sécurité : des failles clairement identifiées subsistent dans le standard jusqu'à ce qu'elles fassent l'objet d'une démonstration pratique.

3.2 L'opacité des modems

Les composants de communication sont souvent opaques. Les documentations détaillées ne sont pas rendues publiques, et tenter de comprendre l'archicture d'un modem est parfois un défi. Cela crée plusieurs obstacles à l'étude de leur sécurité. Il est plus difficile de rechercher des vulnérabilités dans un système mal compris, et le manque d'outils de tests décourage parfois les recherches.

3.2.1 La sécurité par l'obstacle matériel

Le principe de Kerckhoffs, énoncé en 1883, stipule que pour qu'un cryptosystème soit fiable, sa sécurité ne doit reposer que sur sa clé, et sur aucun autre secret. Par exemple, on ne devait pas pouvoir décrypter des messages chiffrés simplement parce qu'on avait mis la main sur la machine ayant servi à les chiffrer.

Ce principe fût, par la suite, généralisé par Claude Shannon dans sa fameuse maxime : *« l'adversaire connaît le système »*. Ce concept s'oppose à la *sécurité par l'obscurité*, qui vise à protéger un système en essayant de masquer ses principes de fonctionnement. La sécurité par l'obscurité est souvent illusoire, et ne peut être envisagée que comme une dernière ligne de défense.

La maxime de Shannon peut être appliquée à nos protocoles de communication : on ne doit pas pouvoir attenter à la sécurité des communications d'autrui simplement parce qu'on comprend leur protocole. Malheureusement, il semble que certains organismes de spécification se reposent sur une subtile variante de la sécurité par l'obscurité, qu'on pourrait appeler *la sécurité par l'obstacle matériel*. Bien que le protocole soit connu, il est difficile à implémenter, et on considère que les attaquants ne disposeront pas du matériel permettant d'exploiter ses faiblesses.

La sécurité par l'obstacle matériel est aussi illusoire que celle offerte par l'obscurité. Si un protocole comporte une vulnérabilité, on doit considérer qu'il est immédiatement exploitable, et que ceux qui veulent en tirer partie pourront se procurer le matériel nécessaire.

La loi de Wright, souvent citée dans le domaine de la sécurité sans fil [3] résume bien le problème : *« Security will not get better until tools for practical exploration of the attack*

3. Attribuée à Joshua Wright, date inconnue

surface are made available ».

3.2.2 Du matériel très soft

Les circuits de communication sans fil sont souvent considérés comme de simples composants matériels. Ils sont pilotés au moyen d'interfaces simples, comme par exemple les commandes Hayes (ou AT), qui permettent de contrôler un modem par des commandes textuelles généralement transmises via un port série. Cela facilite grandement leur intégration, et il est possible de faire évoluer une interface de communication, en remplacant la puce de communication dans différentes itérations du produit, sans nécessiter beaucoup de travail supplémentaire.

Mais pour atteindre une telle simplicité d'intégration, il faut cacher une grande partie de la complexité dans le composant de communication.

On pourrait naturellement imputer la complexité de modems aux tâches très spécifiques qu'ils ont à accomplir, en particulier les différents types de couches physiques qu'ils ont à supporter. Pour supporter la dernière version du Wi-Fi, on devra ainsi forcément utiliser une nouvelle carte qui supporte ses fréquences et modulations.

Mais en réalité, la complexité de ces composants ne s'arrête pas là : une partie des couches logiques s'y trouve déportée. Pour faciliter l'intégration et réduire la consommation d'énergie et la charge sur le processeur de l'hôte, on place les fonctions d'association, d'authentification et de gestion du lien directement dans le composant de communication. C'est par exemple le cas des modems de téléphonie qui réalisent l'intégralité de ces tâches. C'est aussi le cas des variantes du Bluetooth, dans lequel la gestion du lien ainsi qu'une partie des fonctions d'association sont réalisées par le contrôleur. Enfin, c'est désormais le cas des cartes Wi-Fi "FullMAC" (ou "HardMAC"), qui à la différence des cartes "SoftMAC" implémentent désormais la couche *Mac Sublayer Management Entity* (MLME), et présentent à l'hôte une interface ethernet (802.3), masquant ainsi les spécificités du 802.11.

Pour accomplir ces tâches supplémentaires, les composants sans fil ont évolué pour devenir des systèmes-on-chip complets qui intègrent un ou plusieurs processeurs basse-consommation qui réalisent les couches hautes des protocoles.

Du logiciel en quantité. Ces processeurs exécutent des micro-logiciels d'une taille non-négligeable. Par exemple, les *firmwares* Bluetooth que nous avons rencontrés avaient des tailles comprises entre 400 kB et 1 Mb. C'est non négligeable, mais peu en comparaison avec ce qu'on trouve dans les *firmwares* de modems de téléphonie mobiles. Les puces de communication mobiles intègrent le support de toutes les normes de téléphonie mobile, allant de la 2G à la 4G (et bientôt la 5G), qui sont définies par des spécifications luxuriantes. Dans un téléphone moderne, le logiciel embarqué [4] chargé de ces couches protocolaires peut atteindre plusieurs dizaines de méga-octets. À titre d'exemple, sur mon téléphone Android les firmwares chargés du Wi-Fi et de la téléphonie mobile occupent 39 méga-octets, soit plus de deux fois la taille du noyau Linux décompressé.

4. qui s'exécute sur le processeur *baseband*

Cette situation semble partie pour perdurer. Les protocoles de communication évoluent régulièrement, mais maintiennent l'interopérabilité avec les anciennes versions. Il en résulte un accroissement de la quantité des spécifications, et pour faire face à cette complexité, les fabricants ont tendance à concevoir des systèmes qui intègrent toutes ces fonctionnalités directement dans le composant.

Comme on le voit, les chipsets de communication sans fil contiennent en réalité une grande quantité de logiciel. Ces composants sont par nature directement exposés à des connexions distantes. C'est donc une porte d'entrée dans le système, qui doit être bien protégée.

3.2.3 Nous dépendons de code fermé et vulnérable

Les *firmwares* embarqués dans des composants sans fil peuvent contenir des vulnérabilités. Comme vu précedemment, il peut y avoir des vulnérabilités dans la spécification, et elles se retrouvent alors reproduites dans tous les produits qui respectent cette spécification. Mais étant donné le nombre de ligne de codes que contiennent ces *firmwares*, on peut également retrouver des vulnérabilités d'implémentation, comme dans tout type de logiciels.

On constate que peu de vulnérabilités sont identifiées et corrigées, eu égard au large déploiement de ces *firmwares* et de leur importante surface d'attaque. Par exemple, les premières failles documentées dans des *firmwares* GSM datent de 2010 [Wei10], alors que les réseaux GSM étaient déployés depuis près de vingt ans.

Ces *firmwares* sont toujours propriétaires. Il n'est pas possible d'en consulter le code source et encore moins de le modifier. Plusieurs facteurs expliquent cette situation :

— propriété intellectuelle : les constructeurs de téléphonie mobile gardent jalousement leurs secrets industriels. En témoigne la guerre des brevets sans merci qu'ils se livrent [5] ;
— contraintes légales : les appareils utilisant le spectre radio sont soumis à différentes lois selon les pays. Certaines bandes de fréquences ne peuvent être utilisées que par des équipements agréés, auquel cas l'utilisateur final ne doit pas pouvoir modifier l'équipement.

Ce manque d'ouverture pose deux problèmes principaux en termes de sécurité :

— moins de regards extérieurs : la fermeture des *firmwares* rend plus difficile l'analyse indépendante de leur sécurité, car les analystes doivent passer par une coûteuse étape de rétro-ingénierie ;
— moins d'outils : en empêchant les tiers de modifier leurs équipements, on les prive des outils expérimentaux indispensables à la compréhension pratique des protocoles, ce qui entrave notamment la recherche en sécurité.

En conclusion, nous devons encore nous reposer sur du code fermé et très peu audité pour pouvoir communiquer au travers de standards établis il y a plusieurs décennies.

5. Voir aussi le procès Qualcomm vs Apple `https://www.developpez.com/actu/226196/`

3.3 L'angle mort des communications réseau

Dans la section précédente, j'ai montré que l'opacité des *baseband* de communication nuit à leur sécurité intrinsèque. Mais selon moi le problème ne s'arrête pas là : cette opacité rend également beaucoup plus complexe la mise en œuvre de mesures classiques de sécurité - les systèmes de détection d'intrusion (IDS).

3.3.1 Une perte de contrôle sur le code réseau du système ?

Pour réaliser des communications sans fil nous devons utiliser des composants complexes, qui renferment une importante quantité de logiciel. Ces logiciels peuvent souffrir de failles de spécification ou d'implémentation, qui sont trop rarement corrigées.

Une telle situation semblerait peu envisageable dans des systèmes plus ouverts, comme par exemple le noyau Linux. L'histoire du TCP-offload engine l'illustre bien. Au début des années 2000, les constructeurs de cartes réseau ethernet ont commencé à proposer des cartes dotées d'un *TCP-offload engine* (TOE). Le TOE promettait de réduire la charge sur le CPU principal en déportant une partie des couches TCP/IP dans la carte réseau. Selon les implémentations, différentes fonctions peuvent être déportées, comme le calcul des sommes de contrôle et la segmentation des paquets sortants, ou d'autres plus sensibles l'aggrégation des paquets entrants ou même la gestion des connexions TCP.

Les développeurs du noyau Linux ont toujours refusé d'inclure le support complet du TOE [Fou]. Pour motiver leur refus, ils invoquent plusieurs problèmes. Le premier qu'ils citent est l'affaiblissement de la sécurité : les implémentations de TOE peuvent contenir des vulnérabilités, qui ne pourraient être corrigées qu'en mettant à jour le microcode de la carte réseau. Les développeurs Linux considèrent que certaines failles dans le TOE risqueraient de ne jamais recevoir de correctifs, par exemple si le constructeur de la carte réseau n'existe plus. Par ailleurs, le code de la pile TCP/IP Linux est particulièrement mature et constitue l'implémentation de référence dans ce domaine, implémentant toujours les dernières améliorations du protocole. Ses développeurs considèrent que les gains offerts par le TOE restent marginaux, au regard des problèmes qu'il pose.

Les mêmes arguments ont conduit Microsoft à déprécier le support du TOE dans Windows en 2017 [Wil].

Finalement, quand on observe l'architecture d'un téléphone mobile, avec ses multiples processeurs exécutant des *firmwares* de tailles exorbitantes, le cas du TOE semble bien dérisoire.

Il reste un long chemin à parcourir avant que nous puissions maîtriser nos communications sans fil avec la même rigueur que celle exigée pour le TCP/IP.

3.3.2 Les HIDS : une vision partielle

Les *Host Intrusion Detection System* (HIDS) sont des systèmes de détection d'intrusion qu'on place sur l'hôte, à la différence des NIDS (Network IDS) par exemple, qui sont situés au niveau du réseau.

La théorie : les HIDS voient l'état de la machine. Comme ils sont installés directement sur le système à protéger, les HIDS peuvent obtenir une vision précise de l'état de la machine, en observant les processus, leurs interactions avec le système d'exploitation, les fichiers échangés, etc. Ils ont aussi la capacité d'observer le trafic réseau, et peuvent même le surveiller au niveau de la couche applicative, ce qui leur permet de détecter des attaques sur des connexions sécurisées par TLS par exemple.

En théorie, les HIDS pourraient donc constituer des outils idéaux pour détecter des attaques par le réseau, car ils ont accès à toutes les ressources de la machine, et peuvent donc observer toutes les couches des piles réseau.

La pratique : les HIDS sont limités au processeur principal. Les systèmes de communication sans fil posent en réalité des obstacles aux HIDS. En effet, les protocoles de communication sans fil peuvent être en grande partie pris en charge par des micrologiciels propriétaires enfouis dans des processeurs séparés de l'OS prinicipal.

Des HIDS sont disponibles pour tous les systèmes d'exploitation les plus répandus, comme Windows et Linux. En revanche, il serait difficile d'adapter les HIDS aux innombrables équipements réseau disponibles sur le marché. Donc en pratique, les HIDS peuvent voir ce qu'il se passe dans le système d'exploitation, mais pas à l'intérieur des *firmwares* qui gèrent pourtant une partie substantielle des piles réseau.

Les impossibles FIDS (Firmware-HIDS^{TM}). Il reste possible d'appliquer des techniques d'HIDS sur les composants réseau. Duflot a par exemple pû surveiller le contrôle du flot d'exécution dans un *firmware* de NIC Broadcom NetXtreme [DPM11]. Malheureusement cette approche est difficile généralisable, car elle vise un équipement précis et une grande partie du travail serait à refaire si on souhaitait utiliser la même approche pour un NIC d'un autre modèle ou d'une autre marque.

On l'a vu plus haut, les développeurs Linux rechignent à supporter le TOE dans Linux, considérant qu'il serait difficile de supporter à long terme une grande variété d'implémentations TOE, quand bien même les constructeurs mettraient à disposition un driver open source. De ce point de vue, assurer le support d'un HIDS capable d'observer l'état interne de tout un écosystème d'adapteurs réseau semble totalement infaisable.

Plusieurs pistes ont été proposées pour assurer la sécurité de micrologiciels, par exemple Chevalier et al. proposent de vérifier le contrôle de flôt d'exécution (CFI) d'un *firmware* UEFI depuis le CPU principal [CVPH17]. Cependant leur méthode repose sur un *firmware* instrumenté qui doit être compilé depuis les sources.

Un autre obstacle semble écarter définitivement la possibilité d'appliquer les HIDS aux *firmwares* réseau : l'authentification des *firmwares*. Dans de très nombreux cas, les *firmwares* des composants réseau sont protégés par des vérifications d'intégrité et d'authenticité. On ne peut donc pas les modifier sans posséder la clé privée qui sert à les signer. Bien souvent, on ne peut pas non plus accéder à leur mémoire interne.

Une alternative permettant de diagnostiquer l'état d'un *firmware* signé est d'exploiter des vulnérabilités dans les systèmes qui servent à les authentifier, comme l'ont par exemple

fait Berard et Fargues dans [BF]. Cette approche est utile à la recherche de vulnérabilités, mais serait peu appropriée pour un HIDS.

L'horizon des HIDS. Finalement, les HIDS semble condamnés à ignorer le fonctionnement interne des modems. Leur vision du système est donc limitée aux frontières de l'OS. Par exemple pour le Bluetooth, l'horizon de l'HIDS se situe au nouveau de la couche HCI, ou en dernier ressort à la couche intermédiaire (par exemple USB ou UART) qui transporte l'HCI.

Cet horizon pose un gros problème pour les attaques qui pourraient se situer intégralement à l'extérieur de cet horizon. Par exemple, lorsqu'un composant PCI est contrôlé par un intrus, il pourrait communiquer avec d'autres périphériques sur le bus PCI sans passer par l'OS [DPVL10]. Néanmoins, les interfaces situées à la frontière entre les modems et le système d'exploitation constituent une importante source d'information, facilement exploitable par les HIDS qui ciblent les principaux systèmes d'exploitation.

Les HIDS dans l'IOT : une impasse ? L'apparation d'objets communicants, qui échangent des données souvent sans fil, mais qui ne sont pas des ordinateurs à proprement parler posent un obstacle aux HIDS. Par exemple, il n'existe pas encore d'antivirus pour ampoule connectée. En fait, les *firmwares* d'objets connectés posent les mêmes problèmes aux HIDS que ceux modems : ce sont des systèmes fermés, basés sur une grande variété d'architectures, et souvent protégés contre l'introduction de logiciels tiers.

Pour protéger ce type d'objets, il semble donc plus commode de surveiller directement les communications au niveau du réseau, soit au niveau de la passerelle - lorsqu'elle existe - soit directement au niveau de la couche physique.

3.3.3 Les WIDS c'est pour le Wi-Fi.

Les WIDS (Wireless-IDS) sont des IDS qui observent les communications sans fil. Ce terme prête à confusion, car les WIDS ne s'intéressent qu'au WLAN, et plus particulièrement au protocole Wi-FI.

Le Wi-FI constitue un cas d'étude intéressant pour la détection d'intrusion sur le médium sans fil. Il est relativement simple de capturer les trames qui sont échangées sur un réseau Wi-FI en utilisant un adaptateur Wi-FI en mode moniteur. Cela ne permet pas d'accéder à tout le contenu des échanges — il peuvent être chiffrés — mais suffit à observer les procédures d'association et de contrôle du lien qui peuvent être la cible d'attaques. À partir de là, il est possible d'élaborer des règles de détection basées sur des signatures d'attaques connues, ou sur la recherche d'anomalies par rapport à un modèle prédéfini.

Plusieurs facteurs peuvent expliquer que le Wi-FI concentre la majeure partie des IDS sans fil : historiquement, ce standard a été conçu pour remplacer des réseaux ethernet. Il est donc très largement répandu, et véhicule souvent des informations sensibles. Par ailleurs, ce standard offre une connexion au réseau dans de nombreuses organisations. Il est donc installés sur des sites fixes, et il est donc possible en pratique de déployer un WIDS dans

un bâtiment pour protéger ses différents points d'accès sans fil. Les WIDS sont donc moins bien adaptés à des protocoles que l'on utilise en se déplacant, comme la téléphonie mobile. En effet, on imagine mal voyager avec un WIDS sur soi. Néanmoins, les IDS placés au niveau du réseau sont souvent la seule option disponible lorsqu'on ne maîtrise ni l'objet à protéger ni les équipements réseau.

Les réseaux sans fil sont appelés à se diversifier, avec des évolutions comme la 5G qui promet d'offrir une connectivité à des milliards d'objets communiquants. L'étude de WIDS adaptables à différents types de couches physiques reste donc un sujet important à l'heure actuelle.

3.4 Pistes étudiées

Cette section résume les principales options qui peuvent répondre aux problèmes que nous soulevons dans ce chapitre. D'abord, nous reviendrons sur les opportunités disponibles pour améliorer la sécurité des communications sans fil en général. Dans un second temps, nous détaillerons les pistes que nous avons choisi d'approfondir au cours de cette thèse.

3.4.1 Opportunités de sécurisation

Revenons plus en détail sur les quelques principales options à notre disposition pour améliorer la sécurité de nos communications sans fil.

Protocoles. L'un des principaux leviers pour communiquer de façon sécurisée sans fil est le choix du protocole qu'on utilise. Comme on l'a vu plus haut, plusieurs facteurs doivent être pris en compte :

- **quelles sont les propriétés de sécurité assurées par le protocole ?** Il est important de connaître les propriétés de sécurité promises par le protocole : les connexions sont-elles authentifiées ? Les échanges sont-ils chiffrés correctement ? Il faut aussi distinguer les propriétés réellement obtenues et celles qui peuvent être contournées à cause d'erreurs dans le standard ;
- **est-ce que ce protocole est maintenu à jour ?** Certains protocoles sont rapidement mis-à-jour suite à l'identification de faiblesses, tandis que d'autres restent vulnérables pendant des années voire des décennies. De la même façon, selon le choix du protocole, des correctifs peuvent être appliqués par des mises-à-jour logicielles, ou peuvent imposer un renouvellement du matériel ;
- **ce protocole est-il sujet à des failles d'implémentations ?** Tous les protocoles ne sont pas aussi simples à implémenter, et certains sont plus périlleux que d'autres, notamment lorsqu'il requièrent d'imposants micrologiciels qui offrent une riche surface d'attaque souvent exempts d'atténuations modernes contre les attaques logicielles.

Détection d'intrusion. Les systèmes de détection d'intrusion (IDS) sont des outils utiles pour pallier les faiblesses connues, ou même pour déjouer des attaques visant des vulnérabilités non publiques.

Les IDS placés sur l'hôte (HIDS), disposent d'une vue détaillée de l'état du système. Ils peuvent ainsi diagnostiquer les paquets réseau entrants et sortants, mais aussi observer les processus en cours et leurs interactions avec le système d'exploitation. Cela peut non seulement leur permettre de détecter des attaques au niveau du réseau, mais également des actions malicieuses qui peuvent survenir après une première intrusion.

Ils sont pratiques à déployer sur des parcs de machines comme des ordinateurs ou des serveurs, comme en témoigne le marché florissant des antivirus.

Cependant, ils sont nécessairement fortement couplés au système d'exploitation, et il est donc difficile d'intégrer un HIDS dans des systèmes contraints ou fermés comme par exemple les objets communiquants. Enfin, leurs capacités d'observation du réseau se limite aux données visibles par le système d'exploitation : ils ne peuvent par exemple pas voir les couches réseau qui ne sont traitées par le « matériel » des interfaces réseau.

Les IDS placés sur le réseau (NIDS) ou au niveau des échanges sans fil (WIDS) ont l'avantage d'être entièrement découplés des hôtes. Ils peuvent donc facilement être mis en œuvre dans des architectures hétérogènes, et détecter des anomalies dans des échanges entre des systèmes fermés. En revanche, ils sont surtout efficaces pour observer les couches basses des protocoles, notamment car une grande partie des données sont chiffrées au niveau de la couche applicative. Ils peuvent néanmoins fournir de précieux indicateurs sur les couches supérieures, en analysant par exemple les volumes de données transmises sur chaque port.

Les *firmwares* alternatifs. Une piste peu explorée pour conjuguer communications sans fil et sécurité est d'ouvrir les couches de communication. Comme on l'a vu dans ce chapitre, ce sont les couches les plus basses qui souffrent de leur fermeture. Les composants qui régissent les communications sans fil reposent sur des micrologiciels fermés qui peuvent masquer une partie des communications effectives au système d'exploitation, et qu'il est très difficile de protéger par des HIDS. Ces micrologiciels sont spécifiques à chaque fabricant, et on est obligé de faire confiance à ces fabricants pour veiller à la sécurité de leurs logiciels et mettre à disposition des correctifs.

En remplaçant ces micrologiciels par des alternatives *Open Source*, dans lesquels il est plus simple d'identifier et de corriger les vulnérabilités, on peut envisager d'offrir un support extérieur à long terme dans nos équipements réseau. Cela permet de rétablir la confiance dans ces équipements, et les rend nettement plus simples à interfacer avec des IDS.

Architectures. La dernière option, peut être la plus efficace mais la plus complexe à mettre en œuvre, est d'adapter les architectures de nos systèmes, afin de pouvoir utiliser en toute sécurité des composants que l'on suppose à priori vulnérables.

Par exemple, admettons que l'on souhaite communiquer en utilisant le protocole Bluetooth. Dans le pire des cas, on peut supposer que les mécanismes d'authentification et de chiffrement offerts par ce protocole ne sont pas effectifs. On peut également considérer que

le composant est compromis, ou que son micrologiciel est malveillant.

On peut résoudre les problèmes d'authentification et de chiffrement en implémentant ces fonctions au niveau de la couche applicative, par exemple TLS. En revanche, cela requiert un surcroît de mémoire et de puissance de calcul qui ne sont pas forcément disponibles dans de petits objets connectés.

On peut également composer avec un composant compromis en veillant à l'isoler du système et des autres périphériques par exemple à l'aide d'une IOMMU [6], et en appliquant des bonnes pratiques en programmant son pilote : *never trust user input.*

3.4.2 Prendre le problème par en-dessous : IDS sur couche physique

Dans la première partie de cette thèse, nous avonc travaillé sur une approche d'IDS sans fil multi-protocoles. Dans un premier temps, nous avons réalisé un outil permettant la capture et l'analyse de signaux de différents protocole intervenant dans une même bande de fréquence. Dans un second temps, nous avons utilisé les données obtenues à l'aide de cet outil pour réaliser du *fingerprinting* sur la couche physique du Bluetooth Classic, afin d'identifier les appareils alentours et de quantifier leurs échanges.

IdO, *baseband*, des problèmes similaires. Dans le contexte de l'*Internet des Objets* (IdO), les appareils que l'on souhaite protéger disposent généralement d'une faible capacité de calcul, et se présentent le plus souvent comme des boîtes noires, basées sur des logiciels fermés. Il est donc difficile de leur adjoindre des HIDS, qui nécessitent de pouvoir installer des logiciels dans le système ciblé. On remarque que ces obstacles aux HIDS ne sont pas seulement rencontrés dans le contexte d'objets connectés. En effet, les modems que l'on trouve dans nos ordinateurs et ordiphones posent des contraintes assez similaires aux HIDS. Il est difficile d'instrumenter et donc de surveiller l'état de ces composants qui utilisent des logiciels et du matériel fermés, et ne disposent pas d'une grande marge de manœuvre en terme de mémoire et de capacité de calcul.

C'est la raison pour laquelle nous avons préféré étudier des méthodes de détection d'intrusions basées sur le réseau, qui sont par nature découplées des objets ciblés, et donc plus à même d'être adaptées à une grande variété de systèmes contraints et fermés.

Où positionner l'IDS ? Les réseaux sans fil peuvent se présenter sous différentes topologies, allant du simple réseau en étoile jusqu'aux réseaux ad hoc maillés. Certains sont presque directement connectés à internet, d'autres s'y connectent via une passerelle dédiée. Enfin, certains réseaux sans fil ont une portée uniquement locale (WPAN).

Comme nous l'avons expliqué dans le paragraphe précédent, nous avons centré nos recherches sur les techniques de détection d'intrusion basées sur le réseau, car ces méthodes sont fortement découplées des objets qu'on souhaite protéger, et ne sont pas liées à un logiciel en particulier.

6. Une IOMMU permet de contrôler les accès d'un périphérique à la mémoire centrale.

De même, nous souhaitions étudier des méthodes d'IDS qui ne soient pas couplées à un matériel en particulier, ou à un protocole unique. En effet, les techniques de communication sans fil sont en constante évolution, et nous recherchons des méthodes qui pourront être utiles aussi bien sur les technologies actuelles que celles qui emergeront dans le futur.

La couche physique : le dénominateur commun. Les réseaux sans fil posent diverses difficultés aux systèmes de détection d'intrusion. L'une des principales difficultés que nous avons identifiée provient de la grande diversité des protocoles en jeu : tous les protocoles n'offrent pas la même quantité d'information lorsqu'on les observe passivement. Dans certains standards, l'essentiel des communications est chiffré et est donc difficilement interprétable par un IDS, par exemple pour rechercher des signatures d'attaque.

De même, lorsque les messages d'un protocole ne véhiculent pas d'identifiants, on ne peut pas facilement observer les flux échangés entre les appareils. Identifier les participants à un réseau sans fil, et quantifier leurs échanges de données est pourtant une fonctionnalité précieuse pour un système de détection d'intrusion, permettant notamment de réaliser des modèles du trafic, et d'appliquer ensuite des techniques de recherche d'anomalies.

Lorsqu'on recherche le plus grand dénominateur commun de tous les réseaux sans fil, on arrive naturellement à la couche physique : les ondes électromagnétiques. Dans ce domaine, un champ a particulièrement retenu notre attention, c'est celui du *fingerprinting* sur la couche physique. Ces méthodes visent à identifier des appareils en fonction de signatures basées sur les subtiles variations qui existent dans les composants qui réalisent leurs communications sans fil.

Des méthodes de *fingerprinting* sur la couche physique ont été appliquées avec succès à une grande variété de standards de communication différents. On peut donc s'attendre à ce qu'elles continueront d'être applicables sur de futurs standards. Le *fingerprinting* pourrait donc être une option intéressante pour pouvoir à minima identifier les appareils, puis quantifier les flux de données qu'ils transmettent. En cela, le *fingerprinting* pourrait constituer un précieux outil pour obtenir une représentation du trafic échangé sans fil, et ainsi créer la base d'un outil de détection d'intrusion adaptable à tous types de protocoles de communication existants ou futurs.

C'est pour ces raisons que nous avons étudié des approches de *fingerprinting* sur plusieurs protocoles sans fil. Nous avons d'abord réalisé un outil qui permet de capturer et de traiter les transmissions de plusieurs protocoles largement utilisés actuellement. Nous avons ensuite utilisé les données obtenues à l'aide de cet outil pour découvrir de nouveaux indicateurs permettant d'identifier des appareils Bluetooth selon le modèle de leur puce Bluetooth.

Software Defined Radio : mise en pratique Les radio-logicielles constituent selon nous une piste intéressante pour la sécurité des réseaux sans fil, car elles permettent d'observer les transmissions radio à différents niveaux.

D'une part, elles permettent d'implémenter des receveurs pour tous types de protocoles sans fil, comme par exemple le Wi-Fi 802.11a/g/p [BSSD13] ou la téléphonie 4G [NKK+14,

GMGSS+16]. On peut donc les utiliser pour décoder des échanges radio et y rechercher les signatures d'attaques connues.

D'autre part, elles peuvent surveiller de larges bandes de fréquence dans lesquelles coexistent de multiples protocoles, et ainsi fournir une vision plus globale de l'environnement radio. C'est l'approche poursuivie par les IDS RadIoT [RAA+18], et SAIFE [RMLP19].

Enfin, elles rendent possible des méthodes de *fingerprinting* sur la couche physique, qui répondent en partie au problème de l'identification des transmetteurs même lorsque leurs paquets ne peuvent pas être entièrement décodés. Cela a par exemple été montré pour le GSM [RTM10, ZJZ+18], Bluetooth [HBK06, USC12], ou le Wi-Fi [VHVHN16].

Même si elles reposent sur un matériel similaire, ces approches mettent en oeuvre des radio-logicielles bien différentes. Les receveurs dédiés présentés dans [BSSD13, NKK+14, GMGSS+16] peuvent décoder les échanges d'un seul protocole : celui pour lequel ils ont été programmés. Ils sont donc difficilement applicables tels quels à la détection d'intrusion multi-protocoles.

À l'inverse, les IDS présentés dans [RAA+18] ou [RMLP19] sont capables d'observer de larges bandes de fréquence et de multiples protocoles mais sans décoder le trafic. Ils reposent sur la seule observation d'un spectrogramme. Ils semblent donc plus appropriés pour détecter des anomalies ne respectant pas la sémantique d'un protocole que pour détecter l'exploitation de vulnérabilités dans l'implémentation de ce protocole.

Enfin, les techniques de *fingerprinting* nécessitent des signaux démodulés, souvent avec une très haute fréquence d'échantillonnage, qui sont généralement collectés au moyen d'équipements très performants et coûteux, différents du matériel SDR conventionnel. Cependant, [ZJZ+18] a montré que ses techniques de fingerprinting basées sur la modulation, d'abord évaluées avec un analyseur de spectre Keysight N9020B (qui coûte environ 100k$), pouvaient être reproduites avec un USRP B210 nettement plus abordable (moins de 2000$) et couramment utilisé dans la recherche.

Selon nous, ces différentes approches sont complémentaires. Leur association pourrait permettre de détecter plus finement certaines attaques. Par exemple, plaçons nous dans un environnement professionnel, où cohabitent des appareils **résidents** autorisés à accéder à toutes les ressources du réseau et des appareils **invités** dotés de permissions réduites. Admettons aussi que les identifiants logiques associés aux transmissions peuvent facilement être usurpés. Un IDS capable de décoder les échanges radio et d'identifier les transmetteurs par *fingerprinting* permettrait de vérifier que les appareils invités ne tentent pas d'accèder aux ressources réservées aux résidents, avec plus de fiabilité que ne pourrait le faire un IDS reposant uniquement sur des identifiants logiques. Le Bluetooth offre un bon exemple de cette situation (c'est pour cela nous avons basé nos expériences dessus). Un réseau Bluetooth change de fréquence 1600 fois par seconde. On ne peut donc pas le surveiller en observant un seul canal et il faut décoder les paquets pour comprendre à quel réseau ils sont destinés. On peut en revanche comprendre quelles procédures sont en cours en décodant ne serait-ce qu'une partie du trafic. Dans notre exemple, un IDS capable d'observer une partie de la bande Bluetooth pourrait donc distinguer un invité légitime (par exemple équipé d'oreillettes Bluetooth), d'un invité malicieux tentant d'établir une connection interdite vers un appareil résident.

Le Bluetooth : un cas d'école et bien plus encore Dans ce document nous parlerons beaucoup de Bluetooth, et en particulier du Bluetooth 2.0 [7].

Même si le Bluetooth n'est pas le principal objet de cette thèse, nous l'avons beaucoup utilisé dans nos expériences.

La première raison qui nous a conduit à ce choix est que le Bluetooth Classic fait se rejoindre nos deux principaux centres d'intérêt : le sujet des objets connectés, et celui des micrologiciels des modems sans fil.

Le contexte actuel voit apparaitre une multitude d'objets communicants — souvent sans fil — qui posent de nouvelles questions en terme de sécurité. Le Bluetooth est l'un des premiers protocoles à avoir été largement adopté pour ces objets communicants, notamment car il a rapidement été supporté par la majorité des téléphones mobiles pour leur associer des périphériques. Le Bluetooth Classic est par ailleurs réputé complexe à implémenter, et pose des défis intéressants aux systèmes de détection d'intrusion, car il est difficile en pratique d'observer passivement la totalité des échanges entre deux appareils Bluetooth, en particulier à cause des sauts de fréquences qui sont complexes à prédire. Nous supposons donc qu'une méthode efficace pour le Bluetooth pourra être adaptée sans mal à d'autres protocoles plus simples. En cela, le Bluetooth Classic constitue à la fois un sujet d'expérimentation complexe sur l'étude de la détection d'intrusion sans fil, mais est également un cas réel déjà largement appliqué, qui pose des problématiques de sécurité concrètes, notamment pour nos ordiphones et nos PC portables.

La seconde raison est que le protocole Bluetooth comporte des couches logiques qui sont entièrement prises en charges par les contrôleurs Bluetooth. Ces contrôleurs Bluetooth sont des composants fermés, qui contiennent des micrologiciels complexes. Comme nous l'avons discuté dans ce chapitre, ces micrologiciels contenus dans nos interfaces de communication peuvent contenir des vulnérabilités, et leur sécurité mérite donc la plus grande attention.

Pour résumer, on peut considérer les contrôleurs Bluetooth comme des objets connectés à part entière [8]. Ils contiennent des logiciels fermés et propriétaires avec lesquels il est difficile de s'interfacer, que ce soit pour analyser leur sécurité, ou pour leur adjoindre des systèmes de détection d'intrusion. Enfin, il s'agit également de systèmes contraints, qui contiennent rarement des contremesures efficaces contre l'exploitation de vulnérabilités logiques.

De notre point de vue, l'étude des contrôleurs Bluetooth est donc un point d'entrée idéal et fascinant dans le monde des objets fermés communicants.

3.4.3 Les micrologiciel

Dans la seconde partie de cette thèse, nous avons tiré partie de l'expérience acquise sur le protocole Bluetooth pour travailler plus en détail sur la sécurité des composants Bluetooth. Durant cette période, nous avons analysé en détail les micrologiciels de plusieurs puces Bluetooth de fabricants différents. Nous avons pû identifier plusieurs vulnérabilités

7. Bluetooth Classic, ou Bluetooth BR/EDR
8. Voir aussi : les SoC Nordic pour communiquer en Bluetooth et piloter un objet connecté.

critiques dans l'implémentation des couches logiques, permettant la prise de contrôle à distance de ces composants.

Nous avons également développé le premier micrologiciel Bluetooth Classic open-source, qui permet de réaliser (en grande partie) des communications sur toutes les couches du Bluetooth Classic sans avoir à se reposer sur aucun logiciel ni matériel propriétaire. Nous avons prouvé que cet outil peut bénéficier à la recherche en sécurité sur le Bluetooth, en l'utilisant pour exploiter les vulnérabilités précedemment découvertes.

Encore du Bluetooth Durant nos travaux sur la capture et le *fingerprinting* de transmissions sans fil, nous avions choisi le Bluetooth Classic comme sujet d'étude.

Au cours de nos expériences, nous nous sommes heurtés à la nature fermée des contrôleurs Bluetooth : on ne peut ni les forcer à transmettre des paquets arbitraires sur une fréquence donnée, ni même observer tous les paquets qu'ils recoivent ou transmettre au cours d'une connexion normale. Cela rendait par exemple difficile la création de jeux de données correctement annotés. Pour les obtenir, il aurait fallu mettre en œuvre du matériel coûteux, ce qui aurait nuit à la reproductibilité de nos expériences.

En fait, l'idéal pour poursuivre nos expériences eut été de disposer d'un contrôleur Bluetooth entièrement transparent. En plus de faciliter l'obtention de jeux de données, cela nous aurait fourni une vision pratique du protocole, dont la spécification reste complexe à appréhender en l'absence d'implémentation de référence. Nous nous sommes donc penchés plus en détail sur les *firmwares* des contrôleurs Bluetooth, en commencant par reproduire des travaux existants.

Analyse de *baseband* Bluetooth

Le projet InternalBlue [MCSH19], développé par l'équipe allemande Seemo-lab, est le seul — à notre connaissance — à lever un peu le voile sur le fonctionnement interne des contrôleurs Bluetooth. Basé sur la rétro-ingénierie de *firmwares* de contrôleurs Bluetooth Broadcom et Cypress, il permet d'y ajouter quelques fonctionnalités utiles, comme l'interception et l'injection de paquets sur les couches basses du Bluetooth.

Cela nous éclaire un peu mieux sur le fonctionnement des appareils Bluetooth Classic, en permettant notamment d'observer les échanges sur la couche LMP, mais n'offre pas assez d'information sur le comportement de la couche physique pour, par exemple, créer des jeux de données pour le *fingerprinting*.

En revanche, un autre sujet nous a rapidement intrigué. Rappelons qu'en 2017, la publication des attaques « Blueborne », visant des piles Bluetooth côté hôte, avait manifestement suscité l'intérêt des chercheurs de vulnérabilités, et de nombreuses autres failles avaient été corrigées dans la foulée. Les auteurs d'InternalBlue avaient identifié une faille exploitable dans la couche LMP des *firmwares* Broadcom qu'ils étudiaient, mais plus d'un an après leur découverte, aucune autre vulnérabilité n'avait été signalée.

Nos travaux précédents tenaient justement compte des risques posés par les modems sans fil, en particulier Bluetooth. Cette première faille LMP posait plusieurs questions :

— une seule vulnérabilité LMP avait été publiée. Est-ce parce que ce type de failles est très rare, ou simplement car ces *firmwares* sont trop peu audités ?
— la vulnérabilité corrigée n'offrait pas un contrôle total à l'attaquant. Peut-on obtenir l'exécution de code entièrement arbitraire en exploitant une faille au niveau de la couche LMP ?

Pour répondre à ces questions, nous avons passé en revue l'implémentation de la couche LMP dans plusieurs contrôleurs Bluetooth de constructeurs différents, afin d'évaluer leur sécurité et de vérifier s'il était réellement possible d'en prendre le contrôle à distance.

Baseband Bluetooth Open Source

Lors de notre campagne d'évaluation de *firmwares* Bluetooth, nous avions constaté qu'en réalité les vulnérabilités au niveau de la couche LMP sont assez courantes, et qu'elles peuvent substiter pendant de longues années sans être signalées. Elles auraient pourtant pû être découvertes plus tôt si ces *firmwares* avaient été mieux audités, que ce soit par des chercheurs extérieurs, ou par les constructeurs eux-mêmes.

Le principal frein à la recherche indépendante sur la sécurité du Bluetooth Classic est l'absence d'outils accessibles pour déboguer ce protocole. Nous avons donc cherché un moyen de combler ce fossé, afin de pérériser les résultats de nos recherches en réduisant considérablement la durée nécessaire à l'identification et la correction des vulnérabilités présentes dans les contrôleurs Bluetooth.

De notre point de vue, la premier jalon permettant le débogage d'un protocole est une implémentation ouverte de ce protocole. Cela permet à ses utilisateurs d'obtenir une vision pratique et transparente du protocole, qu'ils peuvent modifier selon leurs besoins. Enfin, cela fournit une base expérimentale pour faciliter la création d'autres types d'outils, comme par exemple le système de *fingerprinting* Bluetooth sur lequels nous avions précedemment travaillé.

Au cours de nos expériences précédentes, nous avions déjà utilisé l'Ubertooth One. Il s'agit d'une carte de développement libre pour l'expérimentation sur des protocoles radio à faible débit dans la bande des 2,4 GHz. L'Ubertooth permet de capturer passivement des paquets Bluetooth Classic, mais son *firmware* ne permet pas d'en transmettre. En pratique nous ne pouvions donc pas l'utiliser pour implémenter des expériences *actives* impliquant de pouvoir envoyer et recevoir des paquets.

Étant déjà familiarisés avec l'Ubertooth One, nous pensions pouvoir implémenter la transmission et la réception de paquets en temps réel et ainsi supporter une grande partie de la couche *baseband* Bluetooth Classic. Nous avons donc entrepris la création d'un nouveau *firmware*, en nous fixant comme premier objectif de pouvoir au moins reproduire nos attaques sur la couche LMP avec un matériel et un logiciel entièrement maîtrisés. Cela nous a ensuite permis de développer une méthode simple — mais à notre connaissance jamais publiée auparavant — pour *sniffer* les connexions Bluetooth établies vers un appareil déterminé.

Fournir des outils Durant cette thèse nous avons conçu plusieurs outils pour servir à nos travaux sur la sécurité sans fil, qui sont distribués librement. Cela représente un temps de travail conséquent, et un aspect important de notre contribution.

En effet il nous tient à cœur que nos outils puissent être utilisés par d'autres chercheurs. D'abord pour que nos expériences soient reproductibles, ce qui est un élément essentiel de la démarche scientifique. Ensuite, pour permettre de nouveaux travaux qu'il n'était pas possible de mener auparavant.

Enfin, pour que nos outils puissent être accessibles au plus grand nombre, nous les avons basé sur du matériel facile d'accès, en évitant d'utiliser des appareillages coûteux ou soumis à l'acceptation de licenses contraignantes.

3.5 Conclusion

Dans ce chapitre, nous avons présenté notre vision du sujet de la sécurité des communications sans fil.

Certains protocoles que nous utilisons couramment pour nos communications sans fil comportent des défauts qui rendent illusoires les propriétés de sécurité qu'ils proposent.

Suffirait-il de créer de nouveaux protocoles ? Ce n'est pas forcément possible ni suffisant, comme le montre l'exemple de la téléphonie mobile dans laquelle l'apparition de nouveaux standards n'empêche pas l'exploitation des failles du GSM. Le Bluetooth ne fait pas exception, et la norme Bluetooth Low Energy est même plus simple à attaquer que l'ancien Bluetooth Classic.

Peut-on mitiger ces failles en les corrigeant dans les implémentations : les modems ? Il s'avère que c'est difficile. Ces composants sont souvent des boîtes noires, qui contiennent de larges micrologiciels propriétaires dont la documentation n'est pas publique. En fait, ces micrologiciels peuvent même comporter des failles d'implémentation, qui les rendent vulnérables à des attaques spécifiques, qui ne relèvent pas du protocole.

Certaines de ces vulnérabilités dans les standards ou les implémentations peuvent subsister durant de longues périodes sans être identifées ou corrigées. Nous considérons que cela est en partie dû à des obstacles matériels : certains protocoles pourtant très répandus ne disposent d'aucune implémentation libre de référence, ce qui entrave l'expérimentation et décourage la recherche. C'est pour cette raisons que nous avons choisi de privilégier des solutions accessibles, qui pourront être reproduites et poursuivies avec un faible investissement.

Vu les difficultés qu'il y a à corriger les vulnérabilités dans les protocoles, et à valider les implémentations, quelles mesures de sécurité peut-on prendre ? Nous soutenons que les systèmes de détection d'intrusion basés sur l'hôte ne disposent que d'une vision partielle du trafic échangé sans fil, ce qui les rend aveugles à des attaques visant les couches basses des protocoles : celles qui sont gérées par des micrologiciels fermés. Finalement, nous montrons que les communications sans fil continuent de poser des obstacles aux systèmes de détection d'intrusion basés sur le réseau, qui sont difficiles à surmonter avec les approches classiques.

Cela nous conduit à privilégier deux sujets précis, qui de notre point de vue peuvent contribuer à améliorer la sécurité dans le domaine des communications sans fil.

Le premier porte sur les IDS, et en particulier ceux qui utilisent la couche physique comme principale source d'information. Nous considérons que les approches de détection d'intrusion basées sur l'analyse de la couche physique permettent d'aborder de manière unifiée un environnement hétérogène dans lequel des standards apparaissent et évoluent constamment. De plus, la couche physique offre une source d'information supplémentaire, que nous avons par exemple exploité dans nos travaux sur le *fingerprinting* Bluetooth. Ces travaux sont présentés dans les chapitres 4 et 5

Le second sujet est celui de la sécurité des micrologiciels de communication sans fil. Dans le chapitre 6, nous nous sommes penchés en détail sur des implémentations de contrôleurs Bluetooth, dans lesquels nous avons identifié plusieurs vulnérabilités critiques. Enfin, le chapitre 7 propose une implémentation (incomplète mais) libre du Bluetooth Classic qui permet d'expérimenter sur les couches basses du Bluetooth et d'exploiter les vulnérabilités identifiées, afin de faciliter la recherche sur la sécurité de ce protocole et d'envisager des alternatives ouvertes aux implémentations propriétaires.

Chapitre 4

Sonde SDR pour la détection d'intrusion

Les équipements que nous utilisons dans la vie de tous les jours mais aussi sur le lieu de travail communiquent désormais souvent via un ou plusieurs protocoles de communication sans fil. Par exemple, un téléphone récent supporte plusieurs normes de téléphonie mobile, mais aussi de Wi-FI, Bluetooth et NFC. Avec le temps ces protocoles se multiplient, pour s'adapter aux nouveaux besoins.

Les systèmes de détection d'intrusions (IDS) servent à protéger nos équipements, en détectant des attaques et éventuellement en les bloquant [1]. On classe généralement les IDS selon leur type de positionnement, on distingue ainsi les Host-based IDS (HIDS) des Network-based IDS (NIDS), et des Wireless-based IDS (WIDS).

Le terme de WIDS désigne en réalité des IDS qui visent les réseaux WLAN. Pour capter tous les échanges sur ces réseaux sans-fil, les WIDS sont généralement basés sur des cartes Wi-FI configurées en mode moniteur. Ils ne peuvent donc pas observer le trafic échangé via d'autres protocoles sans-fil.

Il existe des concepts d'IDS pour quasiment tous les protocoles sans fil. Par exemple, OConnor [OCo08] présentait déjà en 2008 un IDS pour le Bluetooth, qui observait les échanges à l'aide d'un analyseur de protocole Merlin LeCroy CATC. Mais ces solutions supportent rarement plus d'un protocole, et ne peuvent pas en supporter de nouveaux sans que l'on ne remplace leur matériel.

Les *software-defined radios* (SDR) sont une piste prometteuse pour faire face à la diversité des protocoles sans fil actuels, et répondre à l'apparition de nouveaux standards. Ce sont des radios définies en logiciel, dans lesquelles des tâches habituellement réalisées par du matériel – modulation, démodulation, etc. – sont faites en logiciel. Elles permettent donc de supporter différents protocoles avec un même matériel, et il est possible d'ajouter le support d'un nouveau standard par une simple mise à jour du logiciel. Enfin, les SDR permettent d'appliquer des techniques de *fingerprinting* sur la couche physique, qui ne pourraient pas être exploitées par des recepteurs conventionnels.

1. On parle dans ce cas de systèmes de prévention d'intrusions (IPS)

Nous avons cherché un moyen d'articuler les SDR avec la détection d'intrusion. Notre principal but était de pouvoir travailler avec plusieurs protocoles différents, dans une large bande de fréquence, et de pouvoir réaliser du *fingerprinting* sur la couche physique. Notre solution doit pouvoir fonctionner en continu, et donc traiter les signaux radio en temps réel.

Les SDR produisent d'importants flux de données. Il est donc difficile de les conserver sur une longue période. Nous avons recherché des solutions pour réduire la taille de ces flux, pour ne conserver que des informations utiles et intelligibles, pouvant être traitées et éventuellement sauvegardées par un IDS.

Plusieurs travaux récents s'intéressent à l'application des SDR à la détection d'intrusion multi-protocoles, et au *fingerprinting* sur la couche physique. Mais à notre connaissance, aucun d'entre eux ne permet de réaliser du *fingerprinting* sur la couche physique de plusieurs protocoles différents, sur une large bande de fréquence. Nous avons donc concu une SDR pour capturer les signaux radio environnants dans une bande de fréquence donnée, les traiter rapidement, et conserver des informations utiles, de taille réduite. Ainsi, nous souhaitons créer une sonde radio pouvant être utilisée par un IDS.

Ce chapitre présente *Phyltre*, une SDR pour l'acquisition de données et de métadonnées sur des réseaux sans fil, à des fins de détection d'intrusion et d'expérimentations. La section 4.1 explique nos motivations pour baser une sonde de détection d'intrusion sur une SDR. La section 4.2 propose ensuite une architecture pour collecter efficacement des informations pertinentes. Enfin la section 4.3 détaille l'implémentation de *Phyltre* qui implémente notre architecture.

4.1 Motivations

Nous avons choisi de nous intéresser aux IDS placés au niveau du réseau, c'est à dire qui observent le trafic échangé entre les différents appareils. Il s'agira donc d'écouter directement les transmissions avec un équipement SDR adapté. Nous avons choisi ce type d'IDS pour plusieurs raisons :

— **Simplicité du déploiement** il est possible d'installer un IDS réseau sans avoir à installer de logiciels dans les appareils à protéger. C'est une propriété particulièrement utile, car il n'est pas toujours aisé d'ajouter des logiciels dans les équipements sans-fil, par exemple lorsqu'on a des objets connectés dont on ne maîtrise pas le logiciel. Pour la même raison, il sera plus facile de faire évoluer la solution de détection d'intrusions lorsqu'elle est indépendante des équipements que l'on souhaite protéger. Cela permet de répondre en partie à la grande hétérogénéité des équipements IdO.

— **Complétude** en observant les transmissions au niveau du canal physique, on a en théorie l'opportunité d'observer tous les échanges, ce qui ne serait pas forcément possible sur l'hôte, qui ne verra pas forcément passer tous les messages du protocole. En effet, certaines couches du protocole peuvent être prises en charge par des composants opaques, qui masquent une partie des échanges effectifs au système

d'exploitation.

— **Multi-protocole** Avec un IDS sans fil, on peut supporter plusieurs protocoles, et donc protéger différents types d'équipements. On peut également envisager de détecter des scénarios plus complexes d'attaques impliquant plusieurs protocoles différents. Par exemple lorsqu'un attaquant utilise une cible qui supporte plusieurs protocoles pour pivoter [2] dans un réseau.

4.1.1 *Software Defined Radio*

Les SDR constituent des outils de choix pour l'implantation d'IDS sans fil. Nous avons choisi de les utiliser pour plusieurs raisons :

— **Multi-protocoles** : les SDR peuvent supporter plusieurs protocoles de communication différents. Il est ainsi possible d'ajouter le support d'un nouveau protocole par une simple mise-à-jour logicielle.

— **Transparence** : avec les SDR, on peut utiliser nos propres implémentations des couches physiques des protocoles sans-fil que l'on souhaite surveiller. Ainsi, nul besoin d'utiliser de composants opaques pour capturer les échanges d'un protocole. C'est particulièrement important, car ces composants opaques peuvent faire partie de la surface d'attaque que l'on veut protéger, et il n'est donc pas souhaitable de les incorporer dans l'IDS.

— **Analyses sur la couche physique** : avec des SDR, on a accès à des informations supplémentaires sur la couche physique des échanges, auxquelles on n'aurait pas accès en utilisant des receveurs standards. On peut ainsi envisager de détecter des attaques plus ou moins sophistiquées sur la couche physique, par exemple les techniques de brouillage. On peut également utiliser des méthodes de *fingerprinting* qui permettent d'inférer l'identité d'un transmetteur, même lorsque les messages qu'il transmet ne contiennent pas d'identifiant logique. Cela peut notamment être utilisé pour détecter des attaquants cherchant à usurper l'identité d'un appareil.

4.1.2 Travaux reliés

Les SDR sont déjà largement utilisées pour travailler sur la sécurité des réseaux sans-fil, notamment pour la mise en œuvre d'attaques. Cependant, aucune des solutions que nous avons identifiées ne répondait entièrement à ce que nous attendons d'une couche d'acquisition pour un IDS sans-fil.

GNU Radio

Le project GNU Radio [Blo04] est une boîte à outil pour le développement et l'exécution de SDR ou d'autres applications de traitement de signal. Il est activement développé, et sert de base à une multitude de projets. Il fournit une librairie de blocs de traitement qui peuvent être combinés dans des ordinogrammes (*flowgraphs*) pour réaliser des tâches

2. Pivoter : Utiliser un système compromis pour attaquer d'autres systèmes situés sur le même réseau.

plus complexes. L'outil gnuradio-companion permet la conception de flowgraphs dans une interface graphique intuitive. Nous avons beaucoup utilisé GNU Radio pour nous familiariser avec les SDR, et tester divers projets basés sur ce dernier. Dans un premier temps, nous espérions pouvoir baser notre solution de capture sur GNU Radio, en réutilisant plusieurs blocs déjà écrits par des tiers pour gérer des protocoles bas débit. Cependant, pour obtenir les informations dont nous avions besoin sous une forme unifiée, il nous aurait en réalité fallu largement modifier chacun des blocs que nous utilisions – voire les réécrire complètement – ce qui aurait fortement limité l'intérêt de cette démarche. Par ailleurs, nous souhaitions pouvoir maîtriser finement l'exécution et le format des données au sein de notre application, ce qui aurait en fait été plus difficile à faire en se basant sur GNU Radio (ou son principal concurrent Pothos). Finalement, nous avons donc choisi de réaliser un projet indépendant de GNU Radio, ce qui nous offre une totale maîtrise du code tout en réduisant considérablement la taille des dépendances.

RFDump RFDump [LSSS09] est un receveur SDR multi-protocoles basé sur GNU Radio, créé en 2009 pour des applications de sécurité. Il comprend une couche de détection qui lui permet de détecter la présence d'un paquet et son protocole en amont, avant de procéder aux coûteuses étapes de démodulation, afin de réduire la charge de calcul. RFDump était basé sur une ancienne version de GNU Radio et son code est aujourd'hui introuvable. RFDump est le projet qui se rapproche le plus de nos besoins. Cependant, il n'était conçu que pour gérer un seul canal, et ne permettait donc pas l'écoute de plusieurs canaux simultanément.

Scapy-Radio

Scapy-Radio [PLD14] est un outil de test d'intrusions qui permet d'interagir avec plusieurs protocoles sans fil au travers de la librairie de gestion de paquets **scapy**. Il gère les protocoles Bluetooth LE, 802.15.4, et Z-Wave, en s'interfacant avec des *flowgraphs* GNU Radio. Scapy-radio a servi de base à d'autres projets : *SecBee* [Cog] qui sert à auditer la sécurité de réseaux ZigBee, et Snout qui vise à devenir l'équivalent de nmap pour les réseaux sans-fil des objets communicants.

Notre approche peut être comparée à celle de scapy-radio, car nous cherchons également à simplifier l'utilisation de SDR pour des applications de sécurité. Mais nos objectifs sont assez différents : d'abord car nous n'avons pas besoin de transmettre, mais surtout car nous souhaitons pouvoir capturer simultanément plusieurs canaux et plusieurs protocoles, éventuellement en enregistrant des portions de signal d'intérêt. Scapy-radio ne gère qu'un protocole et un canal à la fois, et ne permet pas de conserver les échantillons de signal correspondants à un paquet.

Universal Radio Hacker

Universal Radio Hacker (URH) [PN18] est une suite d'outil pour simplifier la rétro-ingénierie de protocoles sans-fil et le développement d'attaques. Il se compose de plusieurs

couches pour capturer des signaux, identifier les paquets et leur transmetteur en fonction de la puissance du signal, puis démoduler et décoder les paquets au moyen d'une interface intuitive. URH est conçu pour l'analyse de protocoles sans-fil inconnus. Il permet donc non seulement d'implémenter rapidement des décodeurs spécialisés pour des protocoles non standards, mais aussi de forger des paquets et de les transmettre, notamment pour tester des attaques. URH est conçu pour fonctionner hors-ligne, l'analyse est effectuée après la capture du signal, sur un seul canal.

Pothos et SoapySDR

L'outil Pothos est une alternative à GNU Radio, créée en 2013 par le développeur de gnuradio-companion, qui souhaitait plus de liberté pour modifier le coeur de GNU Radio. Pothos est donc assez semblable à GNU Radio dans ses fonctionnalités, mais constitue un projet indépendant, qui offre notamment une plus grande flexibilité dans la gestion de la mémoire et des files d'exécution. En revanche, on trouve moins de *flowgraphs* complets basés sur Pothos que sur GNU Radio, qui reste la référence la plus citée. Pothos est accompagné de la librairie **SoapySDR**, qui a pour rôle d'abstraire la gestion du matériel, en fournissant une API unique pour gérer la plupart des SDR disponibles sur le marché. Cette API est disponible dans les langages C, C++, Python, Go et Rust. Nous avons utilisé SoapySDR dans notre projet pour pouvoir facilement supporter le matériel à notre disposition.

4.1.3 Approche générale

Nous avons choisi d'étudier des approches de détection d'intrusions qui pourraient s'appliquer à un large éventail de protocoles différents, notamment ceux que l'on voit apparaître avec les objects communicants. Afin de découpler au maximum l'IDS de ces cibles, et d'éviter d'interférer dans leurs échanges, nous avons étudié des méthodes basées sur l'écoute passive des communications au niveau du réseau sans fil.

Écoute passive

L'écoute passive de communications sans fil n'est pas toujours évidente. Avec certains protocoles il est possible d'activer le mode « moniteur » d'un équipement standard pour récupérer les échanges bruts, c'est par exemple le cas du Wi-Fi. Avec d'autres protocoles, comme par exemple le Bluetooth Classic, les adaptateurs disponibles sur le marché ne proposent pas ce genre de fonctionnalités, et il faut donc recourir à des équipements beaucoup plus coûteux, capables d'écouter tous les canaux simultanéments, pour écouter leurs communications.

Écoute multi-protocoles et SDR

Pour écouter simultanément des communications sans fil de différents protocoles, nous distinguons deux principales options. La première option consiste à utiliser plusieurs modules d'acquisition ad-hoc : un pour chaque protocole visé. Par exemple, nous pouvons

utiliser des adaptateurs Wi-FI, ZigBee, etc. pour écouter les échanges dans ces protocoles. Cette méthode a l'avantage d'être simple à mettre en œuvre, et elle permet d'obtenir des données de qualité car ces modules offrent généralement de bonnes performances – ils sont faits pour ça. En revanche, cette approche n'est pas forcément généralisable à tous les protocoles : certains ne proposent pas de mode moniteur, ou sont même spécifiquement construits pour rendre difficile l'écoute passive.

La seconde option consiste à utiliser un seul matériel pour l'acquisition des signaux radio, puis d'implémenter en logiciel (SDR) la démodulation pour chaque protocole ciblé. L'avantage de cette méthode est qu'elle peut être généralisée à tous les protocoles du moment qu'ils utilisent des fréquences et des bandes passantes supportées par notre matériel. Sans même forcément démoduler entièrement tous les paquets, il sera ainsi au moins possible d'en déceler la présence, et d'obtenir des informations utiles pour la détection d'intrusions, y compris pour détecter des attaques intervenant sur la couche physique, par exemple de brouillage. Par contre, cette méthode a le désavantage d'être plus complexe à mettre en œuvre, et nécessite une plus grande puissance de calcul pour réaliser le traitement du signal en logiciel.

C'est cette seconde approche que nous avons choisi d'explorer.

4.1.4 Objectifs détaillés

Nous avons décidé d'utiliser une SDR pour récolter des informations sur les réseaux sans-fil environnants, afin de nourrir un système de détection d'intrusions. Pour être utile dans un tel contexte, notre SDR se devait de répondre aux besoins suivants :

Observation de canaux prédéterminés.

Dans un contexte de détection d'intrusions, nous avons besoin de pouvoir observer l'activité sur des canaux prédéterminés : ceux qui sont utilisés par les appareils à protéger. Observer des fréquences qui ne sont pas supportées par nos propres réseaux de communication n'offre pas de réel intérêt pour notre cas d'usage. Nous souhaitons également pouvoir observer plusieurs canaux simultanément lorsque c'est possible. En effet, les protocoles employés dans le contexte de l'IdO ont généralement un bas débit, et utilisent donc une faible bande passante. Si notre matériel SDR supporte une plus large bande passante, on a donc l'opportunité d'observer l'activité de plusieurs canaux contigus. De plus, plusieurs protocoles utilisent les mêmes bandes de fréquences ISM, par exemple celle du 2,4 GHz. Dans ce cas, on peut observer les canaux de plusieurs protocoles différents simultanément.

Permettre la démodulation et le décodage de protocoles connus.

Nous considérons que les analyses sur la couche physique, rendues possibles par les SDR, peuvent apporter une plus-value aux IDS sans fil. Cependant, les méthodes qui se basent sur les paquets démodulés sont également utiles. Les deux approches sont en réalité complémentaires.

Notre système doit donc pouvoir démoduler les paquets qu'elle collecte, afin de permettre de récupérer les mêmes informations que celles fournies par un récepteur ad-hoc. Dans certains protocoles, le fonctionnement de la couche physique n'est pas documenté. C'était par exemple le cas de celle du LoRa à ses débuts, avant qu'elle ne soit documentée par rétro-ingénierie. Pour ces protocoles, notre solution doit permettre un fonctionnement en mode dégradé, qui permettra les analyses sur la couche physique, mais pas celles sur les paquets démodulés.

Fonctionner en continu.

Pour pouvoir servir de sonde à un IDS, notre solution doit pouvoir fonctionner en continu. Ainsi, l'IDS pourra fonctionner en permanence, et générer rapidement des alertes. On peut donc accepter une latence de quelques secondes, mais notre système ne doit pas avoir besoin d'être mis en pause pour traiter les données qu'il ingère. Si le volume de données entrant dépasse temporairement nos capacités de traitement, on admettra une perte de certains paquets, qui pourra être elle-même journalisée.

Collecte d'indicateurs sur chaque couche.

Notre système doit permettre la collecte d'indicateurs à chaque étape du traitement du signal. Ces indicateurs peuvent être des données brutes, comme par exemple les échantillons correspondant à un paquet reçu, ou des métadonnées, par exemple le rapport signal sur bruit (SNR), ou un identifiant extrait d'un paquet démodulé.

Chaque indicateur doit être associé au canal physique sur lequel il a été reçu, et porter une estampille temporelle précise (*timestamp*). En effet, les temps d'arrivée des paquets peuvent constituer une source d'information non négligeable.

Enfin, si les données brutes constituent une riche source d'information, elles peuvent être trop volumineuses pour être conservées sur de longues périodes. Notre système doit donc être flexible, et on doit pouvoir choisir les informations qui seront journalisées selon le type d'analyses que l'on veut mener.

Minimiser le temps de calcul.

Dans une SDR, l'essentiel du traitement du signal est réalisé en logiciel. Cela peut donc nécessiter une grande puissance de calcul, qui augmente avec la bande passante. Nous cherchons à concevoir un outil capable d'écouter simultanément plusieurs canaux adjacents, correspondant à différents protocoles. En supportant une plus large bande passante, on pourra donc observer plus de canaux simultanément et récupérer plus d'information. Il est donc important que notre système soit efficace, pour permettre d'exploiter la plus large bande de fréquence possible.

CPU conventionnels Les radio-logicielles nécessitent une importante puissance de calcul. C'est la raison pour laquelle on privilégie souvent les FPGA par rapports aux CPU

conventionnels : ils sont mieux adaptés pour traiter d'importants volumes de données en temps réel. Cependant, les FPGA sont plus complexes à programmer, et peuvent être très coûteux. Cela constitue un important frein à leur adoption, et restreint leur domaine d'application. Les GPU peuvent également considérablement accélérer le traitement du signal, grâce à leur architecture massivement parallèle qui fournit une importante puissance de calcul. C'est donc une alternative de plus en plus crédible aux FPGA. Mais comme les FPGA, les GPU sont coûteux et complexes à programmer. Nous avons donc choisi de baser notre solution sur des CPU conventionnels.

Réduire les traitements inutiles Pour illustrer le coût des traitements nécessaires à notre SDR, décrivons d'abord une approche naïve.

Prenons l'exemple des protocoles Bluetooth Classic et 802.15.4 (voir figure 4.7) qui coexistent dans le spectre 2,4 GHz. Les canaux Bluetooth ont une largeur de un MHz, et sont espacés d'un MHz. Les canaux 802.15.4 ont une largeur de 2 MHz et sont espacés de 5 MHz. Dans la bande de fréquence comprise entre 2402 MHz et 2412 MHz, on trouve donc dix canaux Bluetooth Classic, et deux canaux 802.15.4.

Si on souhaite détecter tous les paquets Bluetooth et 802.15.4 dans cette bande, on doit d'abord isoler les canaux, puis les démoduler et rechercher des paquets dans le flux démodulé. Pour isoler un canal, il faudra appliquer un filtre passe-bande, recentrer le signal, puis éventuellement le rééchantillonner à la fréquence voulue.

Cette approche fonctionne bien pour un nombre limité de canaux, mais sera rapidement submergée si on augmente la largeur de la bande. D'abord, en augmentant la largeur de la bande, on aura plus de canaux et donc plus de filtres à exécuter. De surcroît, si la bande passante en entrée est plus large, alors nos filtres devront comporter plus de coefficients et chaque filtre sera donc plus coûteux à exécuter. La puissance de calcul nécessaire pour filtrer les canaux augmentera donc linéairement à mesure que l'on élargit la bande observée

Heureusement, cette approche naïve comporte une importante marge d'amélioration. Tous les canaux ne sont pas utilisés en permanence, et en réalité il sont silencieux la majeure partie du temps.

En placant une première couche en amont du filtrage, on peut détecter les canaux actifs, et n'isoler que ceux là. On peut ainsi considérablement réduire la charge sur le processeur en évitant de réaliser des calculs inutiles.

4.2 Architecture proposée

Nous avons réalisé un outil basé sur une radio-logicielle pour la capture multi-standards sur une large bande de fréquence. Ayant choisi de n'utiliser que des CPU conventionnels, il était important de limiter au maximum la quantité de calculs, afin d'obtenir des performances satisfaisantes.

Pour cela, nous avons choisi de découper notre solution en plusieurs couches, à la manière de RFDump. Les couches les plus basses doivent traiter d'importants volumes de

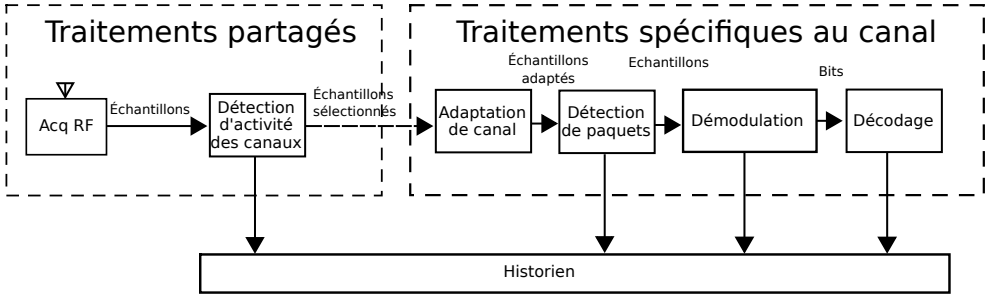


FIGURE 4.1: Infrastructure de capture

données, et ont pour rôle de réduire ce volume, tout en annotant les données afin de guider le travail des couches supérieures.

4.2.1 Description générale

La figure 4.1 décrit l'architecture générale. Nous la décrivons de gauche à droite. Tout commence avec le matériel d'entrée RF, qui renvoie les échantillons bruts correspondant à la bande de fréquence observée. Cette bande est ensuite traitée par le **détecteur d'activité des canaux**. La détection d'activité est un traitement **partagé**, qui correspond à une file d'exécution unique. Lorsqu'un canal est actif est détecté, les échantillons correspondants sont transférés à la file d'exécution spécifique à ce canal, où le signal est **adapté**, puis éventuellement **démodulé** et **décodé**. Chaque couche de cette architecture fournit des informations qui sont journalisées. Ces informations condensées peuvent ensuite faire l'objet de traitements supplémentaires, et être analysées par un IDS afin de détecter des menaces.

Nous présentons ici les termes utilisés dans *Phyltre* pour désigner les traitements et les formes du signal à chaque étape.

— **Canal**. Un **canal** représente une bande de fréquence correpondant à un des canaux physiques d'un des protocole supporté. Comme nous supportons plusieurs protocoles, et que ces protocoles utilisent généralement plusieurs canaux, on peut donc avoir à traiter plusieurs canaux de différents protocoles dans un même signal brut.

— **Signal brut.** Le **signal brut** correspond aux échantillons qui sont acquis par notre entrée RF. Ce signal représente une bande de fréquence donnée, définie par sa fréquence centrale et sa bande passante. Ce signal est en principe continu, et possède donc un débit fixe. Le signal brut peut porter un *timestamp* qui donne une référence précise sur le temps d'arrivée des échantillons.

— **Signal actif.** Un **signal actif** est une section du signal brut dans laquelle un canal actif a été détecté. Le signal actif est donc discontinu : c'est un extrait du signal brut, étiqueté avec le canal détecté et son temps d'arrivée. Seuls les signaux actifs font l'objet de **traitements spécifiques**.

— **Signal adapté.** Avant de pouvoir être démodulé, les signaux actifs doivent être adaptés. En effet les signaux actifs représentent encore toute la bande passante de

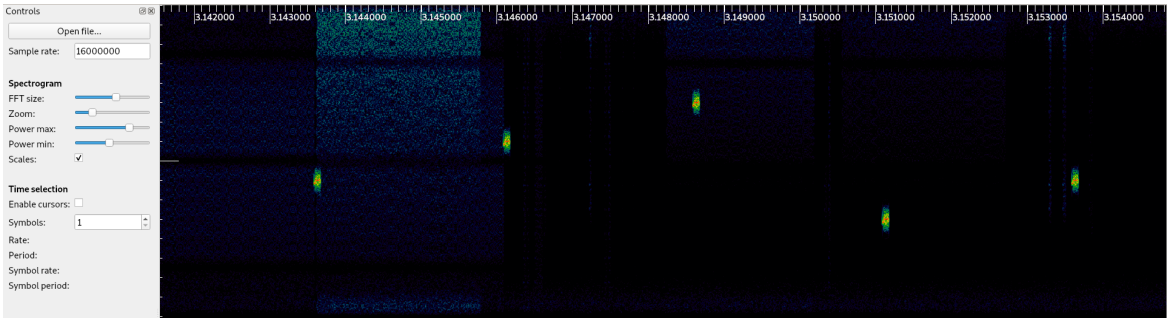


FIGURE 4.2: Signal brut

l'entrée RF. Le signal adapté ne contient plus que la bande passante du canal ciblé, correctement centrée sur la fréquence du canal, à un taux d'échantillonage adéquat. Le signal adapté est donc généralement moins volumineux que le signal actif.

— **Traitement spécifique.** Ces traitements sont spécifiques à un canal donné. Ils effectuent notamment les tâches de démodulation correspondantes au protocole de ce canal. Comme ces traitements peuvent être coûteux en puissance de calcul, ils ne sont réalisés que sur les signaux adaptés.

4.2.2 Chaine de traitement.

Cette section détaille les différentes étapes de la chaine de traitement décrites dans la figure 4.1.

Entrée RF.

L'entrée RF est l'unique élément matériel de notre architecture, qui nous sert à acquérir les ondes RF. Il existe une large gamme de matériel SDR, qui varient dans leur fonctionnalités – émission, réception, nombre de chemins d'antennes, les bandes de fréquences et taux d'échantillonnages supportés. Le signal est capturé « en bande de base », sous forme de signaux IQ bruts, qui en pratique contiennent toute l'information sur la bande de fréquence correspondant au taux d'échantillonnage. Les signaux IQ sont transférés à l'hôte le plus souvent via un lien USB ou IP. Ces signaux peuvent être volumineux, et les traiter en temps réel est alors un défi. L'entrée RF fournit généralement les échantillons sous forme de paires de nombres signés à virgule fixe de 16 bits. Ainsi, à une bande passante de 40 MHz, le signal d'entrée brut dépasse un gigabit par seconde. Le signal brut peut être enregistré afin d'être analysé avec d'autres outils, moyennant une importante capacité de stockage et d'une vitesse d'écriture suffisante pour tenir le débit du signal.

La figure 4.2 montre le spectrogramme d'un signal brut réalisé dans la bande ISM 2,4 GHz pendant un scan Bluetooth [3]. L'axe vertical représente la fréquence en Hertz et

3. Les images de signaux ont été réalisées avec l'outil *Inspectrum*.

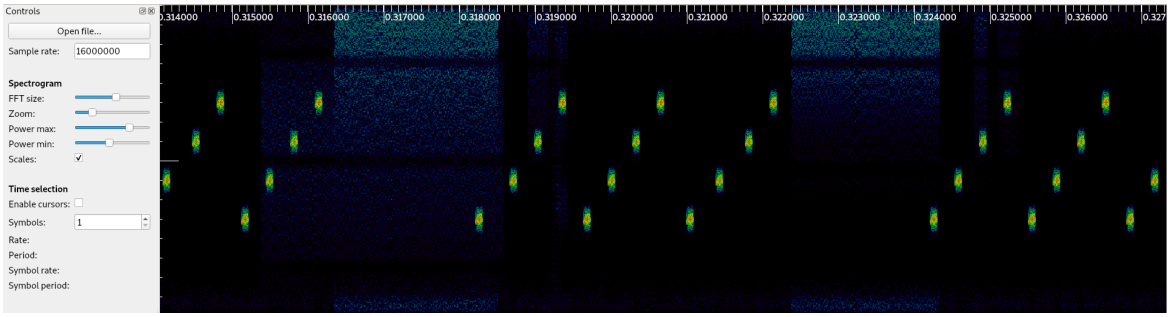


FIGURE 4.3: Ensemble des signaux actifs de la capture 4.2.

l'axe horizontal le temps en secondes. La capture a été réalisée par une BladeRF 2.0, avec un taux d'échantillonnage de 16 Msps, et une fréquence centrale de 2412MHz. On voit bien des paquets Bluetooth provenant de notre scan, et en arrière plan quelques trames Wi-Fi (plus larges) émises par des appareils plus distants.

Détection d'activité des canaux.

La détection d'activité des canaux a pour rôle de détecter les canaux actifs dans la bande de fréquence observée. Pour cela, elle met en œuvre une détection d'énergie dans le domaine fréquentiel sur le signal brut, et considère une canal actif lorsque l'énergie détectée sur ce canal dépasse un seuil prédéfini. Cette couche permet ainsi d'éviter de réaliser des traitements coûteux et inutiles : elle selectionne les parties actives du signal, et ne transmet que les signaux actifs pour un canal à la file d'exécution spécifique à ce canal. La détection d'énergie dans le domaine fréquentiel permet de bien distinguer les fréquences, mais en contrepartie elle n'offre qu'une faible résolution dans le domaine temporel. Pour éviter de perdre des morceaux de paquets, on ajoute donc une marge au début et à la fin des signaux actifs.

La figure 4.3 montre l'ensemble signaux actifs sélectionnés dans le signal brut de la figure 4.2 par notre détecteur d'activité. Nous avons configuré notre détecteur d'activité pour détecter les canaux Bluetooth actifs avec un rapport signal-bruit d'au moins 8 dB. On voit des paquets Bluetooth nettement plus rapprochés, car le détecteur a éliminé une grande partie des périodes silencieuses du signal brut. On voit aussi que quelques trames Wi-Fi ont été conservées par erreur par le détecteur d'activité. Dans notre expérience, cela ne pose pas de problème : le rôle du détecteur d'activité est d'abord de réduire au maximum la quantité de travail des couches supérieures, en tolérant des faux positifs.

Les signaux actifs sont annotés avec le ou les canaux détectés. La figure 4.4 montre les signaux actifs de la figure 4.3 annotés avec le canal Bluetooth 31. Seuls ces signaux sont transférés aux traitements spécifiques au canal Bluetooth 31. Cette fois-ci, seuls les paquets du canal Bluetooth 31 sont visibles. Cependant, à ce stade le signal n'est pas encore prêt à être démodulé. Il n'est pas centré autour de la fréquence du canal Bluetooth 31, et possède encore la large bande de l'entrée RF (16 MHz). C'est l'étape d'adaptation de canal qui

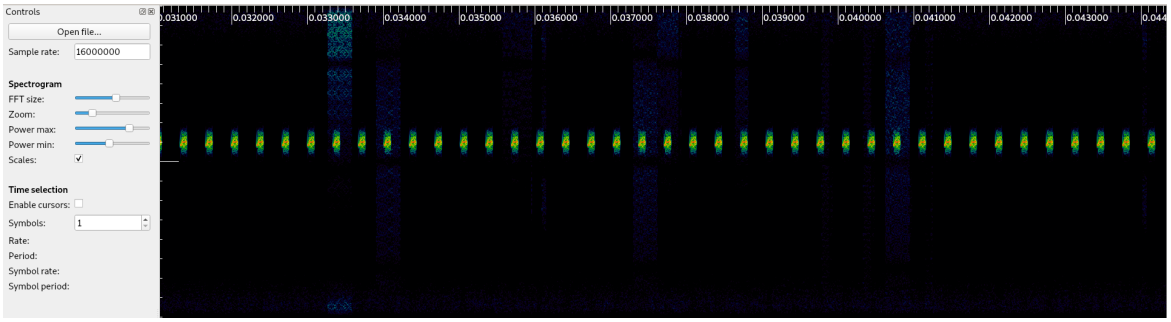


FIGURE 4.4: Signaux actifs de la capture 4.2 pour le canal Bluetooth 31.

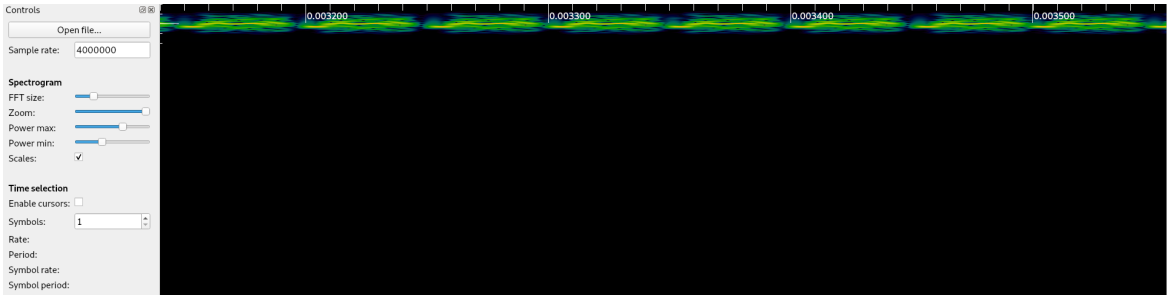


FIGURE 4.5: Signaux adaptés pour le canal Bluetooth 31.

permettra d'isoler le signal avant de procéder à la démodulation.

Adaptation de canal.

L'adaptation de canal est le premier traitement spécifique à un canal. Cette étape permet de passer du signal actif – contenant toute la bande passante de l'entrée RF – au signal adapté. Pour cela, le signal est recentré autour de la fréquence du canal, filtré pour ne contenir que la bande passante correpondant au canal, et rééchantilloné à un taux adéquat pour permettre la démodulation. Une fois le signal adapté, une seconde étape de détection d'énergie – dans le domaine temporel – détecte cette fois avec précision le début et la fin d'un paquet en observant les variations de SNR, afin d'éliminer les marges conservées lors de l'étape précédente. Au cours de cette étape on peut journaliser les variations de SNR mesurées, la taille estimée des paquets, ou simplement le signal adapté. La figure 4.5 montre les signaux du canal Bluetooth 31 de la figure 4.4 après leur adaptation. Le signal est désormais plus étroit (ici échantillonné à 4 Msps), et centré autour de la fréquence du canal. Dans cette image, l'échelle de temps est raccourcie, et on voit huit paquets consécutifs prêts à être démodulés.

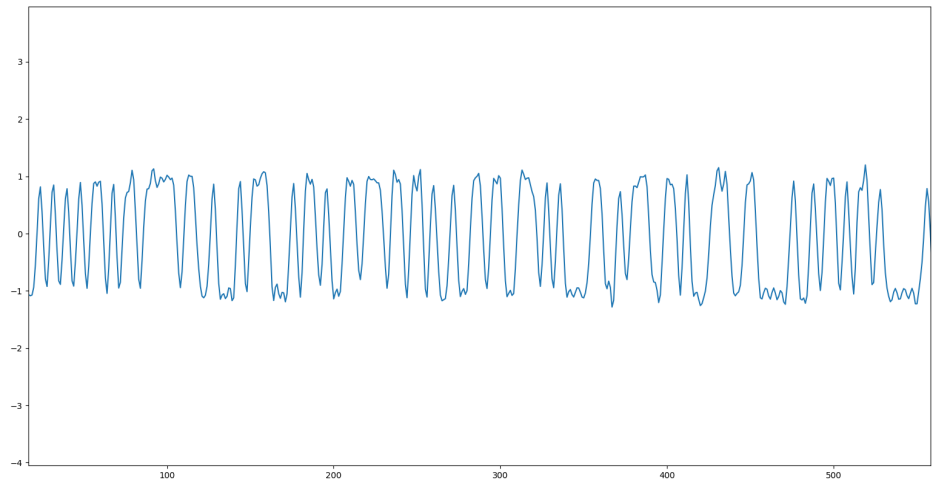


FIGURE 4.6: Début d'un paquet Bluetooth Low Energy après démodulation FM.

Démodulation.

La démodulation a pour rôle de passer du signal adapté aux symboles. Ainsi, elle permet de passer d'une représentation IQ du signal à des bits. À l'heure actuelle, seule la modulation de fréquence (FM) est implémentée dans notre solution. Elle peut être configurée pour supporter les protocoles Bluetooth Classic, Bluetooth Low Energy, et 802.15.4. Dans le futur, il sera possible d'ajouter le support d'autres types de modulation par des mises à jour logicielles. La phase de démodulation peut générer des métadonnées sur les erreurs mesurées, qui peuvent permettre de distinguer des transmetteurs. La figure 4.6 montre le début d'un paquet Bluetooth Low Energy démodulé en fréquence. L'axe vertical montre la variation de fréquence normalisée, et l'axe horizontal le nombre d'échantillons. Une variation positive de la fréquence encode un bit 1, et un variation négative un bit 0.

Décodage.

Le décodage suit la phase de démodulation. Cette phase est très spécifique, et son implémentation varie donc beaucoup selon le protocole. Si les paquets contiennent un préambule, le décodeur peut d'abord rechercher ce préambule dans les symboles qui le composent afin d'aligner le reste du paquet sur le préambule. Dans la plupart des protocoles que l'on a étudié, les paquets sont encodés avant d'être émis, pour augmenter la résistance au bruit (Forward error correction, Viterbi, etc.), et d'éliminer la composante continue du signal en enlevant les suites de "0" ou de "1" (Whitening). Il faut alors les décoder pour rendre leur contenu intelligible. Cette phase peut donc générer différents types de métadonnées qui renseignent sur le contenu des paquets : identifiants, en-têtes, symboles

alignés, ou même le contenu décodé des paquets.

4.3 Implémentation : *Phyltre*

Nous avons implémenté notre solution : *Phyltre*. Dans cette section, nous décrivons l'implémentation des différents composants présentés dans la partie précédente.

4.3.1 Format du projet

Nous avons conçu *Phyltre* sous forme d'un projet autonome, fonctionnant pour l'heure sur les systèmes POSIX, mais avec un nombre de dépendances réduites qui permet d'envisager des portages pour d'autres OS, ou même des plateformes embarquées. Le projet est écrit en langage C pour obtenir une implémentation efficace, et compte environ 7000 lignes de code.

La librairie liquid-dsp [Gae] nous fournit l'essentiel des fonctions de traitement du signal dont nous avons besoin, à l'exception des transformées de Fourrier rapides (FFT) qui sont faites à l'aide la librairie FFTW [FJ].

Pour le support de l'entrée RF nous utilisons la librairie SoapySDR. Elle fournit une couche d'abstraction qui permet de manipuler des équipements de différents modèles et constructeurs avec une interface unique. Cela nous a permis en pratique de tester notre logiciel avec tous les récepteurs SDR dont nous disposions au laboratoire : HackRF, PlutoSDR, LimeSDR, BladeRF.

Notre projet se compose d'une librairie incluant le moteur d'exécution et les modules de traitement du signal, et de plusieurs applications correspondant à différents scénarios expérimentaux.

Configuration

La configuration de l'outil est décrite dans des fichiers INI, que l'on peut facilement modifier avec un éditeur de texte, *vim* dans notre cas. Au moins trois fichiers de configuration sont nécessaires :

- **Source d'acquisition** : les paramètres de l'entrée RF : fréquence centrale, bande passante, gain, et valeur du plancher de bruit (qui peut être fixée ou déterminée automatiquement) ;
- **Détecteur d'activité des canaux** Le principal paramètre est le nombre de points du spectrogramme, qui détermine la résolution temporelle et fréquentielle du détecteur ;
- **Couche physique** : il peut y avoir plusieurs fichiers de configuration des couches physiques, un pour chaque protocole supporté. Ces fichiers décrivent les canaux physiques : la bande passante, le taux d'échantillonage, la fréquences des canaux, la taille minimale d'un paquet, et le SNR minimal. Ces informations sont utilisés par le détecteur d'activité et le module d'adaptation de canal. Viennent ensuite la

configuration de la démodulation et du décodage, ainsi que le choix des métadonnées qui doivent être sauvegardées.

Journalisation

Notre outil sert à collecter des informations sur la couche physique. Il est donc important de correctement journaliser ces informations dans un format qui puisse être interfacé avec d'autres outils.

Le principal format utilisé pour stocker des signaux radio est le *complex64*, qui représente les échantillons IQ sous la forme de paire de flottants IEEE 754 de 32 bits. Ce format est brut : il ne contient ni en-tête ni métadonnées d'aucune sorte, même pas les paramètres qui ont servi à réaliser la capture. Ce format très simple est supporté par la plupart des logiciels SDR. Pour que notre outil soit compatible avec les autres logiciels, il est donc important de pouvoir générer des traces au format complex64.

En revanche, nous avons besoin d'un format de fichier plus complet, qui puisse contenir non seulement des échantillons bruts, mais aussi les paramètres et l'heure d'acquisition, et les métadonnées collectés par les couches successives. Enfin, chaque donnée sauvegardée doit porter un *timestamp* permettant d'identifier précisément l'heure de l'évènement.

Pour stocker les informations collectées, on génère lors d'une capture un **fichier de capture** et un ou plusieurs **fichiers de canal**.

Le fichier de capture contient les informations des traitements partagés : l'heure d'acquisition, les paramètres de l'entrée RF, et peut optionnellement stocker les détections d'énergie, ainsi que les échantillons ayant contribué à ces détections.

Un fichier de canal stocke les données générées par la file de traitement spécifique à un canal. Ces traitements ne sont pas faits sur un signal continu, mais sur des **paquets** de données, dont on détermine les bornes à l'aide d'une détection d'énergie dans le domaine temporel. Les métadonnées correspondant à un paquet sont stockées dans le fichier de canal sous la forme d'une suite d'enregistrements qui correspondent aux différentes étapes de traitement. Le premier enregistrement doit être de type PEAK, et contient les variations de SNR renvoyées par le détecteur d'énergie temporel. La séquence des autres enregistrements dépend du protocole du canal et comprend les métadonnées générées par le démodulateur et le décodeur du canal. Optionnellement, le fichier de canal peut également conserver les échantillons adaptés des paquets détectés.

Les fichiers de capture et de canal utilisent un format *Type-Length-Value* (TLV) que nous avons conçu pour les besoins de notre application. Les fichiers TLV permettent de stocker une suite d'enregistrements de tailles arbitraires, et offrent une grande flexibilité. Une application peut les parcourir et n'utiliser que certains enregistrements, sans avoir à connaitre le format des autres types d'enregistrements. Ainsi si on ajoute un nouveau type de métadonnées dans notre outil de capture, on pourra toujours utiliser nos anciens outils qui ne connaissent pas cette métadonnée : ils pourront simplement ignorer ces enregistrements de type inconnu. De même, on peut rapidement créer des outils d'analyse qui n'exploitent qu'un sous-ensemble des métadonnées, en n'implémentant que le support de ces enregistrements TLV.

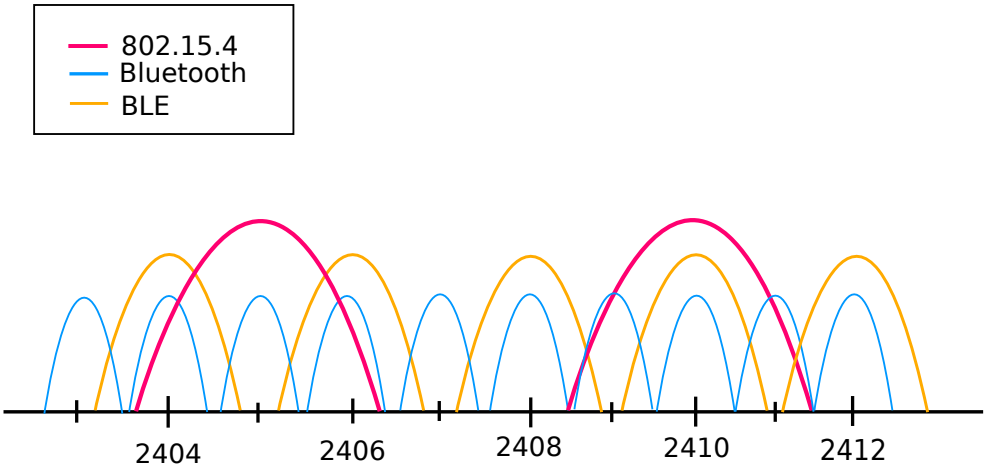


FIGURE 4.7: Canaux superposés dans la bande ISM 2,4 GHz

Enfin pour assurer la compatibilité avec les outils existants, on peut extraire les échantillons stockés dans les fichiers TLV vers un fichier complex64.

4.3.2 Détection d'activité des canaux

La détection d'activité des canaux est un traitement partagé qui a pour rôle d'appliquer une détection d'énergie dans le domaine fréquentiel, afin de distinguer les canaux actifs dans la bande de fréquence de l'entrée RF. Il peut y avoir plusieurs canaux de différents protocoles dans une même bande de fréquence. Par exemple, la figure 4.7 montre des canaux Bluetooth, BLE et 802.15.4 sur une portion de la bande ISM 2,4 GHz.

Configuration de la détection d'activité

Comme le détecteur d'activité s'applique à tous les protocoles, il est agnostique au type de modulation. Il ne repose que sur la description des canaux physiques, et un seuil défini dans le fichier de configuration de la couche physique. Nous représentons les canaux physiques par trois attributs :

- **La fréquence centrale du canal (fc)** ;
- **Les bandes de fréquences (fb)** : une ou plusieurs bandes de fréquences autour de la fréquence centrale ;
- **La durée minimale d'une trame (mfd)** : la durée de la plus courte trame du protocole, exprimée en secondes.

Il peut y avoir plusieurs valeurs de *fc* : une pour chaque canal du protocole, tandis que les valeurs *fb* et *mfd* sont communes à tous les canaux du protocole.

Le seuil de détection est piloté par deux valeurs :

- **minSNR** : pour chaque bande *fb*, un seuil en dB au dessus du plancher de bruit mesuré ;

— **maxRel** : SNR maximum (en dB relatifs au niveau de référence).

Exemple de configuration 1. Nous avons observé que pour les paquets Bluetooth Low-Energy (BLE), l'énergie est concentrée dans une bande de 1,5 MHz autour de la fréquence centrale du canal. L'exemple que nous proposons peut être représenté par une seule bande de fréquence :

$$fb = -750e3, 750e3$$

On peut ensuite définir un seuil sur cet unique bande : $minSNR = 5$ (dB au dessus du plancher de bruit).

Ces paramètres permettent de filtrer les périodes de blanc, en ne conservant que les échantillons où des transmissions BLE sont possiblement présentes (avec un SNR d'au moins 5). Cependant, une transmission intervenant sur une bande beaucoup plus large – recouvrant la bande de notre canal BLE – pourrait également déclencher une détection. Par exemple, un paquet Wi-Fi (d'une largeur de 11 à 20 MHz), sur un canal superposé avec notre canal BLE, pourrait être mépris pour un paquet BLE.

C'est pour cette raison qu'un canal peut être décrit par plusieurs bandes de fréquences (fb), qui peuvent être à l'intérieur ou à l'extérieur du canal physique réel.

Nous reprenons ici l'exemple du BLE, que nous décrivons cette fois à l'aide de trois bandes des fréquences :

$$fb = \{[-1.5e6, -750e3][-750e3, 750e3][750e3, 1.5e6]\}$$

$$minSNR = 0; 5; 0$$

$$maxRel = -5; 0; -5$$

Dans cette configuration, deux bandes de fréquences extérieures sont ajoutées. On présume qu'au cours d'une transmission BLE, ces bandes extérieures auront un niveau d'énergie bien inférieur à celui de la bande centrale. La configuration de minSNR est laissée à l'identique, car on ne spécifie un SNR minimum que pour la bande centrale. Nous avons cette fois-ci ajouté une configuration maxRel, qui spécifie un SNR maximum relativement au niveau de référence. Le niveau de référence correspond à la moyenne du SNR des bandes dont le maxRel vaut zéro (dans notre exemple, la bande centrale). Pour déclencher notre détecteur, le niveau d'énergie des deux bandes extérieures doit désormais être inférieur d'au moins 5 dB au niveau mesuré dans la bande centrale.

Cette configuration est beaucoup moins sujette à confusion avec des paquets Wi-Fi, au prix d'un nombre plus élevé de faux-positifs, qui pourraient par exemple se produire lorsque deux canaux adjacents sont actifs au même moment.

Exemple de configuration 2. Cette façon flexible de configurer la détection d'énergie nous permet également d'éviter des faux-positifs lorsqu'une transmission puissante intervenant sur une bande étroite pourrait déclencher la détection d'un plus large canal. Par exemple, on peut définir un canal Wi-Fi de 11 MHz comme :

$$fb = [-5.5e6, 5.5e6]$$

$$minSNR = 5$$

Ce détecteur simple pourrait également être déclenché si un signal à bande étroite suffisamment puissant apparaissait n'importe où dans la bande spécifiée de 11 MHz.

Nous pourrions alors mieux décrire le même canal avec deux *fb* :

$$fb = \{[-5.5e, 0][0, 5.5e6]\}$$

$$minSNR = 5; 5$$

Cette seconde spécification est assez similaire à la première, mais est désormais beaucoup moins sujette à confusion avec des signaux à bande étroite, du moment qu'ils ne partagent pas la fréquence centrale de notre canal Wi-Fi.

Détection dans le domaine fréquentiel.

Nous avons implémenté la méthode de Bartlett [Bar48] pour évaluer la puissance des signaux sur les différentes fréquences de la bande de l'entrée RF. Bartlett estime la densité spectrale de puissance (PSD) en calculant la transformé de Fourier discrète (DFT) sur des sections disjointes de N échantillons, et en moyennant leur amplitude sur une période de K sections.

Nous calculons une valeur de K par protocole de la configuration. Cette valeur est dérivée du *mfd*, ce qui nous permet de minimiser la variance dans l'estimation, tout en conservant la possibilité de détecter le début et la fin d'une transmission.

$$K = \left\lceil \frac{mfd \times sampleRate}{N} \right\rceil$$

Moyenner la PSD sur une période supérieure à la *mfd* pourrait provoquer des faux-positifs, car la solution ne serait pas capable de détecter des transmissions puissantes mais courtes, dont la puissance serait amoindrie par un moyennage de trop long terme.

La valeur de N doit elle être choisie avec soin, pour être petite au regard du plus court *mfd* de notre configuration, sans trop perdre de résolution dans le domaine fréquentiel. Par exemple, nous fixons habituellement N à 512 pour un taux d'échantillonnage de 20 MHz, ce qui nous donne une résolution de 25.6 μs dans le domaine temporel, et approximativement 39 kHz dans le domaine fréquentiel.

Le processus de détection d'activité se déroule en trois étapes :

— **Traitement commun :** calculer la DFT pour chaque section de N échantillons.
— **Pour chaque protocole :** estimer la PSD en accumulant l'amplitude de la DFT sur une période K.
— **Pour chaque canal :** sommer les valeurs de la PSD pour chaque bande de fréquence *fb* du canal, appliquer les seuils de détection minSNR et maxRel.

Initialisation. Lors de la phase d'initialisation, nous déterminons les paramètres internes du détecteur d'activité, en les dérivant de la spécification des canaux physiques et des seuils de détection qui les accompagnent.

Pour chaque bande de fréquences *fb*, ses bornes sont calculées par :

$$bFreqMin(i) = fc+bandLow(i)$$

$$bFreqMax(i) = fc+bandHigh(i)$$

La résolution fréquentielle est donnée par :

$$freqReso = \frac{sampleRate}{N}$$

Les index des points correspondants à *fb* dans la PSD sont obtenus de la façon suivante :

$$loBin(i) = \left\lfloor \frac{fc+bFreqMin(i) - capLowFreq}{freqReso} \right\rceil$$

$$hiBin(i) = \left\lfloor \frac{fc+bFreqMax(i) - capLowFreq}{freqReso} \right\rceil - 1$$

Le nombre de points correspondants dans la PSD vaut donc :

$$binCnt(i) = hiBin(i) - loBin(i) + 1$$

Le seuil de puissance ajusté $T(i)$ pour la bande de fréquence i est défini par :

$$T(i) = binCnt(i) \times 10^{\frac{\left(nfe+minSNR(i)\right)}{10}}$$

Où *nfe* (pour *Noise Floor Estimation*) est l'estimation du plancher de bruit (en dBm) pour un point de la PSD. Ce paramètre peut être fixé dans la configuration, ou déterminé automatiquement au démarrage (*nfe=auto*).

Le niveau maximal de référence *mrl* pour une bande de fréquence i est donné par :

$$mrl(i) = maxRel(i) \times \frac{binCnt(i)}{refBinCnt}$$

Où *refBinCount* est le nombre total de points de la PSD dans toutes les bandes de fréquences de référence.

Exécution. L'algorithme de détection de canal utilise les paramètres internes et la PSD courante, pour décider si une section de signal excède son seuil de détection.

Pour chaque bande de fréquence, son niveau de puissance total est estimé par :

$$p(i) = \sum_{i=loBin(i)}^{hiBin(i)-1} PSD_i$$

Le niveau de référence *refSum* est calculé en sommant $p(i)$ pour toutes les bandes de référence.

Le niveau de puissance d'une bande(i) peut alors être directement comparé au seuil $T(i)$:

$$matchMinSNR(i) = p(i) \geqslant T(i)$$

Le seuil de puissance maximal relatif au niveau de référence est vérifié par :

$$matchRelLevel(i) = \frac{p(i)}{refSum} < mrl(i)$$

Lorsque les deux seuils de détections *matchMinSNR* et *matchRelLevel* sont atteints, les échantillons actifs sont passés au processeur de canal correspondant, où commencent les traitements spécifiques à ce canal.

4.3.3 Adaptation des canaux

L'adaptation de canal est la première étape de traitement spécifique à un canal. Ce processus a pour rôle d'isoler le signal correspondant à ce canal dans la bande passante de l'entrée RF. En effet, les signaux actifs – transmis par le détecteur d'activité – sont des sections du signal d'entrée, qui n'ont à ce stade subi aucun traitement. Ces signaux ont donc la bande passante de l'entrée RF, qui est typiquement plus large que celle des canaux que l'on observe.

Pour mieux comprendre ce processus, situons nous à nouveau dans l'exemple de la figure 4.7. L'entrée RF est centrée sur 2408 MHz avec une bande passante de 12 MHz, et on souhaite démoduler le canal Bluetooth donc la fréquence centrale est de 2404 MHz. En pratique, le signal renvoyé par notre entrée RF est en bande de base, on considère donc que sa fréquence centrale vaut zéro et notre canal Bluetooth a une fréquence relative de -8 MHz. Un canal Bluetooth a un débit d'un 1 Mbps, et pour le démoduler il nous faut au moins deux échantillons par bit, en vertu du théorème de Nyquist-Shannon. En sortie, notre signal adapté devra donc avoir un taux d'échantillonnage d'au moins 2 MHz, et une fréquence centrale de zéro.

Le processus se déroulera ici en trois étapes :

- **Filtrage :** le signal est convolué avec un filtre à réponse impulsionnelle finie (FIR) passe-bande, pour éliminer les signaux qui ne sont pas dans la bande passante de notre canal. À l'issue de ce traitement, le signal conserve donc sa bande passante mais les fréquences extérieures à notre canal Bluetooth sont désormais silencieuses.
- **Translation de fréquence :** le signal est multiplié avec une sinusoïde complexe, afin de translater sa fréquence pour le recentrer autour de la fréquence du canal.
- **Rééchantillonnage :** le signal est décimé, pour obtenir le taux d'échantillonnage souhaité. Ici, on ne conservera qu'un échantillon sur six, ce qui aura pour effet de réduire la bande passante du signal de 12 MHz à 2 MHz.

Notez qu'il n'est pas possible de rééchantillonner le signal par une simple décimation lorsque le taux d'échantillonage de l'entrée RF n'est pas multiple de celui du signal adapté.

Dans ce cas, on effectue donc un second rééchantillonnage, à l'aide de l'*arbitrary resampler* fourni par la librairie liquid-sdr. Cependant cela augmente le temps d'exécution et en pratique il est donc préférable de configurer le taux d'échantillonage de l'entrée RF avec un multiple de la bande passante des signaux adaptés.

Pour le reste, l'étape d'adaptation des canaux implémente un comportement similaire au bloc *Frequency Xlating FIR filter* de GNU Radio, que nous avons pour notre part implémenté à l'aide des librairies liquid-dsp et Volk [Vol].

4.3.4 Détection d'énergie dans le domaine temporel.

La détection d'énergie dans le domaine temporel intervient après l'adaptation du canal. À ce stade, les sections de signal selectionnées par le détecteur d'activité ont été adaptées et ne contiennent plus que la bande de fréquence du canal étudié.

Cependant, le détecteur d'activité des canaux ne possède qu'une faible résolution dans le domaine temporel, et les sections qu'il choisit comportent donc des périodes de marge à leurs extrémités, dont la taille est choisie en fonction de la valeur K utilisée pour lisser la PSD. Pour déterminer précisément les bornes du paquet ayant déclenché la détection, on doit donc effectuer une seconde étape de détection d'énergie, cette fois dans le domaine temporel.

Par ailleurs, les sections choisies par le détecteur d'activité sont susceptibles de contenir plus d'un paquet. Cela peut par exemple se produire lorsque l'intervalle inter-trame du protocole de notre canal est plus court que la **durée minimale d'une trame** (mfd), que l'on utilise pour déterminer K. Dans ce cas, la PSD sera moyennée sur une période trop longue pour distinguer la séparation entre deux paquets très rapprochés – par exemple une trame et son acquitement. Dans ce cas, notre détecteur en domaine temporel doit également pouvoir isoler les différents paquets contenu dans la section.

Le détecteur en domaine temporel recherche le début et la fin des transmissions, en observant les changements soudains dans l'amplitude du signal.

Etant donné le signal complexe X, son amplitude est calculée par (4.1) :

$$M_n = \mathrm{Re}(X_n)^2 + \mathrm{Im}(X_n)^2 \tag{4.1}$$

On élimine ensuite les fluctuations de court terme, en réalisant une moyenne mobile sur une période P qui agit comme un filtre passe-bas (4.2) :

$$\overline{M}_n = \frac{1}{P} \sum_{i=0}^{P-1} M_{n-i} \tag{4.2}$$

Finalement, les variations du SNR sont estimées par (4.3) :

$$\Delta S_n = 10 \times \log_{10} \left(\frac{\overline{M}_n}{\overline{M}_{n-P}} \right) \tag{4.3}$$

Le début et la fin d'une transmission sont identifiés en recherchant des pics positifs et négatifs dans la sortie, dont les valeurs absolues représentent les variations du SNR en dB.

Une fois ces bornes identifiées, le signal adapté peut être découpé en paquets avant d'être démodulé et décodé.

4.3.5 Démodulation

Après l'adaptation du canal et la détection d'énergie dans le domaine temporel, on obtient de courts paquets d'échantillons qui contiennent de probables transmissions sur notre canal. Ces paquets sont correctement filtrés et rééchantillonnés pour permettre leur démodulation. À l'heure actuelle, un seul module de démodulation est implémenté : *demod_fm*. Il gère les variantes des modulations *frequency shift keying* (FSK) et *phase shift keying* (PSK), dans lesquelles l'information est encodée par des variations de la fréquence.

Ce module nous suffit à démoduler les signaux Bluetooth Classic (BR phy), Bluetooth Low-Energy (1M phy) et 802.15.4 OQPSK [4].

Pour les démoduler, on calcule d'abord la fréquence instantanée du signal, ce qui revient à mesurer la variation de l'angle des échantillons successifs (4.4) :

$$if = arg(s(n) \times \overline{s(n-1)}) \times modGain \tag{4.4}$$

s représente le signal IQ [5], et $modGain$ est une constante de mise à l'échelle proportionnelle au taux d'échantillonage (sr) et à la déviation de fréquence attendue (df) (4.5) :

$$modGain = \frac{sr}{(2\pi \times dv)} \tag{4.5}$$

Au cours de la démodulation FM, nous calculons une première métadonnée propre au transmetteur. Il s'agit du décalage de la fréquence porteuse, ou en anglais *carrier frequency offset* (CFO). Les transmetteurs sont censés émettre sur des fréquences prédéfinies, mais en pratique, il existe toujours un décalage entre la fréquence attendue et celle que l'on observe. Ce décalage ne nuit pas aux performances du moment qu'il respecte une marge d'erreur spécifiée.

Pour estimer le CFO d'un paquet, on se repose sur une hypothèse sur les protocoles radio : les paquets sont précédés d'un préambule composé d'une suite alternée de 0 et de 1. Ce type de préambule est très couramment utilisé pour aider le récepteur à se synchroniser sur l'horloge du transmetteur.

En pratique, les signaux modulés ne sont pas représentés par des 0 et des 1, mais par des -1 et des 1. Notre préambule démodulé sera donc constitué d'une suite alternant des -1 et des 1. Dans nos signaux modulés en déplacement de fréquence, la valeur moyenne d'if pour le préambule devrait donc valoir zéro si le CFO est nul. En pratique, la valeur moyenne d'if sur le préambule est différente de zéro, et proportionnelle au CFO.

On estime le CFO sur le préambule (en Hertz) par (4.6) :

$$CFO = \frac{sr}{2\pi} \times \bar{if} \tag{4.6}$$

4. Les signaux 802.15.4 OQPKS peuvent être démodulés en MSK avec la méthode proposée dans [Sch06].

5. La barre supérieure désigne la conjugée complexe.

La dernière étape de démodulation a pour rôle de passer du train démodulé, qui pour l'instant représente toujours les variations de fréquences sous formes de nombres réels, à des bits.

Pour cela, nous devons d'abord synchroniser notre receveur avec l'horloge du transmetteur, afin de pouvoir échantillonner les bits au bon moment. Dans des sytèmes radio classiques, cette étape est faite par du matériel : le receveur utilise le préambule pour mesurer la différence entre son horloge et celle du transmetteur, puis décale son horloge pour échantillonner les symboles au bon moment.

Dans notre SDR, les signaux sont capturés bien avant leur démodulation, et ne peut donc pas décaler physiquement l'horloge qui gouverne l'échantillonnage. On peut par contre déplacer virtuellement l'échantillonnage en procédant à une interpolation. Nous interpolons le signal au moyen d'un filtre *Minimum mean square error* (MMSE) extrait de GNU Radio, et estimons l'erreur d'horloge en observant les points de passage à zero. Ce procédé ressemble beaucoup au bloc GNU Radio *clock recovery MM*, mais remplace la règle de *Mueller et Muller* par une règle *zero-crossing* qui nous a donné de meilleurs résultats.

Une fois l'horloge recouvrée, on peut découper les symboles qui composent le paquet par la règle suivante :

$$sym(n) = \begin{cases} -1, & \text{si } s(n) < 0, \\ 1, & \text{sinon} \end{cases} \tag{4.7}$$

Au cours de la démodulation, des données supplémentaires peuvent être journalisées. On peut ainsi sauvegarder les échantillons du paquet, le train démodulé if, et les symboles démodulés. Comme ces informations peuvent être plus volumineuses, leur sauvegarde dépend de paramètres spécifiés par l'utilisateur dans le fichier de configuration de la couche physique.

Décodage

Le décodage est la dernière étape de traitement prise en charge par notre outil. Ce traitement est le plus spécifique au protocole, et notre outil contient donc trois modules différents de décodage qui s'appliquent au Bluetooth Classic, Bluetooth Low-Energy, et 802.15.4.

Le décodage n'a pas vocation à être exhaustif. Il ignore le contexte de réception des paquets et ne permet donc pas toujours de recouvrer le contenu des paquets. Si on souhaite analyser tout le contenu des paquets, on peut en revanche assez simplement écrire des outils annexes, qui prendront en entrée les paquets démodulés sauvegardés par notre outil.

De la même manière que les processus précédents, le rôle d'un décodeur est d'isoler des métadonnées utiles pour enrichir notre compréhension du contexte réseau sans fil.

Ainsi, le décodeur Bluetooth Classic peut détecter le mot de synchronization d'un paquet Bluetooth pour en extraire les 24 bits du *lower-address part* (LAP). En plus d'identifier le piconet Bluetooth, l'identification du LAP nous permet de distinguer encore plus précisément le moment d'arrivée d'un paquet.

Le décodeur Bluetooth Low-Energy ne peut décoder que les paquets transmis sur les canaux d'*advertisement* Bluetooth, qui servent notamment à l'établissement des connexions. En effet, les paquets transmis sur ces canaux utilisent des paramètres connus. On peut ainsi détecter l'*adresse d'advertisement* prédéfinie [6], pour connaitre précisément l'heure d'arrivée du paquet, puis retirer le *whitening* du message pour recouvrer son contenu, et vérifier son CRC [7]. Encore une fois, il est possible d'implémenter un décodeur plus complet, conscient du contexte, en exploitant les paquets démodulés journalisés précedemment.

Le décodeur 802.15.4 est le plus complet car le décodage de ces paquets ne requiert aucun contexte. Le décodeur peut donc d'abord détecter le *start frame delimiter* présent au début du paquet pour déterminer son temps précis d'arrivée, puis désétaler le reste du paquet par la méthode de Schmid [Sch06], et enfin vérifier son CRC. Toutes ces métadonnées, ainsi que le contenu du paquet peuvent aussi être journalisées.

4.4 Conclusion

Dans cette partie, nous avons présenté l'architecture et l'implémentation d'une SDR pour capturer des paquets de multiples protocoles sur une large bande de fréquences.

En plus de permettre efficacement la capture des paquets des protocoles qui nous intéressaient, cet outil permet de conserver tous les métadonnées calculées par les différentes étapes de traitement. Ces métadonnées nous permettent de comprendre rapidement la nature des données que l'on observe, mais aussi de distinguer les transmetteurs par des indicateurs simples comme le SNR des paquets, et leur CFO.

C'est sur cet outil que nous avons basé la suite de nos travaux. D'abord en étudiant des indicateurs pour le *fingerprinting* de transmetteurs Bluetooth, puis en utilisant *Phyltre* pour déboguer les transmissions de notre *baseband* Bluetooth UBTBR.

6. Avec une valeur de 0x8e89bed6
7. Dont la valeur d'initialisation est fixée à 0x555555

Chapitre 5

Fingerprinting sur la couche physique du Bluetooth Classic

Dans le chapitre 4, nous avons présenté *Phyltre*, qui nous permet de récupérer rapidement des informations pertinentes et condensées sur les transmissions sans fil environnantes sur des canaux prédéterminés.

La capture de transmissions sans fil au moyen de SDR permet de récupérer plus d'informations que si on avait utilisé des récepteurs conventionnels, grâce à ces informations on peut envisager de mettre en œuvre des techniques de *fingerprinting* sur la couche physique.

Le *fingerprinting* de transmetteurs sans fil est un ensemble de méthodes pour distinguer différents transmetteurs en observant des différences dans la façon qu'elles ont d'implémenter ce protocole. Ces informations peuvent apparaître dans les couches logiques : identifiants matériels, valeurs spécifiques de certains paramètres, etc. Mais aussi au niveau de la couche physique, où les imperfections dans un signal peuvent révéler l'identité de leur transmetteur.

Nous avons utilisé les données capturées à l'aide de *Phyltre*, pour rechercher des traits caractéristiques pouvant permettre d'identifier un transmetteur Bluetooth en n'utilisant que des informations observables au niveau de la couche physique.

Nous présentons une expérience de caractérisation du modèle d'un transmetteur Bluetooth, par l'observation passive des transmissions qui ne contiennent pas d'identifiant logique. Après avoir précisé les conditions de notre expérience, nous présentons plusieurs traits distinctifs qu'il est possible de mesurer avec les données issues de Phyltre, et évaluons une classification automatique des transmetteurs.

Enfin, pour mieux comprendre les limites de ces identifiants physiques, nous avons évalué l'effet de la température sur le décalage de fréquence mesuré sur un transmetteur Bluetooth. Ces résultats ont été publiés dans [HLBGH20].

5.1 Travaux reliés

Le *fingerprinting* sur la couche physique est un sujet de plus en plus étudié depuis une vingtaine d'années. Ces méthodes permettent d'identifier l'emetteur d'un paquet radio, en l'observant au niveau de sa couche physiquee.

Nous distinguons deux grandes approches de fingerpriting sur la couche physique. Les approches basées sur la forme d'onde, et celles basées sur la modulation.

Forme d'onde. Les méthodes basées sur la forme d'onde peuvent s'appliquer à tous types de protocoles sans fil, et permettent de distinguer précisément des modèles de transmetteurs différents, et peuvent parfois même distinguer des transmetteurs du même modèle. L'exemple le plus répandu d'analyse sur la forme d'onde est le fingerpriting du signal transitoire d'allumage (*transient*). Cette méthode consiste à observer l'évolution de l'amplitude du signal lorsqu'il commence à transmettre. En 2006, Hall et al. [HBK06] étudient le transient d'allumage d'appareils Bluetooth, pour permettre la création d'IDS basés sur la détection d'anomalie. Ils mesurent l'amplitude et la phase instantanée du signal transitoire à un taux d'échantillonnage de 500 Msps, et en extraient quinze caractéristiques. Sur la base de ces caractéristiques, il utilisent une classification statistique pour distinguer le modèle des transmetteurs, et obtiennent un taux de détection compris entre 85 et 100% selon le modèle. En 2019, Ali et al. [AUK19] poursuivent une approche similaire sur le Bluetooth, avec cette fois ci un taux d'échantillonnage de 20 Gsps. Ils identifient des caractéristiques permettant de caractériser différents modèles d'appareils, et dans certains cas à différencier des appareils du même modèle.

Le *fingerprinting* du signal transitoire d'allumage reste difficile à mettre en oeuvre en pratique, car elle nécessite un grand nombre d'échantillons pour une très courte période, et qui nécessite des équipements de mesure coûteux, et importante puissance de calcul. De plus, les analyses basées sur l'amplitude du signal sont plus sensibles au bruit ambiant que celles basées sur la modulation.

Dans ce chapitre, nous avons utilisé une caractéristiques nécessitant d'étudier l'amplitude du signal, mais elle peut être mesurée avec un taux d'échantillonnage considérablement réduit. C'est la *durée du préambule*, présentée dans la section 5.2.2.

Modulation. Les signaux modulés doivent respecter les paramètres définis par une spécification, mais une marge d'erreur est permise. Le fingerpriting basé sur la modulation repose sur l'observation de ces erreurs mesurées au cours de la démodulation du signal. Ces méthodes sont moins précises que celles basées sur la forme d'onde, et ne sont généralement utilisées que pour caractériser différents modèles de transmetteurs. Elles ont cependant l'avantage de pouvoir être mises en oeuvre avec des taux d'échantillonnage beaucoup plus modestes, de l'ordre de celui nécessaire à la démodulation du signal. En 2018, Zhuang présente FBSleuth [ZJZ+18], qui vise à détecter des fausses stations de base GSM. Ils utilisent une partie fixe contenue dans les signaux GSM, pour pouvoir mesurer ses différences avec un signal idéal. Il observent que l'erreur de phase, l'erreur de fréquence, et le

décalage IQ sont différents dans leurs appareils. Ils parviennent à distinguer six antennes relais de modèles différents avec une précision supérieure à 98%. Le signal est échantillonné à une fréquence de 25 Msps, ce qui rend cette approche réalisable avec un matériel à coût réduit.

Dans ce chapitre, nous observons l'erreur de fréquence sur des transmetteurs Bluetooth, et montrons que même à un faible taux d'échantillonnage, elle permet de distinguer différents modèles d'appareils.

Une autre approche, présentée par Huang en 2014 dans BlueID [HAX14], nécessite encore moins de moyens. Ils utilisent un Ubertooth-One (qui coûte environ 100$), pour calculer le décalage de l'horloge d'appareils Bluetooth, en se basant sur le temps d'arrivée des paquets. Ils peuvent limiter l'erreur de mesure à 0.1ppm en observant 25 secondes de trafic de données, ou 65 secondes de trafic audio. Finalement, ils utilisent une classification automatique pour distinguer 56 modèles d'appareils avec une précision de 94.3%.

Dans ce chapitre, nous avons utilisé une SDR pour mesurer le décalage de l'horloge d'appareils Bluetooth, et pu reproduire les résultats de Huang. Enfin, nous montrons que le décalage de l'horloge et l'erreur de fréquence ont en fait la même valeur sur certains transmetteurs Bluetooth.

5.2 Expérience sur l'*inquiry* Bluetooth Classic

Nous avons choisi la procédure d'*inquiry* pour évaluer la collecte d'empreintes sur la couche physique Bluetooth. Cette procédure sert à effectuer un *scan* Bluetooth, pour découvrir les appareils à proximité.

Au cours d'une procédure d'*inquiry*, l'initiateur du scan envoie un grand nombre de paquets *ID*, sur diverses fréquences. Ces *ID* contiennent l'identifiant réservé pour les scans, le *General Inquiry Access Code* (GIAC). L'*Access Code* est le seul champ variable dans un paquet ID. Donc tous les *ID* transmis au cours d'un scan contiennent exactement les mêmes bits.

Cette procédure est donc un cas parfait pour le *fingerprinting* sur la couche physique, car tous les paquets ont le même contenu, et les seules différences observables proviennent de la couche physique.

5.2.1 Cadre expérimental

Nous avons collecté les paquets Bluetooth en utilisant Phyltre et une LimeSDR-mini, avec un taux d'échantillonnage de 28 MHz, afin d'observer une portion significative des canaux Bluetooth [1]. La fréquence centrale a été fixée à 2420 MHz pour éviter les canaux Wi-Fi voisins et ne couvrir que le spectre Bluetooth.

Les appareils testés ont été placés à quatre mètres de l'antenne du récepteur. Nous avons déclenché une procédure d'*inquiry* pour chaque appareil (en démarrant un scan Bluetooth),

1. Nous avons observé les canaux 5 à 32, soit 28 canaux sur 79.

et collecté les paquets à l'aide de *Phyltre*. Nous avons réalisé trois captures différentes par appareil.

Durant une procédure d'*inquiry*, l'initiateur transmet le même paquet *ID* encore et encore à un rythme constant. Au lieu d'analyser un seul paquet, nous pouvons donc exploiter des séries de paquets pour évaluer les traits distinctifs avec plus de précision. Pour travailler sur des séries de paquets, nous devons d'abord nous assurer qu'ils ont tous été émis par le même appareil. Cette tâche n'est pas sans importance, car elle est intimement liée au problème sur lequel nous travaillons.

Heureusement, on peut y parvenir sans trop de difficultés avec la procédure d'*inquiry*. D'abord, les paquets envoyés sur un même *piconet* Bluetooth peuvent facilement être associés à ce piconet, car ils sont tous synchronisés sur l'horloge du maître, et apparaissent donc à des intervalles fixes. Deuxièmement, la procédure d'*inquiry* est une situation particulièrement idéale pour distinguer les transmetteurs : les paquets *ID(GIAC)* sont utilisés uniquement durant la procédure d'*inquiry*, et l'initiateur de la procédure n'utilise pas les mêmes canaux que les répondeurs. Nous pouvons considérer que tous les paquets *ID(GIAC)* sur ces canaux sont envoyés par l'initiateur de la procédure.

Pour isoler ces paquets, nous avons réalisé un script Python qui parcourt les enregistrements TLV réalisés par Phyltre, et ne conserve que les paquets préfixés par le GIAC, ayant une durée inférieure à 100 µs [2].

5.2.2 Traits selectionnés

Nous avons recherché des traits distinctifs permettant d'identifier des appareils Bluetooth en n'utilisant que des informations observables sur la couche physique, collectées à l'aide de Phyltre.

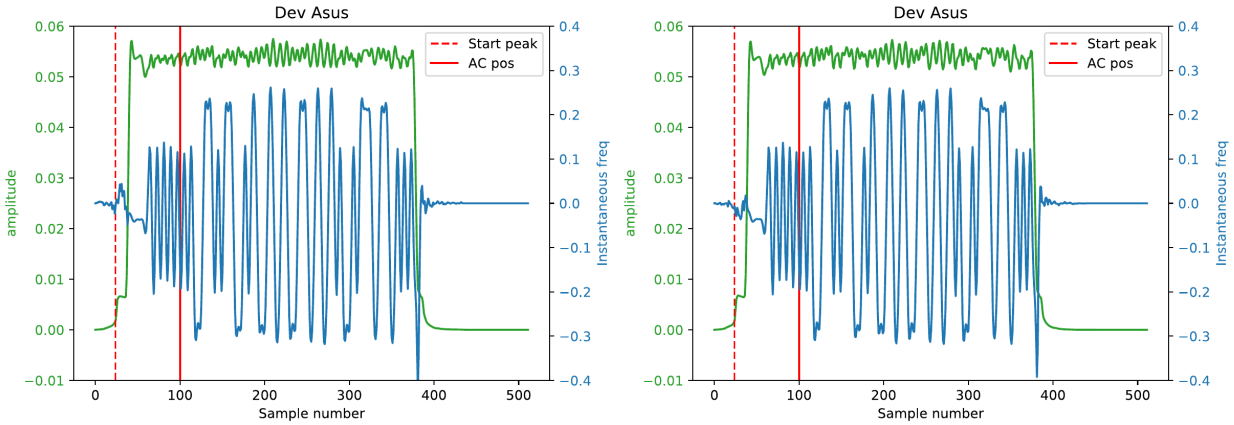
Phyltre permet de réduire le volume des informations au fil de leur traitement. Pour conserver cet avantage, il était donc important que les traits caractéristiques que nous utilisons aient une dimension réduite, ce qui facilite leur traitement et leur stockage.

Par ailleurs, nous souhaitions pouvoir comparer facilement les mesures en utilisant différentes configurations de Phyltre et différents matériels. Il fallait donc que les mesures soient exprimées dans des unités de mesures consistentes.

Nous avons mesurés des traits distinctifs déjà proposés dans la littérature : les décalages d'horloge du saut de fréquence [HAX14], et de la fréquence porteuse [ZJZ+18]. Nous montrons par ailleurs que la durée du préambule est un critère utile pour distinguer le constructeur d'un modem Bluetooth.

Les figures 5.1a et 5.2 représentent les données de deux appareils : un Asus BT400 et un dongle anonyme à bas coût. On peut voir que la durée du préambule est différente pour ces appareils. À l'inverse, la durée du préambule est la même sur les figures 5.1a et 5.1b qui représentent deux traces distinctes du même Asus BT400. La durée du préambule est donc un bon candidat pour le *fingerprinting*.

2. Les paquets ID durent environ 72 µs. Tous les autres paquets durent 122 µs ou plus.



(a) Trace d'amplitude et de fréquence accumulées de 1464 paquets ID (Asus BT-400)

(b) Autre scan avec 1734 paquets ID (même Asus BT-400)

Décalage de l'horloge de saut de fréquence.

Les transmissions Bluetooth suivent une séquence de saut de fréquence basée sur l'horloge de l'initiateur. Toutes les transmissions sont censées commencer au début d'un *créneau*[3]. La fréquence des sauts peut être de 1600 ou 3200 sauts par seconde selon la procédure et les capacités de l'appareil. Comme dans BlueID [HAX14], nous mesurons le décalage de l'horloge de saut, en comparant le temps d'arrivée de paquets ID successifs. Pour raffiner les mesures, nous utilisons le *temps du code d'accès* (AC pos dans la figure 5.1a), qui est obtenu depuis la trace démodulée et est donc plus résistant au bruit que le détecteur de pics.

Nous définissons le décalage de l'horloge de saut de fréquence ainsi : tous les paquets doivent être alignés sur la durée d'un créneau, noté $Hdur = \frac{1}{3200} sec.$ $Tcur$ et $Tprev$ sont les temps d'arrivée de deux paquets consécutifs.

Le délais attendu $Texp$ est l'intervalle arrondi à l'échelle d'un créneau ($Hdur$) :

$$Texp = Hdur \times \left\lfloor \frac{Tcur - Tprev}{Hdur} \right\rceil$$

Le décalage de l'horloge de saut de fréquence (*Hopping Clock Skew*) HCS est exprimé comme :

$$HCSppm = 1\text{M} \times \left(\frac{Texp}{Tcur - Tprev} - 1 \right)$$

en partie par million (ppm[4]).

Lorsque nous utilisons Phyltre en mode hors-ligne, c'est à dire en sauvegardant d'abord la capture brute dans un fichier avant de l'analyser, les échantillons ne comportent pas de *timestamp*. On doit alors se référer au nombre d'échantillons pour déterminer le temps

3. *time slot*
4. Les décalages d'horloge sont habituellement mesurées en parties par million ou milliard

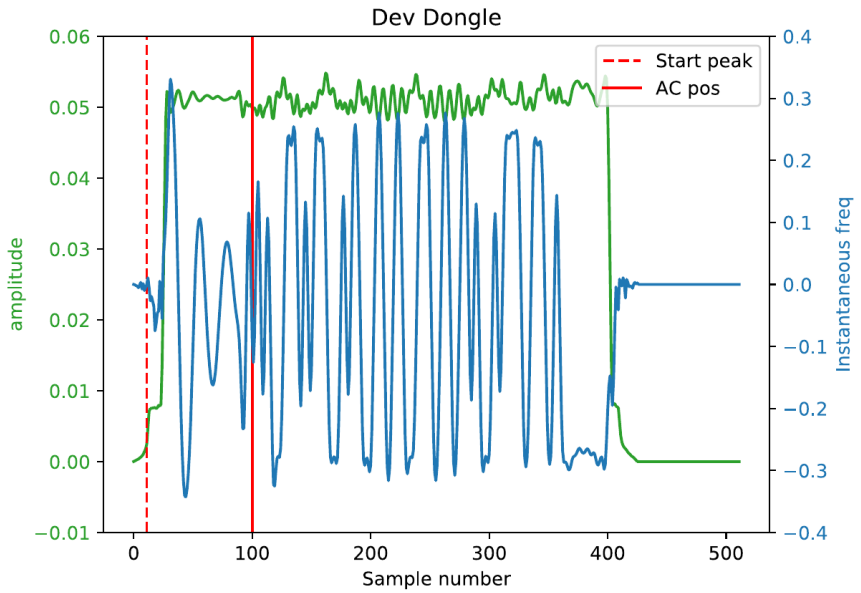


FIGURE 5.2: Trace d'amplitude et de fréquence accumulées de 1082 paquets ID (Dongle BT anonyme)

d'arrivée d'un paquet. Mais les captures peuvent être incomplètes si des échantillons ont été perdus. Dans ce cas, on observe parfois des HCS aberrants entre deux paquets. Pour éliminer ces paquets, nous avons fixé un seuil à 100 ppm. Les valeurs excédant ce seuil sont ignorées. Au cours de nos expériences, ce problème s'est rarement produit (nous avons ignoré 7 mesures sur 47184). Cette correction n'est pas nécessaire lorsqu'on effectue la capture en ligne avec Phyltre, avec un matériel fournissant des *timestamp* précis.

Dans cette expérience, les paquets Bluetooth étaient échantillonnés à un taux de 4 Msps, pour stocker quatre échantillons par symbole comme dans [SB07]. Pour une paire de paquets, nous pouvions donc mesurer le HCS avec une résolution de 250 ns. Pour obtenir une résolution supérieure, nous prenons la moyenne des mesures.

Décalage de la fréquence porteuse.

Le protocole Bluetooth exploite 79 canaux, larges de un MHz chacun. Nous avons mesuré le décalage entre la fréquence du canal et la fréquence des paquets effectivement reçus.

Ce décalage est habituellement mesuré en Hertz dans la littérature, et appelé *Carrier Frequency Offset* (CFO). Pour conserver la même unité de mesure que celle utilisée pour l'horloge de saut de fréquence, nous parlons plutôt de *Carrier Clock Skew* (CCS) que nous exprimons en ppm.

Pour réaliser cette mesure, nous avons utilisé une formule de calcul du CFO plus précise

que celle de Phyltre, mais moins générique [5]. Elle s'applique sur l'intégralité du paquet ID détecté :

$$\Delta F = \frac{4\text{M}}{2\pi} \times (\overline{IF} - \overline{idealIF})$$

$$CCSppm = 1\text{M} \times \frac{\Delta F}{ChannelFreq}$$

IF est notre signal démodulé en fréquence en radians par seconde, et idealIF est le signal *de référence* correspondant. IdealIF a été obtenu en modulant un paquet ID(GIAC) à l'aide de GNU Radio. ΔF représente le CFO en Hertz, et CCSppm le décalage de l'horloge de la fréquence porteuse en ppm.

Le CCS tient compte de tous les échantillons constitutifs du paquet. L'estimation est donc plus précise que celle du HCS car nous avons un grand nombre d'échantillons à notre disposition dans chaque paquet.

Nous avons remarqué que presque tous les appareils testés exhibaient un décalage de fréquence constant, avec des valeurs comprises entre ±22 KHz autour de la fréquence centrale. Les décalages mesurés sont resté stables au cours des différentes mesures.

Les auteurs de BlueID [HAX14] avaient également remarqué que les mesures du HCS étaient stables pour chaque modèle d'appareil. Pour mieux comprendre les limites de cette approche, nous avons également mesuré l'effet de la température sur le CCS dans la section 5.3.2.

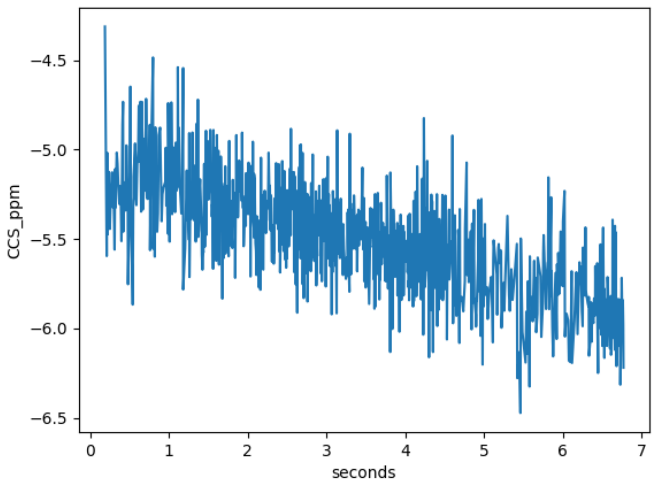


FIGURE 5.3: CCS d'un Dongle anonyme sur un scan complet

Un seul appareil a montré une variabilité dans les mesures du CCS et de l'HCS. Dans nos expériences sur le dongle Bluetooth anonyme low-cost, nous avons observé que le CCS et le HCS dérivaient au fil du temps (comme le montrent les figures 5.3 et 5.4). De fait, le

5. Phyltre était configuré pour conserver les échantillons bruts des paquets

rythme de cette variation constitue un trait caractéristique de cet appareil. Ce n'est pas très surprenant puisque nous recherchons des imperfections dans les transmissions, et les appareils de moindre qualité sont naturellement plus susceptibles d'en comporter.

Durée du préambule.

Un préambule de synchronisation est généralement une séquence de bits alternés, précédant un paquet pour permettre la synchronisation d'horloge par le récepteur. Beaucoup de protocoles requièrent la présence d'un tel préambule, et le Bluetooth ne fait pas exception. Le nombre de bits du préambule est fixé à quatre dans la spécification Bluetooth [6]. Cependant, nous avons observé que le nombre exact de bits transmis varie selon les équipements Il existe un court délai entre l'allumage du transmetteur – et donc la hausse d'énergie sur le canal – et la transmission effective de données modulées [7].

Nous avons observé que les appareils Bluetooth que nous avons testés exhibent différents comportements durant le préambule. Le nombre et la qualité des bits de synchronisation ne sont pas toujours les mêmes. Le délai entre la hausse d'énergie et le premier bit de synchronisation peut également varier.

Pour prendre en compte à la fois la période d'allumage et le nombre de bits du préambule, nous appelons *durée du préambule* l'intervalle de temps entre la hausse de l'amplitude qui indique le début d'un paquet (*Start peak* sur la figure 5.1a) et la position réelle de l'*Access Code* (AC) dans le signal démodulé (*AC pos* sur la figure 5.1a).

5.3 Évaluation

La première partie de cette section présente les observations que nous avons faites en analysant manuellement le jeu de données. La seconde partie étudie une technique de classification automatique, pour distinguer automatiquement le type d'un transmetteur parmis 11 classes différentes.

5.3.1 Observation des caractéristiques

D'abord, nous avons remarqué que pour certains appareils, les valeurs du HCS et du CCS semblent concorder. Nous avons également observé que la durée du préambule permet de déterminer le constructeur de l'appareil. Finalement, nous débattons de l'usage d'un grand nombre de paquets pour mesurer les traits caractéristiques avec une SNR élevé.

Relation entre les décalages d'horloge.

La figure 5.4 montre les décalages d'horloge de saut et de la fréquence porteuse (HCS et CCS) en ppm pour dix appareils. La ligne bleue dénote l'identité. On peut voir que

6. Bluetooth Core v5, Vol 2, Part B, 6.3.2 : Preamble
7. Cette courte période est extensivement étudiée dans la littérature sur le *transient fingerprinting*

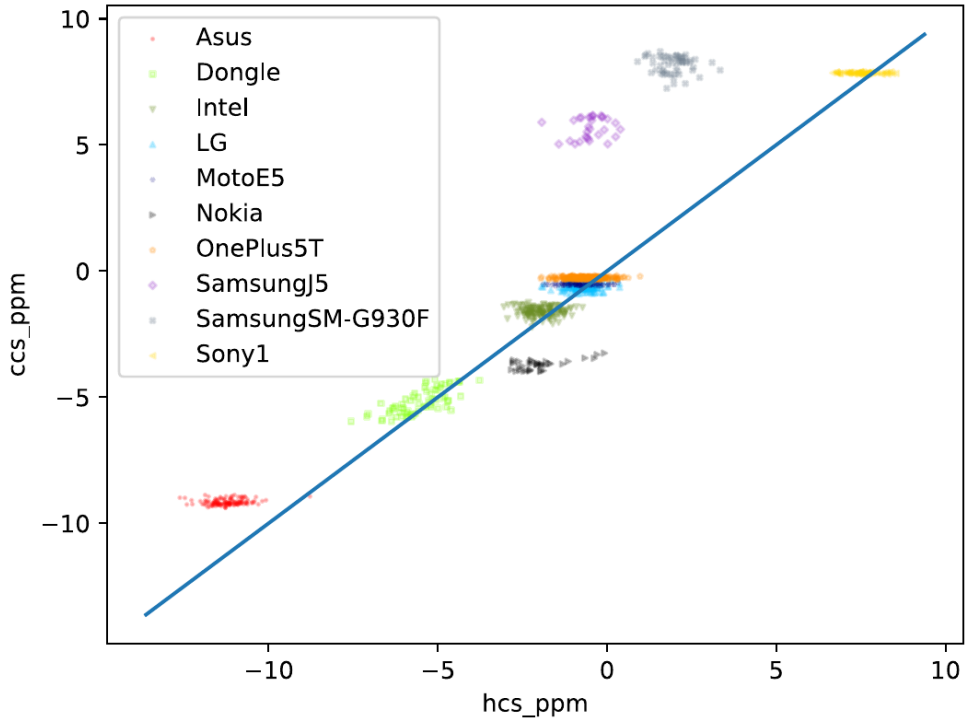


FIGURE 5.4: HCS vs. CCS sur des ensembles de 50 paquets.

pour la majorité des appareils, le HCS et le CCS sont très proches. Pour les deux appareils Samsung testés (basés sur un chipset Exynos), les deux décalages d'horloge sont clairement découplés, avec un CCS proche de 5 ppm, et un HCS plus faible, de ±0.5 ppm.

Nous supposons que dans les appareils qui apparaissent sur la ligne bleue, les périphériques Bluetooth – qui génèrent la fréquence porteuse et l'horloge de saut – sont gouvernés par un même oscillateur. Dans d'autres architectures, les deux périphériques pourraient être basés sur deux oscillateurs différents, ou pourraient subir des corrections additionnelles. Il faudrait analyser plus en profondeur l'architecture de ces appareils pour être en mesure de confirmer ou d'infirmer cette hypothèse.

Durée du préambule et constructeur

La figure 5.5 représente la durée du préambule en abscisse, et le CCS en ordonnée. Les durées de préambule sont rassemblées dans deux groupes principaux : un autour de 16,5 µs, et l'autre autour de 18,7 µs. Le dongle anonyme est le seul dont la durée du préambule se situe autour de 22 µs. En fait, tous les appareils situés dans le groupe de gauche sont des téléphones basés sur un chipset Qualcomm, tandis que les appareils du groupe du centre possèdent un chipset Bluetooth Broadcom.

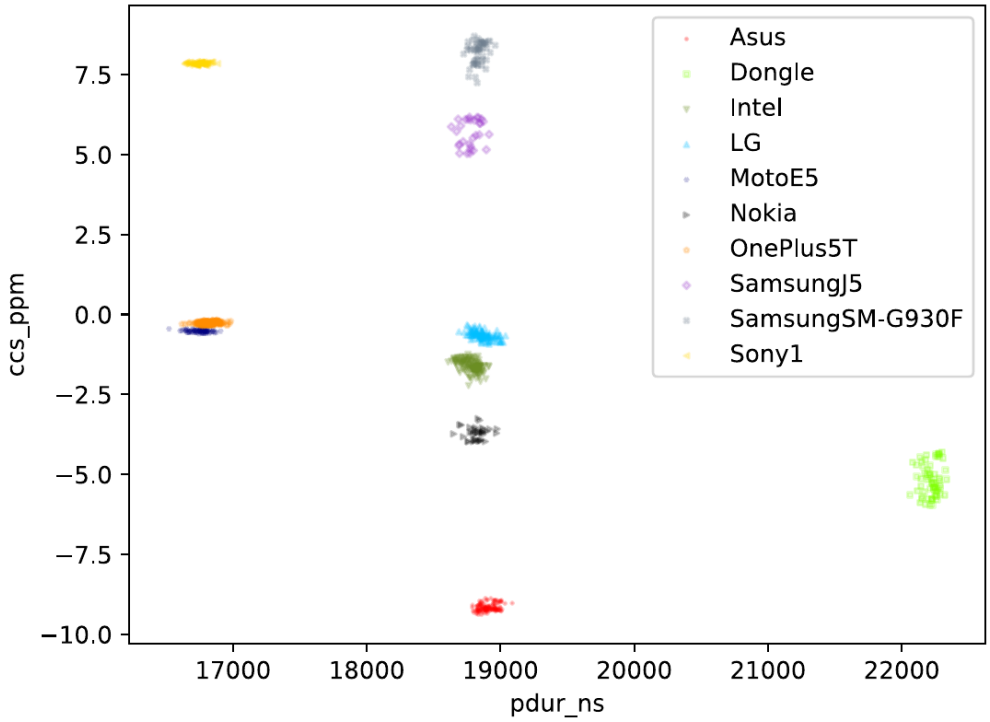


FIGURE 5.5: Durée du préambule vs. CCS sur des ensembles de 50 paquets

Nous en concluons que la durée du préambule seule peut être un élément suffisant pour distinguer des appareils de différents constructeurs.

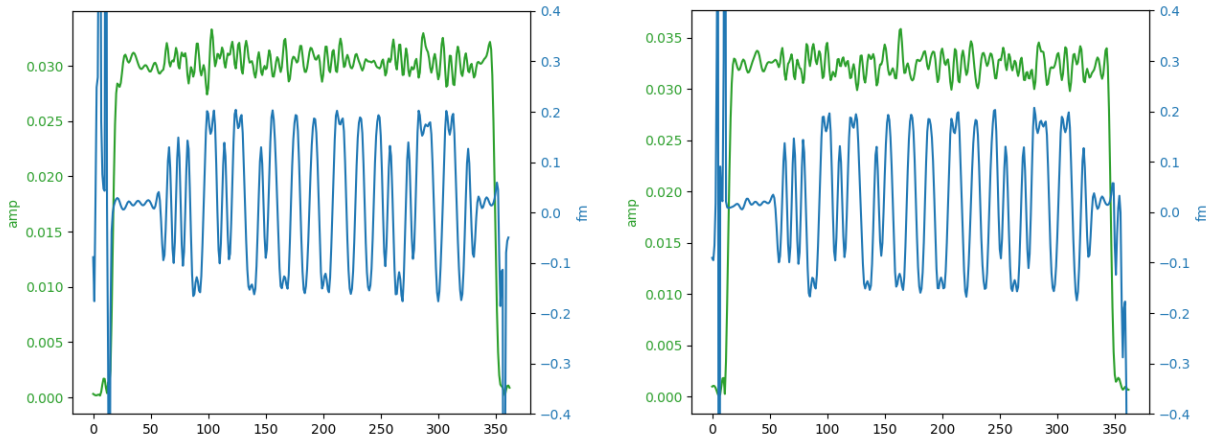
Images à haut SNR

Les figures 5.1a à 5.2 ont été obtenues en accumulant les traces de tous les paquets collectés au cours de procédures d'*inquiry* pour un adapteur Asus BT-400 et un dongle Bluetooth anonyme.

Accumuler les signaux d'un grand nombre de paquets augmente grandement le rapport signal sur bruit (SNR), ce qui améliore la précision des traits caractéristiques. Par exemple, on peut voir que le dongle anonyme de la figure 5.2 envoie un préambule pour le moins singulier et néglige d'envoyer les bits de fin du paquet.

Cette méthode peut se révéler utile pour exploiter des traits supplémentaires. Par exemple, en observant les trois traces accumulées de chaque appareil, nous remarquons que le vecteur d'amplitude seul est également distinctif de chaque modèle. C'est par exemple visible dans les figures 5.1a à 5.2.

À titre de comparaison, les figures 5.6a et 5.6 présentent des traces simples de deux paquets consécutifs capturés dans une même procédure d'*inquiry* avec le téléphone Sony1.



(a) Paquet 1 (b) Paquet 2

FIGURE 5.6: Trace d'amplitude et de fréquence pour deux paquets consécutifs (Sony1)

Le signal démodulé en fréquence semble identique d'un paquet à l'autre. En revanche, les courbe qui représentent l'amplitude du signal apparaissent très différentes, et donc peu distinctives lorsque l'on considère des paquets uniques.

Nous n'avons pas utilisé le vecteur d'amplitude dans nos expériences de classification. En effet, il faut accumuler un grand nombre de paquets pour le mesurer précisément. Pour créer un jeu d'apprentissage incluant ce vecteur, nous aurions donc eu besoin de réaliser beaucoup plus de captures par appareil.

Classification

Nous avons montré qu'il était possible de distinguer les appareils en visualisant leurs traits caractéristiques. Cependant, dans un IDS cette tâche devrait être automatisée. Nous avons réalisé une expérience de classification automatique sur notre jeu de données.

Ce jeu de données comprend trois captures d'une procédure d'*inquiry* pour chacun des dix appareils. Chaque capture est composée d'au moins 1500 paquets bruts. Nous pouvons générer un vecteur (HCS, CCS, durée du préambule) pour chaque paquet, mais pour estimer plus précisément ces traits nous avons utilisé des ensembles de paquets consécutifs et généré un vecteur moyen (HCS, CCS, durée du préambule) par ensemble. Pour chaque test, nous avons selectionné aléatoirement une série de 1500 paquets d'une même capture. Ces séries sont découpées en ensembles de taille N, et un vecteur de traits moyens est calculé pour chaque ensemble. Donc plus la taille N d'un ensemble est grande, moins nous avons de vecteurs différents pour entraîner le classifieur.

Nous avons comparé les performances de plusieurs classifieurs fournis par la librairie Python scikit-learn [ea11] : *logical regression*, *multi-layer perceptron* (MLP), *random-forest* (RF), et *support-vector classifier* (SVC). Nous avons évalué leurs performances avec une une validation croisée à k=5 blocs.

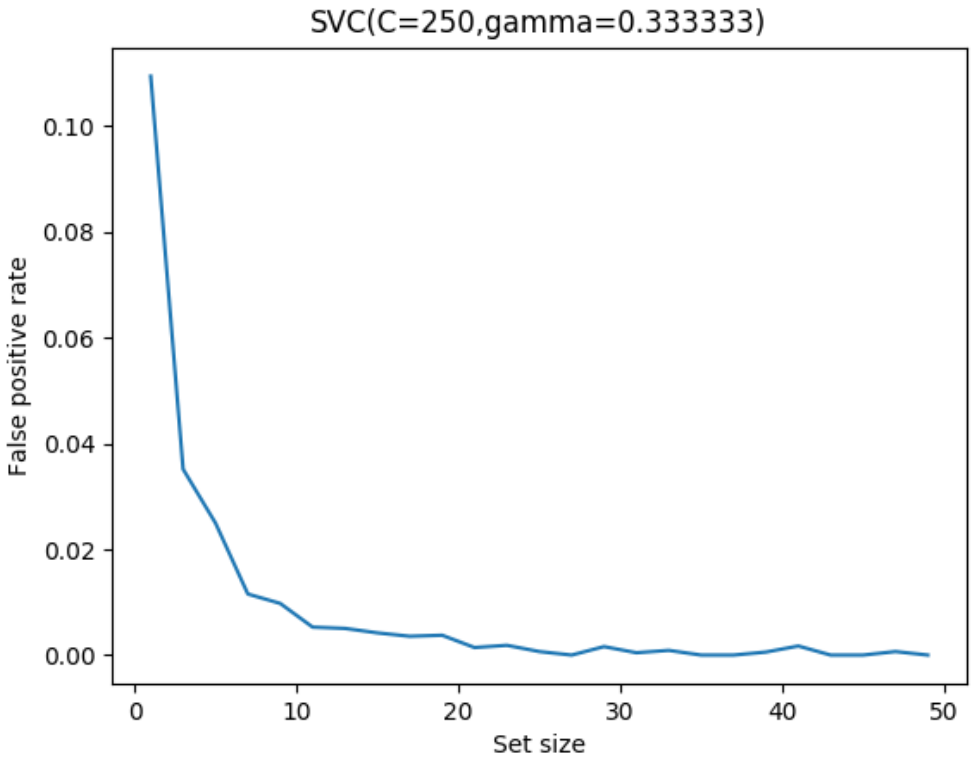


FIGURE 5.7: SVC FPR vs. taille d'un ensemble

La configuration qui a produit les meilleurs résultats est un SVC avec $C = 250$ et $gamma = 0,33$.

Nous avons mesurés les performances du SVC en faisait varier la taille de l'ensemble utilisé pour estimer les traits caractéristiques.

La figure 5.7 présente la proportion de faux positifs en fonction du nombre de paquets utilisés pour estimer les traits caractéristiques. Le classifieur fournit de meilleurs résultats lorsqu'on augmente la taille d'un ensemble et obtient une précision supérieure à $99,8\%$ lorsque les ensembles contiennent vingt paquets ou plus.

Une procédure d'*inquiry* provoque l'émission de plusieurs milliers de paquets ID, nous sommes donc convaincus que même en ne capturant qu'une partie de ces paquets, nous obtiendrions suffisamment de données pour caractériser le modèle de leur transmetteur.

5.3.2 Effet de la température

Dans notre expérience de classification automatique, nous avons utilisé le décalage de la fréquence porteuse (CFO) comme critère discriminant. Nous savons que la température d'un oscillateur à quartz affecte sa fréquence de résonance [HAX14]. Nous avons évalué cet effet sur un des appareils de notre échantillon (*Sony1*).

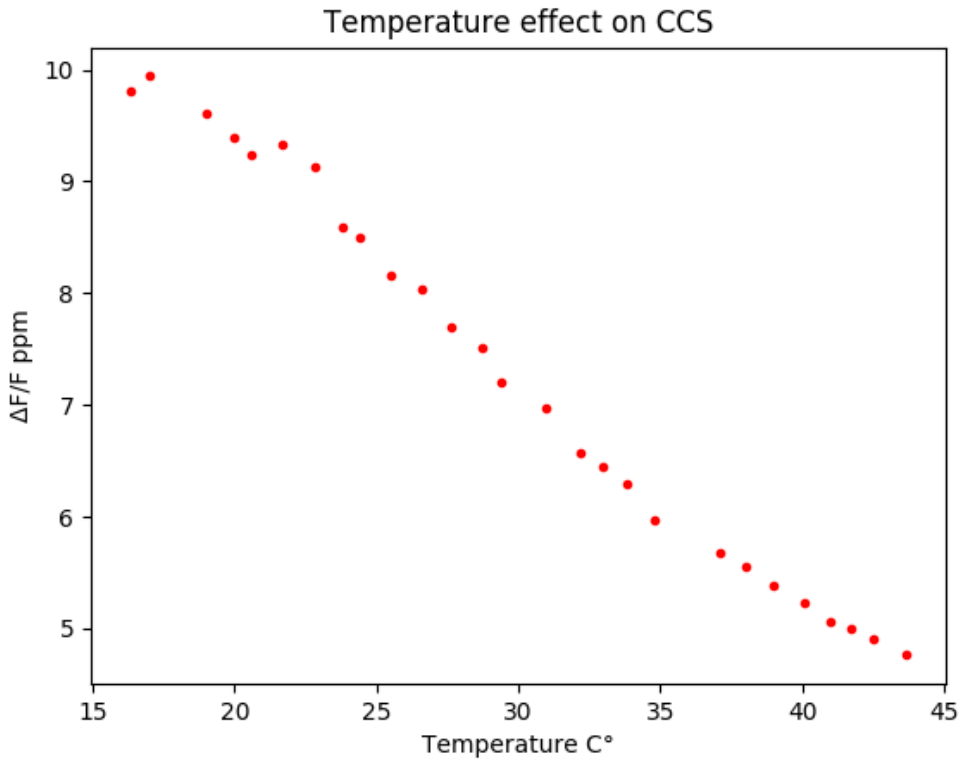


FIGURE 5.8: Effet de la température sur le CCS d'un téléphone Sony

Nous avons choisi de modifier directement la température de l'appareil pour obtenir une compréhension plus claire du phénomène. Pour cela, nous avons utilisé une plaque thermique permettant de réchauffer ou de refroidir l'objet, ainsi qu'une sonde thermique placée au niveau de la batterie de l'appareil pour mesurer précisément sa température. Pour chaque mesure, une procédure d'*inquiry* a été initiée, et tous les messages contenant un GIAC ont été enregistrés. Dans la figure 5.8, chaque point représente le CCS mesuré sur l'ensemble des paquets enregistrés.

Dans la plage de température observée, la fréquence décroit linéairement à mesure que la température augmente. Ce n'est pas surprenant, car les oscillateurs à quartz sont conçus pour minimiser la dépendance en température autour de la température ambiante (définie à 25 °C). Autour de 25 °C, la fréquence varie donc presque linéairement selon la température [ZNKS08].

5.3.3 Analyse.

Nous avons observé que pour l'appareil selectionné (*Sony1*), le CCS et le HCS restent les mêmes au cours de l'expérience. Cela accrédite notre hypothèse selon laquelle un oscillateur commun génère l'horloge de la fréquence porteuse et celle de saut de fréquence.

S'agissant des traits caractéristiques, la température a un effet important sur la fréquence mesurée : près de 5 ppm de variation sur une plage de 30 °C, pour une valeur absolue de 8 ppm à 25 °C.

L'écart que nous mesurons est plus élevé que celui rapporté par Huang dans BlueID [HAX14], qui mesurait une augmentation du HCS de moins de 0.5 ppm après avoir réduit la température de 70 °F à 35 °F (soit environ 21 °C à 2 °C).

La discordance entre nos résultats et ceux de Huang peut provenir de l'approche différente que nous avons utilisé pour réaliser la mesure : modifier la température de l'appareil plutôt que celle de la pièce. Nous supposons que dans l'expérience de Huang, la température de l'appareil n'était pas aussi basse que celle de la pièce, possiblement à cause de la chaleur dissipée par l'appareil. Selon nos résultats, le décalage de fréquence induit par une variation de la température pourrait être bien plus élevé, par exemple lorsqu'on allume un appareil qui a été maintenu éteint suffisamment longtemps dans un environnement froid.

Nous en concluons qu'une solution de *fingerprinting* devrait utiliser des modèles physiques pour tenir correctement compte de l'effet de la température. Pour ce faire, Huang proposait de suivre le changement graduel du décalage de fréquence dans le temps. Cette approche pourrait être améliorée en vérifiant que le décalage de la fréquence mesuré coincide avec le profil de température de l'appareil observé. Par exemple, une augmentation de la température devrait provoquer une réduction de la fréquence pour le téléphone Sony1 dans un environnement tempéré.

Cette expérience montre aussi que les solutions de *fingerprinting* sans fil basés sur le CFO pourraient être attaqués à plus faible coût que ce qu'on pensais auparavant. Par exemple, Huang considérait par exemple que seule une modification du *firmware* d'un transmetteur BT permettrait de modifier son HCS. Par ailleurs, nous savons que des signatures radiométriques peuvent être mimées par un attaquant utilisant une SDR soigneusement programmée. Notre expérience montre que le décalage de fréquence pourrait être forgé à moindre coût, en modifiant la température d'un transmetteur plus classique.

5.4 Conclusion

Dans cette section nous proposons une approche de *fingerprinting* sur la couche physique Bluetooth Classic, qui permet de distinguer avec précisions onze modèles d'appareils différents.

Elle repose sur un nombre réduit de caractéristiques, qui peuvent être exprimées dans des grandeurs physiques bien établies.

Cette étude montre également que le *fingerprinting* sur la couche physique peut être réalisé avec du matériel abordable, et fonctionner en temps réel. En effet, nous l'avons basé sur des données collectées à l'aide de l'outil *Phyltre* présenté dans le chapitre 4.

La principale limite de notre approche est qu'elle ne permet que de distinguer le modèle d'un appareil, et confondra le plus souvent deux appareils du même modèle. Cependant, elle est simple à mettre en oeuvre, et fournit rapidement des informations synthétiques et intelligibles. On peut ainsi envisager de l'intégrer rapidement à un système de détection

d'anomalies.

Chapitre 6

Attaques de contrôleurs Bluetooth

6.1 Introduction

Dans les chapitres précédents, nous avons présenté une approche de détection d'intrusion qui utilisait des SDR pour capturer des transmissions sans fil, et les étudier directement au niveau de la couche physique. L'un des principaux avantage de cette approche est qu'elle nous permettait de collecter des informations pertinentes, sans avoir à se reposer sur des composants fermés. Par exemple, nous pouvions envisager la conception d'un IDS pour le Bluetooth, sans avoir à se reposer sur un contrôleur Bluetooth – potentiellement vulnérable – pour collecter des paquets. Cette propriété nous avait semblé particulièrement importante, pour éviter que notre IDS ne puisse être compromis par les mêmes attaques qu'il était justement censé détecter.

Ce chapitre décrit une étude dans laquelle nous avons mis à l'épreuve cette problématique, en recherchant des techniques pour compromettre un contrôleur Bluetooth. Nous nous sommes en priorité penché sur les attaques permettant l'exécution de code arbitraire à distance (RCE : *Remote Code Execution*), dans un scénario d'attaque crédible inspiré de BlueBorne [SV17].

Nous avons étudié quatre *firmwares* de constructeurs différents. Ils comportaient tous des vulnérabilités, et trois d'entre eux contenaient des RCE dont nous avons vérifié l'exploitabilité.

Le contenu de ce chapitre se déroule comme suit. La première section décrit les principaux travaux reliés aux notres sur la sécurité Bluetooth. La seconde section détaille notre modèle d'attaque sur les contrôleurs Bluetooth, et notre méthodologie pour en évaluer le niveau de sécurité. La troisième section décrit trois attaques sur des contrôleurs

Nous fournirons ensuite quelques élements d'information sur la couche LMP et les raisons pour lesquelles nous l'avons revue en priorité. Suivront une description générale de notre méthodologie, et son illustration par trois cas pratiques. Enfin pour conclure ce chapitre, nous discutons des implications de ces vulnérabilités, et proposons des contremesures.

6.2 Travaux antérieurs

La sécurité des protocoles Bluetooth est un sujet activement recherché depuis quelques années.

En 2017, Seri et al. de l'entreprise Armis ont publié le rapport *BlueBorne* [SV17] dans lequels ils décrivent des attaques contre des implémentations des couches hôtes du Bluetooth. En étudiant de façon systématique les différentes implémentations des mêmes fonctionnalités, ils découvrent des vulnérabilités similaires dans plusieurs implémentations, en particulier des fuites d'information dans les couche SDP de Linux et Android, ainsi que d'autres bogues permettant l'exécution de code arbitraire. La publication de BlueBorne a été suivie d'une vague de correction de nombreuses autres vulnérabilités dans des piles Bluetooth. L'approche présentée dans ce chapitre est très inspirée de BlueBorne. Nous sommes partis du même modèle d'attaque et l'avons appliqué aux processeurs *baseband* des contrôleurs Bluetooth.

En 2017 également, plusieurs recherches ont mené à la correction de vulnérabilités dans des *firmwares* de modems Wi-Fi Broadcom [Ben17, Art17, Ang19]. Beniamini avait notamment montré une attaque qui visait d'abord le chipset Wi-Fi d'un iPhone 7, puis exploitait une seconde vulnérabilité pour prendre le contrôle du kernel de l'iPhone 7 depuis la puce Wi-Fi [Ben17]. Artenstein avait pour sa part utilisé le chipset Wi-Fi compromis pour rediriger le navigateur mobile vers un site web malicieux [Art17].

Présenté en 2019, le projet InternalBlue [MCSH19] est un framework développé par l'équipe Seemo-lab pour modifier et étudier les *firmwares* des contrôleurs Bluetooth Broadcom/Cypress. Il permet d'y ajouter des instructions et d'étendre leurs fonctionnalités, et notamment d'injecter des paquets LMP et observer les échanges sur cette couche normalement inaccessibles à l'hôte. Frankenstein [RCGH20] également développé par Seemo-lab permet d'émuler des *firmwares* Broadcom notamment pour faciliter le *fuzzing*. Ces projets on permis la découverte de plusieurs vulnérabilités dans des contrôleurs Broadcom. En particulier, la CVE-2019-11516 est une RCE pouvant être exploitée lorsque la cible est en train d'effectuer un scan Bluetooth. La CVE-2019-13916, visant la pile Bluetooth Low-Energy, pouvait aussi permettre l'exécution de code arbitraire, mais aucun code d'exploitation n'a pu être développé.

En 2019, Seri et al. [SZV19] ont poursuivi leur exploration des protocoles Bluetooth par l'analyse des *firmware* des modems Bluetooth Texas Instrument. Il ont notamment présenté une RCE dans le décodage des paquets du canal d'*advertisement* Bluetooth Low-Energy (CVE-2018-16986).

En 2020, Kovah [Kov20] a présenté les résultats de son analyse des *firmwares* de modems Bluetooth Low-Energy Texas Instrument et Silicon Labs. Elle a développé des attaques pouvant conduire à l'exécution de code arbitraire dans chacun d'entre eux. Kovah explique qu'elle avait commencé par travailler sur le Bluetooth Classic, mais s'était ensuite rabattue sur le BLE faute d'outils pour transmettre des paquets Bluetooth Classic.

6.3 Vulnérabilités des contrôleurs Bluetooth Classic

Durant cette étude, nous avons poursuivi une approche similaire à celle présentée dans BlueBorne. Nous sommes parti d'un modèle d'attaque très proche, et nous avons recherché des vulnérabilités qui répondaient à ce modèle dans plusieurs implémentations différentes de couches LMP.

La première partie de cette section décrit plus en détail ce modèle d'attaque, ainsi que la couche LMP que nous avons choisi d'auditer en priorité. La seconde partie décrit la chronologie de l'analyse d'un contrôleur Bluetooth, en commencant par la rétro-ingénierie de son *firmware*, pour aboutir au développement d'une attaque.

6.3.1 Modèle d'attaque

Le Bluetooth n'est la porte d'entrée idéale pour une intrusion. Il n'a qu'une courte portée, et n'est pas forcément allumé. Les attaques sans fil nécessitent une proximité physique avec la cible. En analyse de risque, ce vecteur d'accès est donc considéré comme équivalent à un réseau local [1].

Pour identifier des attaques Bluetooth réalistes, nous avons recherché en priorité des vulnérabilités critiques permettant une compromission totale d'un contrôleur Bluetooth.

Pré-requis. Notre scénario d'attaque est identique à celui utilisé dans *BlueBorne*. Les pré-requis pour une attaque sont les suivants :

— le Bluetooth de la cible est allumé et *connectable* ;

— l'attaquant connait l'addresse Bluetooth de la cible (`BD_ADDR`).

En fait, un contrôleur Bluetooth actif est quasiment toujours dans l'état *connectable*. En pratique, ces attaques sont donc possibles dès lors que le Bluetooth de la cible est allumé. Plusieurs techniques permettent de récupérer l'adresse Bluetooth d'un appareil. Par exemple, un scan Bluetooth suffit à la révéler. On peut également *sniffer* un piconet pour récupérer la partie significative de l'adresse Bluetooth du maître. Enfin, l'adresse Bluetooth d'un téléphone est parfois directement reliée à l'adresse MAC de sa puce Wi-Fi.

Lorsque le Bluetooth de la cible est allumé, et que l'attaquant connait sa `BD_ADDR` , il peut ouvrir une connexion Bluetooth et commencer à échanger des paquets. Il peut ainsi exercer la grande surface d'attaque offerte par la pile Bluetooth, sans avoir besoin de s'authentifier.

Les attaques BlueBorne visaient l'hôte Bluetooth, c'est à dire les couches protocolaires qui sont prises en charge par le système d'exploitation de la machine (Linux, Windows, MacOS, Android). Dans notre étude, nous n'avons pas ciblé l'hôte, mais le contrôleur Bluetooth.

Les contrôleurs Bluetooth (souvent appelées puces Bluetooth) exécutent les couches les plus basses du protocole, et communiquent avec l'hôte via l'interface *Host Controler Interface* (HCI). L'implémentation des couches contrôleur est en grande partie implémentée

1. `https://insights.sei.cmu.edu/cert/2015/09/cvss-and-the-internet-of-things.html`

par du logiciel qui est exécuté par un processeur *baseband* situé dans la puce Bluetooth. Ce sont ces *firmwares* que nous avons analysés dans notre étude, et plus particulièrement leur implémentation de la couche LMP.

Exécution de code arbitraire. Dans cette étude, nous avons uniquement recherché des attaques permettant l'exécution de code arbitraire à distance (RCE), en privilégiant les vulnérabilités de type corruption mémoires. Nous aurions pu rechercher des vulnérabilités logiques, par exemple pour tenter de contourner les mécanismes de sécurité et usurper l'identité d'un autre appareil Bluetooth ou détourner ses communications. Mais des vulnérabilités inhérentes à la spécification Bluetooth offraient déjà ce type de possiblités [ATR19].

Les contrôleurs Bluetooth utilisent des CPU embarqués, et n'implémentent généralement aucune séparation des privilèges. Dans ce contexte, les attaques RCE permettent donc une totale prise de contrôle à distance du CPU et leur sévérité est maximale.

Maintien d'un accès au contrôleur. L'exécution d'une attaque Bluetooth nécessite un investissement important en temps et en matériel spécialisé. Nous considérons donc qu'un attaquant réel se contenterait pas de compromettre le contrôleur Bluetooth. Il chercherait plutôt à se déplacer du contrôleur Bluetooth vers le système d'exploitation de l'hôte, par exemple pour y implanter un cheval de troie permanent. Pour cela, il pourrait par exemple utiliser des failles dans l'interface entre le contrôleur et l'hôte.

Pour attaquer un hôte Bluetooth depuis un contrôleur compromis, l'intrus devrait avoir une entière maîtrise du contrôleur Bluetooth, et être en mesure d'y exécuter des charges actives complexe et potentiellement volumineuse. Cela n'est pas forcément évident, dans l'environnement d'une puce Bluetooth, qui dispose de peu de mémoire.

En pratique, c'est possible si l'attaquant peut placer une porte dérobée dans le contrôleur Bluetooth, pilotable à distance, capable de lire et d'écrire en mémoire, et d'appeler des fonctions arbitraires. En 2017, Beniamini avait développé une *backdoor* de ce type pour contrôler à distance un chipset Wi-Fi Broadcom compromis. Il a ensuite pu l'utiliser pour exploiter des vulnérabilités dans l'interface entre le chipset Wi-Fi et le kernel XNU d'un iPhone 7 pour compromettre le système d'exploitation du téléphone [Ben17].

Dans cette étude, nous n'avons pas cherché à développer des *backdoor* élaborées. Nous avons toutefois démontré qu'il était possible de le faire, en utilisant les failles RCE pour implanter de simples calculettes dans les contrôleurs. Ces calculettes implémentaient l'addition et la multiplication. Elles pouvaient recevoir un calcul de l'attaquant, et lui renvoyer le résultat. Cela démontre qu'on pouvait implanter un code arbitraire et le solliciter à distance. Ces calculettes auraient facilement pû être remplacées par une porte dérobée permettant de lire et d'écrire la mémoire du contrôleur, et de lui faire exécuter des charges actives supplémentaires.

6.3.2 Choix de la couche LMP

Nos travaux ont porté exclusivement sur la partie *baseband* du protocole Bluetooth Classic (BR/EDR), et en particulier sa couche *Link Manager Protocol* (LMP).

Nous aurions pû nous pencher sur les versions plus récentes du protocole Bluetooth, par exemple la couche LLCP du Bluetooth Low-Energy (BLE). En effet de plus en plus d'objets connectés ne supportent plus que le BLE. Mais selon nous, les objets connectés ne sont pas les cibles prioritaires : les données et processus les plus sensibles sont généralement situés dans les smartphones et les ordinateurs. Et comme les contrôleurs Bluetooth intégrés dans les smartphones et les PC supportent tous le Bluetooth Classic [2], les recherches sur sa sécurité restent d'actualité.

En Bluetooth Classic, le protocole LMP est, comme son nom l'indique, chargé de la gestion du lien. Il permet aux appareils d'échanger leurs fonctionnalités, de négocier les paramètres de la couche physique et de procéder à l'appairage. La couche LMP est entièrement implémentée en logiciel par le contrôleur Bluetooth, et constitue une grande partie de son *firmware*. Par ailleurs, les paquets LMP ne sont en principe pas visibles par l'hôte, ce qui en fait une surface d'attaque difficile à observer par un HIDS.

Nous nous sommes d'abord intéressés à la couche LMP car elle est selon nous la plus susceptible de contenir des vulnérabilités – ou des portes dérobées. Dans la suite de cette sous-section, nous caractérisons rapidement la surface d'attaque qu'offre la couche LMP.

Exposition. La couche LMP est responsable de la gestion du lien et de l'authentification. Il est donc directement exposé, sans authentification, dès lors que le contrôleur BT accepte les connexions.

Complexité. Les *firmwares* de *baseband* Bluetooth sont écrits dans des langages de bas niveau comme le C, dans lesquels des erreurs dans l'utilisation de la mémoire peuvent être exploitées. Il y a environ 90 types des messages LMP différents. Certaines des procédures LMP ont à manipuler des tampons mémoire et doivent vérifier scrupuleusement la valeurs des paramètres. Le code chargé de la couche LMP est donc sensible, et il était important de s'attarder sur la qualité des implémentations.

Surface peu explorée. Quand nous avons démarré nos recherches, il existait peu d'outils permettant d'interagir avec la couche LMP d'un contrôleur Bluetooth Classic. InternalBlue était le seul projet qui le permettait, grâce à une fonctionnalité pour injecter un paquet LMP arbitraire dans une connexion existante [MCSH19]. Cependant, cette fonctionnalité ne permettait pas de développer nos attaques. En effet, nous avions besoin de contrôler l'ensemble des messages LMP échangés avec la cible, afin de pouvoir manipuler précisément son état interne. Avec InternalBlue, les messages LMP injectés étaient simplement ajoutés au trafic normal d'un contrôleur Bluetooth Broadcom. Pour pouvoir communiquer sur la

2. Le Bluetooth Classic fournit des débits plus élevés que le Bluetooth Low-Energy, et reste très utilisé par exemple pour transfert de flux audio.

couche LMP, nous avons utilisé un contrôleur Bluetooth Qualcomm patché, en utilisant l'interface d'InternalBlue pour interagir avec la pile HCI d'Android.

6.4 Méthodologie

Pour rechercher des vulnérabilités et développer des attaques contre des contrôleurs Bluetooth, nous avons suivi une même méthodologie, que cette section résume en trois étapes.

6.4.1 Première phase de rétro-ingénierie

Pour pouvoir analyser le fonctionnement des *firmwares* Bluetooth, nous avons besoin de mettre en place un environnement de rétro-ingénierie. À l'issue de cette phase, nous devons pouvoir lire les instructions qui composent le *firmware* que l'on souhaite analyser.

Il faut d'abord récupérer un fichier contenant le *firmware* que l'on souhaite analyser. Ensuite, il faut déterminer quelle est l'architecture du CPU du contrôleur Bluetooth, et identifier les addresse de base des différents segments mémoire. Enfin, on peut charger le *firmware* dans l'outil de désassemblage de notre choix (par exemple IDA Pro ou Ghidra), et commencer à observer le code.

Pour grandement accélerer la suite de l'analyse, il est souhaitable de disposer d'un décompilateur capable de produire un code pseudo-C. En raisonnant au niveau du code C, on peut rapidement reconstruire les structures définies dans le programme, et les suivre à travers les appels de fonction. Par ailleurs, cela évite d'avoir à lire des instructions assembleur dans des architectures exotiques dont on ignorait parfois l'existence quelques jours auparavant.

Ghidra est de ce point de vue un excellent outil puisque son décompilateur fonctionne sur toutes les architectures qu'il peut désassembler. En outre, il est assez simple d'étendre Ghidra pour améliorer le résultat de la décompilation.

6.4.2 Identification du code pertinent

Une fois l'environnement de décompilation en place, nous avons besoin d'identifier le code à analyser.

Il n'est pas nécessaire de comprendre l'intégralité d'un programme pour y découvrir des vulnérabilités. Au contraire, on peut gagner beaucoup de temps en se concentrant sur un composant unique du code.

D'abord, cela permet de se concentrer principalement sur la surface d'attaque que l'on a sélectionné. De plus, lorsque l'on n'analyse qu'un seul composant, on peut se permettre de reconstruire en détail les structures qu'il utilise et les fonctions les plus fréquemment appelées. Ainsi lorsqu'on lit une nouvelle fonction, on comprend rapidement la nature des données qu'elle manipule, et les appels de fonction qu'elle réalise, ce qui permet de rapidement comprendre son rôle.

Lorsque nous avons analysé des *firmwares* Bluetooth, nous n'avons généralement pas eu besoin de comprendre le fonctionnement de leur noyau de système d'exploitation. Nous avons en priorité observé deux composants :

— la couche *Host Controler Interface* (HCI) qui réalise l'interface entre le controller Bluetooth et son hôte ;
— la couche *Link Manager Protocol* (LMP) qui réalise l'interface avec les appareils Bluetooth distants.

Les couches HCI et LMP sont définies dans la spécification Bluetooth. Le format de leurs messages est donc connu. Elles constituent donc d'excellents points d'entrée pour comprendre le fonctionnement d'un *firmware*.

L'identification du code de la couche HCI permet d'obtenir un grand nombre d'informations. Par exemple, en lisant le code qui gère la commande `HCI_CREATE_CONNECTION`, on peut comprendre comment l'état d'une connexion courante est stockée. En effet, dans cette fonction, le contrôleur doit commencer par vérifier s'il existe déjà une connexion vers l'adresse ciblée. On peut donc observer la façon dont la `BD_ADDR` donnée dans la commande HCI est utilisée pour retrouver une connexion courante, ce qui permet d'avoir une première idée de la taille de cette structure, sa localisation en mémoire, et le champ qui contient la `BD_ADDR` de l'appareil distant. En lisant l'implémentation d'autres commandes HCI, on peut ensuite identifier davantage de champs de cette structure et d'un grand nombre d'autres variables globales. Enfin, comprendre l'implémentation de la couche HCI permet deux choses supplémentaires :

— découvrir des commandes non documentées : la spécification HCI permet aux constructeurs d'ajouter des commandes propriétaires non documentées (*vendor-specific*), Ces commandes peuvent s'avérer utiles pour déboguer le contrôleur. Par exemple on trouve couramment des commandes pour lire et écrire la mémoire ;
— ajouter des fonctionnalités : si on peut modifier le *firmware* d'un contrôleur, on peut lui ajouter des nouvelles commandes HCI pour réaliser les opérations de notre choix.

La gestion des couches HCI et LMP constitue une partie significative du travail d'un contrôleur Bluetooth. Il est donc assez simple de les localiser et de remonter à la fonction chargée de distribuer les messages aux différents gestionnaires.

Pour localiser des fonctions de la couche LMP ou HCI, on peut recourir à différentes méthodes :

— *switch* volumineux : les *opcodes* HCI ou LMP peuvent être distribués au moyen de *switchs*. Comme il y a de nombreux *opcodes* différents, les fonctions contenant ces *switchs* ont une taille imposante, et on peut donc rapidement les identifier en observant les fonctions les plus volumineuses du *firmware* ;
— pointeurs de fonctions : les *opcodes* peuvent être distribués au moyen d'un tableau qui associe un *opcode* à un pointeur de fonction. Dans ce cas, on retrouvera de grandes tables de pointeurs de fonctions pour tous les messages d'une couche donnée ;
— constantes : les couches HCI et LMP manipulent de constantes définies dans leur spécification. Par exemple, une commande `HCI_CREATE_CONNECTION` commence par l'opcode `0x405`. De même, on peut rechercher la valeur 248 (la taille maximum d'un

nom d'appareil) pour retrouver les fonctions HCI et LMP qui gèrent les requêtes de nom d'appareil ;
— références croisées : les couches HCI et LMP manipulent fréquemment les structures qui représentent l'état des connexions Bluetooth courantes. En observant les références vers ces structures, on peut identifier davantage de code pertinent.

Finalement, on peut commencer à se constituer une vue d'ensemble de la couche LMP. Cette vue d'ensemble se compose de deux choses :
— les gestionnaires LMP : la liste des fonctions qui gèrent des messages LMP entrants, qui constituent le cœur de notre surface d'attaque ;
— les fonctions utiles : les fonctions qui servent allouer et libérer de la mémoire, ainsi que les fonctions essentielles de la libc ; les fonctions qui servent à communiquer avec les autres couches : celles qui transmettent des messages LMP vers l'appareil distant, ou des trames HCI à destination de l'hôte.

6.4.3 Recherche de vulnérabilités dans la couche LMP

La phase de recherche de vulnérabilités a généralement été plus courte que la première phase de rétro-ingénierie. En effet, à ce stade nous disposons d'une vue d'ensemble du fonctionnement de la couche LMP, dans un code pseudo-C dont nous avons identifié les principales fonctions et structures. Nous pouvons donc inspecter rapidement l'implémentation du protocole LMP, sans avoir à comprendre davantage le fonctionnement du matériel. Nous l'avons dit précedemment, la couche LMP comporte près de 90 messages différents. En observant la spécification LMP, nous avons remarqué que la plupart de ces messages possèdent des structures fixes, et sont peu susceptibles d'erreurs de décodage. Comme nous recherchions en priorité des corruptions mémoire pouvant mener à l'exécution de code arbitraire, nous avons d'abord passé en revue les gestionnaires de messages LMP impliquant des tampons mémoire, des messages de taille variable ou des indices. Cette approche nous a suffit a découvrir rapidement des défauts, et notamment de constater qu'aucun des *firmwares* n'implémentait correctement l'échange de nom LMP.

6.4.4 Développement de l'attaque

Après avoir découvert un bogue menant à une corruption mémoire, nous devons réaliser une seconde phase d'analyse statique pour identifier les possibilités offertes par cette corruption et déterminer si elle permet d'obtenir l'exécution de code arbitraire. En pratique, nous lirons tout le code susceptible d'utiliser les données corrompues pour comprendre comment le manipuler. Par exemple, lorsqu'il s'agit d'une écriture au-delà des bornes d'un tampon alloué dans le tas, il faut comprendre le fonctionnement de l'allocateur mémoire pour savoir comment le bogue peut influencer son comportement. De même, lorsqu'il s'agit d'un tampon statique, il faut connaître les variables susceptibles d'être écrasées et comment elles sont utilisées par le programme, pour pouvoir cibler précisément une variable intéressante à corrompre. Par exemple, en partant d'un bogue permettant d'écrire à une adresse relative à un tampon mémoire, on peut chercher une primitive pour réaliser une

écriture à une adresse arbitraire. Enfin, pour exécuter du code arbitraire dans une cible, il nous faut une vue d'ensemble du *firmware*. Nous devons donc au moins connaître les zones mémoires accessibles en lecture et en écriture, ainsi que les différents mécanismes de transfert du flôt de contrôle dont nous devrons tenir compte pour rediriger le code vers notre charge active (ou *shellcode*).

Après avoir choisi une primitive permettant l'exécution de code arbitraire, nous pouvons passer au développement du code d'exploitation.

Au cours de cette phase, nous développons deux programmes :

— le code d'exploitation : ce programme réalise des échanges LMP spécialement conçus pour installer une charge utile dans la cible, en exploitant une vulnérabilité ;

— la charge utile : c'est le cheval de troie que l'on va installer dans la mémoire de notre cible.

Les codes d'exploitation sont écrits en Python et utilisent une librairie LMP simplifiée, que l'on peut facilement étendre pour réaliser des transactions arbitraires. La charge utile est de préférence écrite en langage C pour gagner du temps, ou en assembleur lorsque ce n'est pas possible [3].

Dans nos attaques, la charge utile était une petite calculette avec laquelle on pouvait communiquer via LMP.

6.5 Cas pratiques

Cette section présente trois cas pratiques qui illustrent les différentes étapes de notre recherche de vulnérabilités.

La première partie présente la première phase de rétro-ingénierie d'un contrôleur Real-Tek. La seconde partie détaille la phase de recherche de vulnérabilités sur la ROM d'un contrôleur Broadcom. Enfin, la troisième section décrit le développement d'une attaque sur un contrôleur Qualcomm.

6.5.1 Première phase de rétro-ingénierie

L'objectif de la première phase de rétro-ingénierie est d'identifier l'architecture de l'appareil testé, de récupérer son *firmware*, et de le charger dans un IDE de décompilation.

Cette étape est plus difficile lorsque l'on ne dispose d'aucune documentation sur le composant. Dans cette section, on prendra l'exemple d'un chipset Bluetooth Realtek, pour montrer comment nous avons pu identifier son architecture, extraire sa ROM, et la charger dans Ghidra malgré l'absence de documentation.

Première recherche d'information. Le composant est un adaptateur Bluetooth Realtek RTL8821c d'un ordinateur Lenovo. Nos recherches ne nous ont pas permis de découvrir

3. Pour l'une des cibles, nous n'avions aucun compilateur supportant notre CPU, seulement un assembleur qui ne supportait pas toutes les instructions nécessaires à l'écriture de la charge utile.

de documentation pertinente ni sur ce chipset, ni sur des variantes. Tout au plus, nous apprenons que les SoC de points d'accès Wi-Fi Realtek utilisent l'architecture MIPS.

La source d'information la plus parlante au sujet du RTL8821c est son pilote Linux. Nous voyons qu'au démarrage, ce driver utilise la commande HCI non documentée `ogf=3f/ocf=20`[4] pour charger un patch dans la RAM du *baseband*. Nous disposons donc d'un patch, mais pas de la totalité du *firmware* qui est situé dans une mémoire ROM sur le RTL8821c.

Systèmes de RAM-patch. L'ajout de patch en RAM est une méthode couramment utilisée, car elle permet d'appliquer facilement des correctifs sur un *firmware* basé dans une ROM. Les systèmes de RAM-patch que nous avons rencontrés fonctionnaient de cette manière : chaque fonction importante peut être détournée, en définissant un pointeur de fonction *crochet* situé dans une mémoire réinscriptible. Lorsque la fonction est exécutée, elle commence par vérifier si le pointeur de crochet est défini, et si oui elle appele le crochet, puis exécute ou non le reste de la fonction selon la valeur de retour du crochet.

Désassemblage du patch. Sous Linux, les fichiers de patch sont situés dans le répertoire */lib/firmware/rtl_bt*. Il y a un fichier `rtl*_fw.bin` et un `rtl*_cfg.bin` par modèle de chipset, que nous supposons contenir repectivement le code exécutable et des éléments de configuration. Le fichier `fw.bin` est le plus important, le fichier de configuration `rtl8821c_cfg.bin` contient seulement dix octets. Le pilote selectionne le bon jeu de fichiers en fonction du numéro de version du *firmware* qu'il récupère via la commande `HCI_OP_READ_LOCAL_VERSION`. Chaque fichier `fw.bin` contient en réalité plusieurs versions des patches, qui correspondent aux différentes versions de ROM du composant. Par exemple, il existe deux versions différentes pour le RTL8821c. Le pilote récupère le version de la ROM au moyen de la commande HCI non documentée `ogf=3f/ocf=6d`.

Après avoir extrait les patchs correspondant à notre *firmware*, nous utilisons `cpu_rec` [Air] pour tenter de reconnaitre son architecture. L'outil échoue sur notre patch, mais pour d'autres versions il indique MIPS16. Nous chargons le patch dans Ghidra et constatons qu'il s'agit en effet de code MIPS16 *little endian* valide.

Il faut ensuite localiser l'adresse de base du patch pour pouvoir suivre les références dans le code et être en mesure de le comprendre. La fonction située au tout début du patch semble avoir pour rôle d'installer les correctifs. Elle assigne une quarantaine de valeurs dans une plage mémoire qui commence à `0x80120000`. Beaucoup des valeurs assignées sont des pointeurs qui commencent autour de `0x80100000`, et la plupart d'entre eux sont impairs, ce qui en MIPS indique un pointeur vers une fonction qui utilise l'extension MIPS16. Cela correspond au fonctionnement d'un système de RAM-patch classique et les pointeurs assignés doivent pointer vers les fonctions situés dans notre patch. On peut facilement en déduire l'adresse de base du patch, en faisant correspondre les pointeurs de fonction

4. Une commande HCI est définie par son *Opcode Group Field* (OGF), et son *Opcode Command Field* (OCF).

au décalage des fonctions définies dans le patch MIPS16. Pour le RTL8821c, le patch est chargé à l'adresse `0x8010A000`.

Extraction du contenu de la ROM. À première vue, le format de notre patch ne contient aucune vérification d'intégrité. Le code MIPS démarre au tout début du fichier et il ne semble contenir aucun épilogue. Nous remarquons que la première opération réalisée par le patch est d'installer le nouveau numéro de version, qui est situé tout à la fin du fichier de patch. Lorsqu'on modifie ce numéro de version et que l'on installe le patch, on peut voir que la commande `HCI_OP_READ_LOCAL_VERSION` nous renvoie bien le numéro renseigné. Nous en déduisons qu'il n'y pas de vérification d'intégrité sur le patch et que nous pouvons donc forger un patch pour faire exécuter du code arbitraire dans notre chipset.

Le premier obstacle que nous rencontrons est que nous ne pouvons pas charger plusieurs patchs dans notre adaptateur Bluetooth sans redémarrer le RTL8821c, ce qui dans notre configuration implique de redémarrer l'ordinateur. Cela rend les expérimentations très laborieuses. Par chance, en testant différentes commandes HCI proches de celle qui sert à récupérer la version de la ROM nous découvrons que la commande `ogf=3f/ocf=66` retire le patch et redémarre le chipset. On peut ainsi charger un nouveau patch sans avoir à redémarrer l'ordinateur.

Pour vérifier que nous pouvons exécuter un patch arbitraire, nous écrivons un premier patch qui modifie le numéro de version. Nous constatons que le numéro de version change comme prévu. À ce stade, nous en savons déjà suffisamment pour extraire le contenu de la ROM.

Nous réalisons une deuxième version du patch qui lit quatre octets au début de la ROM (`0x80000000`) et les écrit dans le numéro de version. Nous pouvons ensuite lire le numéro de version de l'adaptateur à l'aide de la commande `HCI_OP_READ_LOCAL_VERSION`, et récupérer ces quatre octets. Il suffit ensuite de retirer le patch et de répéter l'opération avec l'addresse suivante, etc. Finalement, cette méthode nous permet de lire la ROM à un débit de 6,2 octets par seconde. Ce n'est pas idéal, mais nous obtenons les `0x70000` octets du contenu de la ROM en moins de 24 heures.

Un meilleur accès à la mémoire. Après avoir obtenu le contenu de la ROM, nous avons pû identifier l'implémentation du protocole LMP, mais aussi des commandes HCI non documentées. Certaines de ces commmandes offrent un accès en lecture et en écriture à la mémoire du RTL8821c.

La commande `ogf=3f/ocf=61` permet de lire un, deux, ou quatre octets en mémoire. Le premier octet des paramètres (`0x21` dans l'exemple), encode la taille accédée dans son semioctet haut (0=1-octet, 1=2-octets, 2=4-octets), le semioctet bas doit valoir 1 [5]. les quatre octets suivants représentent l'adresse en petit-boutiste. Par exemple, pour lire quatre octets à l'adresse `0x12345678` :

```
hcitool cmd 3f 61 21 78 56 34 12
```

5. Nous n'avons pas identifié précisément le rôle du semioctet bas.

La commande `ogf=3f/ocf=62` permet d'écrire un, deux ou quatre octets en mémoire. Le premier octet des paramètre utilise le même format que dans la commande de lecture. Les quatre octets suivants représentent l'adresse, suivie des octets à écrire. Par exemple, pour écrire ABCD à l'adresse `0x12345678` :

```
hcitool cmd 3f 62 21 78 56 34 12 41 42 43 44
```

Ces commandes facilitent nettement l'analyse de ce *firmware* : la technique que nous avons utilisé pour lire le contenu de la ROM nécessitait de redémarrer le contrôleur, et ne permettait donc pas de récupérer la mémoire du *firmware* en cours d'exécution.

Avec ces commandes, nous pouvons obtenir rapidement une image de la ROM mais aussi de la mémoire vive du contrôleur en fonctionnement. De plus, nous pouvons désormais écrire en mémoire, ce qui permet d'instrumenter le *firmware* pour faciliter son débogage ou étendre ses fonctionnalités.

6.5.2 Recherche de vulnérabilité

Après avoir identifié les fonctions des couches LMP et HCI dans le *firmware* et reconstitué les principales structures qu'ils utilisent, nous pouvons passer en revue les principales fonctions de la couche LMP.

Cette section décrit une intéressante vulnérabilité présente dans la ROM de contrôleurs Bluetooth conçus par Broadcom (depuis acquis par Cypress Semiconductors).

Comme pour tous les *firmwares*, nous avons commencé par observer l'implémentation des procédures LMP qui comportaient des mécanismes de fragmentation. Ces procédures doivent réassembler des paquets. Elles manipulent des tampons mémoire et des données maléables par un adversaire. Il semble donc naturel de commencer par les auditer.

Nous avons commencé par observer l'implémentation de l'échange de nom LMP, qui ne comportait pas de défaut exploitable. Nous avons poursuivi par l'étude de l'implémentation de l'échange de clé publique.

Échange de clé publique. La charge utile d'un paquet LMP ne peut excéder 16 octets. Certaines procédures requièrent pourtant des paramètres de tailles plus importantes. Le protocole LMP comprend donc deux opcodes pour échanger des PDU (*protocol data unit*) encapsulés. Le paquet `LMP_ENCAPSULATED_HEADER` indique le type et la taille du message encapsulé qui suivra. Les `LMP_ENCAPSULATED_PAYLOAD` transportent ensuite les fragments de PDU (16 octets).

Nous n'avons pas trouvé de défaut particulier dans le décodage des paquets `LMP_ENCAPSULATED_HEADER`[6]. En fait, les PDU encapsulés sont uniquement utilisés pour l'échange de clé publique au cours de l'appairage[7]. La taille de la clé publique peut être

6. Dans un des *firmwares* cependant, un bogue pouvait se produire si on omettait d'envoyer le *header* et que l'on envoyait directement le *payload*.

7. C'est dommage, car ils auraient aussi pû être utilisés pour l'échange de nom, au lieu de quoi la spécification a introduit deux mécanismes de fragmentation différents, et créé une plus grande surface d'attaque logicielle.

de 192-bit ou 256-bit selon les paramètres de l'appairage, nécessitant donc trois ou quatre paquets `LMP_ENCAPSULATED_PAYLOAD`. Le contenu des `LMP_ENCAPSULATED_HEADER` est donc connu à l'avance par son destinataire, et dans les implémentations que nous avons observé leur contenu était soit ignoré soit exhaustivement validé.

C'est le traitement des paquets `LMP_ENCAPSULATED_PAYLOAD` qui posait problème dans la ROM Broadcom.

Dans les *firmwares* de ces contrôleurs, les évènements concernant des procédures de sécurité sont envoyés dans une unique machine à état (`PFSM`). À la réception d'une commande HCI ou d'un message LMP, un évènement correspondant est envoyé dans `PFSM`. `PFSM` n'accepte que certains types d'évènements selon la valeur son état. Ainsi, dans notre scénario d'attaque[8], les paquets LMP de type `LMP_ENCAPSULATED_PAYLOAD` (évènement 6) ne sont acceptés que lorsque `PFSM` est dans l'état 5.

```
void pfsm_st5_traite_ev6_encap_payload(bt_conn_s *conn)
{
  // Copie un fragment de 16 octets
  memcpy(tampon + (uint)indice * 4,
         lmp_rx_pdu + 0xd,
         0x10);
  /* ... */
  indice = indice + 4;
  // Vérifie que tous les fragments ont été reçus.
  if (indice*4 >= taille_de_clef_attendue())
  {
    /* ... */
    // Change l'état de PFSM
    etat_pfsm = 4;
  }
}
```

Listing 6.1: Extrait simplifié du gestionnaire vulnérable

Le listing 6.1 présente une version simplifiée du gestionnaire de `PFSM` pour l'évènement 6. Le *tampon* utilisé pour accumuler les fragments de clé peut contenir jusqu'à 64 octets. Le nombre d'octets reçus est stocké (divisé par 4) dans *indice* qui est un entier non signé de 8 bits. *tampon* et *indice* sont deux variables statiques, leur adresse en mémoire est fixe.

Le fragment qui vient d'être reçu (`lmp_rx_pdu+0xd`) est d'abord copié à la suite du tampon. Après la copie, l'indice de réception est incrémenté. Enfin, si l'indice atteint la taille de la clé attendue, `PFSM` passe dans l'état 4 dans lequel les paquets `LMP_ENCAPSULATED_PAYLOAD` ne sont plus acceptés.

Cette fonction parait suspecte car l'indice n'est pas vérifié avant la copie mémoire, mais après. On pourrait donc tenter d'envoyer plus de fragments que prévu pour écrire au-delà des bornes du tampon. La seule chose qui l'empêche est le changement d'état de

8. L'attaquant initie la connexion et l'appairage.

PFSM. Notez que l'on ne peut pas simplement recommencer l'appairage, car l'indice est réinitialisée dans le premier état de PFSM.

Pour pouvoir écrire davantage dans *tampon*, il faut donc trouver un moyen de faire revenir PFSM de l'état 4 à l'état 5, sans passer par la fonction d'initialisation. Nous avons recensé les modifications d'état possibles de PFSM en recherchant toutes les fonctions qui modifiaient la variable etat_pfsm. Pour chacune d'entre elles, nous avons recherché dans quelles conditions elles pouvaient être déclenchées. Dans la plupart des cas, ces conditions dépendent de l'état courant de PFSM, et du rôle du *firmware* ciblé dans l'appairage [9].

Boucle infinie. Des transitions d'état inattendues peuvent survenir lorsque le contrôleur reçoit certains LMP_NOT_ACCEPTED indiquant le refus d'une LMP_IO_CAPABILITY_REQ.

Un paquet LMP_NOT_ACCEPTED est envoyé en réponse à un paquet LMP pour signifier son refus. Il contient l'opcode du paquet refusé ainsi qu'un code d'erreur. Le message LMP_IO_CAPABILITY_REQ est pour sa part envoyé par le maître pour initier un appairage.

Lorsque le *firmware* reçoit un message de refus d'un paquet LMP_IO_CAPABILITY_REQ avec le code d'erreur 42, PFSM passe dans l'état 2, quel que soit état courant.

Il peut ensuite passer de l'état 2 à l'état 3 s'il recoit un message de refus d'une LMP_IO_CAPABILITY_REQ pour la raison 35 [10].

Enfin, il est possible de forcer PFSM à passer de l'état 3 à l'état 5, en lui envoyant un paquet LMP_IO_CAPABILITY_REQ(tid=0) puis un paquet LMP_IO_CAPABILITY_REQ(tid=1).

Dans l'état 5, PFSM accepte de nouveau les paquets LMP_ENCAPSULATED_PAYLOAD. On peut donc envoyer un fragment supplémentaire, qui sera ajouté à la suite du tampon, avant que PFSM ne passe à nouveau dans l'état 4.

En répétant ces étapes, on peut déclencher une boucle infinie dans PFSM, et continuer d'ajouter des fragments à la suite du tampon, jusqu'à ce que l'indice dépasse sa capacité (lorsque 1024 octets ont été écrits dans le tampon).

Dépassement creux. La variable *indice* est stocké peu après le tampon vulnérable, dans la même structure. Écrire au-delà du tampon permet donc rapidement de contrôler la valeur de l'indice. On peut ainsi écraser une variable située dans les 1024 octets suivant le début du tampon (de 64 octets), sans avoir à écraser les données situées entre le tampon et la variable. Cela permet d'écrire un code d'exploitation fiable, qui ne corrompt que les variables nécessaires à l'exécution de la charge active. Ainsi, le contrôleur Bluetooth ciblé reste en parfait état de marche, et nous pouvons continuer à communiquer avec notre charge active via LMP.

Nous avons pu vérifier que ce dépassement de tampon permettait l'exécution de code arbitraire sur le contrôleur BCM4345C0 d'un Raspberry Pi 3B+, dont la ROM date d'Aout

9. La cible était esclave dans la procédure d'appairage.

10. Les erreurs 42 (*Different transaction collision*) et 35 (*LL procedure collision*) sont normalement prévues pour gérer les cas dans lesquels deux appareils tentent simultanément d'initier une même procédure ou deux procédures incompatible. Il semble que le *firmware* vulnérable vérifie mal son état et son rôle avant de traiter ces erreurs, ce qui permet de lui faire recommencer l'appairage à un état antérieur, sans passer par la fonction d'initialisation.

2014. En écrasant un pointeur de fonction utilisé par un gestionnaire LMP, nous avons pu rediriger le flot de contrôle vers le tampon statique, et finalement exécuter notre petite calculette LMP.

Ce genre de vulnérabilités illustre bien l'importance des revues manuelles de code. Étant donné les conditions très spécifiques qu'il faut réunir pour déclencher ce bogue, il semble impossible de le découvrir par *fuzzing*.

Nous avons prévenu Broadcom qui nous a indiqué n'être pas concernés, et Cypress Semiconductors qui ne nous a pas répondu. Nous supposons que cette vulnérabilité avait été corrigée dans les ROM plus récentes, entre 2015 et 2019.

6.5.3 Développement d'une attaque

Dans les sections précédentes, nous avons décrit les phases suivies pour comprendre une implémentation LMP et y rechercher des vulnérabilités.

La dernière phase de notre analyse a consisté à vérifier en pratique si ces failles étaient exploitables et permettaient réellement d'obtenir l'exécution de code arbitraire à distance.

Cette section traite d'une vulnérabilité que nous avions trouvée dans le *firmware* Bluetooth des SoC Qualcomm Snapdragon (CVE-2019-14095). Nous y présentons en détail cette vulnérabilité, ainsi que les techniques utilisées pour développer une attaque réaliste.

Vulnérabilité dans l'échange de nom LMP

Les deux premiers opcodes définis dans le protocole LMP concernent la procédure d'échange de nom. L'opcode 1 indique une requête de nom (`LMP_NAME_REQ`), et l'opcode 2 une réponse (`LMP_NAME_RES`).

Échange de nom. Les noms d'appareils Bluetooth Classic peuvent compter jusqu'à 248 octets, mais un paquet LMP ne peut transporter que 16 octets de données utiles [11]. La procédure d'échange de nom doit donc permettre d'envoyer un nom d'appareil fragmenté.

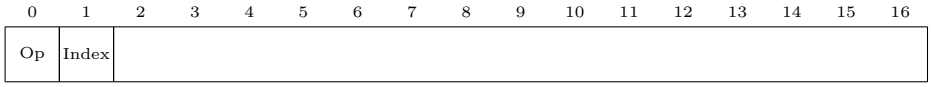


Figure 6.1: Format d'une requête de nom.

Les figure 6.1 et 6.2 montrent les formats des paquets utilisés pour l'échange de nom d'appareil. La requête (`LMP_NAME_REQ`) indique l'indice du fragment de nom demandé sur 1 octet. La réponse (`LMP_NAME_RES`) indique la position de la réponse sur un octet, la taille totale du nom sur un octet, puis le fragment de nom (14 octets maximum).

11. Les paquets LMP ont une taille maximale de 17 octets, dont un octet d'opcode, et 16 octets de données utiles

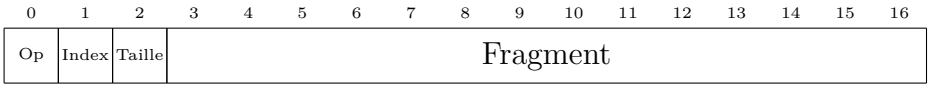


FIGURE 6.2: Format d'une réponse de nom.

Dans un langage bas niveau, on imagine facilement qu'une mauvaise implémentation de cette fonction pourrait conduire à des vulnérabilités exploitables. D'abord, le nom d'appareil ne devrait pas excéder 248 octets, mais sa taille est transmise dans un octet qui peut valoir jusqu'à 255. Une absence de vérification sur cette valeur pourrait conduire à un dépassement de tampon. Par ailleurs, le taille du nom est transmis dans chaque réponse, le *firmware* doit valider la cohérence de cette taille entre plusieurs réponses successives, et vérifier que la position indiquée dans la réponse n'excède pas la taille. L'absence de validation adéquate pourrait induire des calculs incorrects sur un indice ou une taille, ce qui pourrait également causer une corruption mémoire.

La procédure d'échange de nom semblait particulièrement susceptible de contenir des vulnérabilités. À chaque fois que nous avons démarré l'analyse d'un nouveau *firmware*, nous avons donc commencé par passer en revue son implémentation de l'échange de nom.

Dans les *firmwares* Qualcomm, cette approche a suffit pour identifier rapidement une vulnérabilité critique.

Implémentation vulnérable. Dans le *firmware* Qualcomm, la fonction qui gère les réponses de nom (`LMP_NAME_RES`) comportait plusieurs défauts.

D'abord, un tampon de 260 octets était alloué pour stocker le nom reçu. Les 9 premiers octets de ce tampon étaient utilisés pour stocker les informations sur la requête de nom courante, tandis que les 251 octets restants étaient disponibles pour stocker le nom reçu.

Pour stocker l'état d'une procédure de requête de nom, le *firmware* stockait essentiellement deux variables dans l'en-tête de 9 octets :

— `taille_courante` : taille reçue (entier non signé de 8 bits) ;
— `taille_totale` : taille totale (entier non signé de 8 bits).

Lorsque le *firmware* débute une requête de nom, `taille_courante`, `taille_totale` et le tampon de nom sont initialisés à zéro, puis un premier `LMP_NAME_REQ` est envoyé. Le gestionnaire de `LMP_NAME_RES` fonctionnait ainsi :

1. Vérifie que `name_offset` correspond à `taille_courante` ;
2. si `taille_courante` vaut zéro, stocke `name_length` dans `taille_totale` ;
3. calcule la taille du fragment courant (`taille_fragment`) ;
4. ajoute le fragment à la fin du tampon de nom ;
5. incremente `taille_courante` de `taille_fragment` ;
6. si `taille_courante` est inférieure à `taille_totale`, envoie un nouveau paquet `LMP_NAME_REQ (indice=taille_courante)`.

Il y avait deux problèmes dans cette implémentation. D'abord, `name_length` n'était pas validé. Même si la taille d'un nom d'équipement Bluetooth ne doit pas excéder 248 octets, il était possible de spécifier une taille de 255 octets, ce qui provoquait l'écriture de 4 octets au-delà des bornes du tampon de 251 octets.

Le second problème se produisait lorsque le *firmware* recevait des `LMP_NAME_RES` successifs avec des champs `name_length` inconsistents. Le champ `name_length` des paquets `LMP_NAME_RES` suivants était utilisé pour calculer `taille_fragment` comme suit :

```
taille_fragment = MIN(14, name_length - taille_courante)
```

Le *firmware* ne vérifiait pas que le champ `name_length` était identique dans chaque réponse. On pouvait exploiter ce défaut à deux fins : incrémenter `taille_courante` d'une valeur arbitraire, et forcer `taille_fragment` à toujours valoir 14.

Normalement, `taille_courante` devrait toujours être incrémenté de 14 octets. Mais il était possible de forcer `taille_courante` à être incrémenté par n'importe quelle valeur $\texttt{n} \in [1; 14]$ en envoyant une `LMP_NAME_RES` avec `name_length = taille_courante+n`. Ainsi, nous pouvions forcer la cible à avancer `taille_courante` jusqu'à 254 [12].

Comme `name_length` et `taille_courante` sont tous deux des nombres non signés, spécifier `name_length` inférieur à `taille_courante` causait un dépassement d'entier dans le calcul de `taille_fragment`. Après l'opération `MIN`, `taille_fragment` valait donc toujours 14.

Avec ces deux méthodes, il était possible de forcer la cible à envoyer une dernière `LMP_NAME_REQ` avec un offset de 254, et de répondre avec un fragment de 14 octets. Il en résultait qu'un total de 268 octets était écrit dans le tampon de nom de 251 octets, soit un dépassement de 17 octets.

Après avoir reçu le dernier fragment, `taille_courante` aurait dû valoir 268, mais comme sa valeur n'est stockée que sur un octet, sa valeur en mémoire était de 12 (268 modulo 256). Le *firmware* envoyait donc une nouvelle `LMP_NAME_REQ` avec `name_offset`=12.

Ce comportement nous a permis de confirmer la présence de la vulnérabilité sur des SoC dont nous n'avions pas le *firmware*.

Primitives d'exploitation

Après avoir bien compris la nature de la vulnérabilité, nous avons recherché comment un dépassement de 17 octets pouvait être exploité. Pour cela, il fallait d'abord comprendre le fonctionnement de l'allocateur mémoire du *firmware* vulnérable.

Allocateur de pool. Le tampon vulnérable est alloué depuis un *pool* d'élements d'une taille fixe de 260 octets que nous appelons `pool260`. Dans la version du *firmware* que nous avons analysé, `pool260` ne comprend que deux élements.

L'allocateur comprend deux structures illustrées par la figure 6.3 : un *descripteur* contenant un pointeur vers une liste d'éléments libres, et un *tampon de données* qui contient les deux éléments. Chaque élément est précédé de 8 octets de métadonnées :

12. Au lieu de 252 qui est le plus grand multiple de 14 inférieur à 255

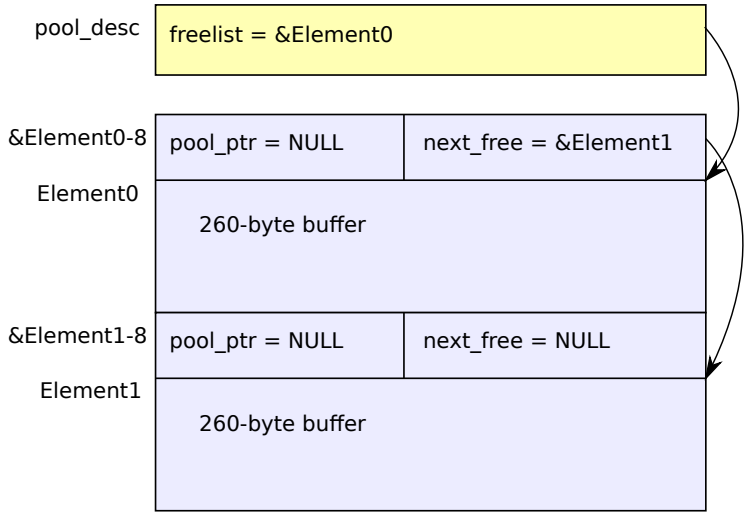


FIGURE 6.3: Etat initial de `pool260`

— `pool_ptr` : présent si l'élément est utilisé, pointe vers le descripteur du pool d'allocation ;
— `next_free` : présent si l'élément est libre, pointe vers le prochain élément libre.

La fonction d'allocation renvoie l'élément pointé par le pointeur `freelist` du descripteur, et le remplace par le `next_free` de l'élément renvoyé.

Exploitation de l'allocateur. En écrivant au-delà de `Element0`, nous pouvons écraser le pointeur `next_free` d'`Element1` avec un pointeur arbitraire (`fake_free`). Puis si nous allouons deux éléments supplémentaires depuis `pool260`, la fonction d'allocation renverra d'abord `Element1` puis `fake_free`.

En fait, `pool260` est très peu utilisé dans le *firmware*. Le seul appel que nous avons identifié était celui de la requête de nom. Il n'etait donc pas trivial d'atteindre un état dans lequel trois éléments seraient alloués depuis `pool260` en même temps (à la base, il ne comporte que deux éléments).

Pour forcer l'allocation de n'importe quel nombre d'éléments depuis `pool260`, nous avons tiré partie d'une fuite mémoire dans la fonction du *firmware* qui gère les requêtes `HCI_NAME_REQUEST` reçues de l'hôte. Lorsque le *firmware* recevait une deuxième commande `HCI_NAME_REQUEST` avant que la requête de nom précédente se termine, un nouveau tampon était alloué depuis `pool260` et le tampon existant n'était pas libéré.

Nous avons remarqué que Bluedroid [13] envoie toujours une commande `HCI_NAME_REQUEST` lorsqu'un appareil tente de s'appairer et que son nom n'est pas connu, même si une requête de nom est déjà en cours. Cela nous a permis de forcer la fuite d'un élément de `pool260` en suivant ces étapes :

1. se connecter à l'appareil et envoyer une `LMP_HOST_CONN_REQ`. Une première com-

13. Bluedroid est la pile Bluetooth en espace utilisateur d'Android

mande `HCI_REMOTE_NAME_REQ` est envoyée au *firmware* ;

2. répondre à la première `LMP_NAME_REQ` sans terminer la procédure,
3. initier un appairage via LMP. Une seconde `HCI_REMOTE_NAME_REQ` est envoyée au *firmware* ;
4. fermer la connexion. Le second tampon de nom est libéré, mais le premier est perdu.

Exécution de code arbitraire. En combinant les différentes techniques décrites ci-dessus, nous pouvions conduire le *firmware* à allouer un tampon de nom à une adresse arbitraire. Les fragments de nom que nous envoyions étaient ensuite stockés à cette adresse, et nous pouvions y écrire 268 octets arbitraires.

Dans les *firmwares* Qualcomm que nous avons analysés, tout le code était accessible en lecture-écriture-exécution. Nous pouvions donc simplement écraser le code d'un gestionnaire LMP avec notre shellcode.

Comme le tampon de 251 octets est initialisé à zéro avant d'être utilisé, il fallait choisir un gestionnaire LMP suffisamment grand pour éviter d'écraser une fonction adjacente. Nous ne pouvions pas non plus écraser un gestionnaire de paquets nécessaires à l'attaque. Nous avons choisi le gestionnaire de `LMP_ENCAPSULATED_HEADER` qui remplissait ces conditions.

Le tampon de nom commence 9 octets après le début de l'élément alloué, il fallait donc soustraire au moins 9 octets de l'adresse d'écriture pour construire `fake_free`. Pour éviter des accès mémoire non-alignés dans les métadonnées du faux élément, nous avons également dû aligner `fake_free` sur 4 octets.

Nous avons rencontré un dernier problème : notre shellcode était écrit dans le cache de données, ce qui faisait échouer l'attaque sur une des cibles. En observant la configuration de la MMU, nous avons découvert un mapping non caché de l'intégralité de la mémoire à `addresse_physique + 0x20000000`. Écrire dans ce mapping a permis d'éliminer les erreurs liées au cache.

Conclusion Finalement, nous avons pu complètement remplacer un gestionnaire LMP avec une calculette. Nous avons testé cette attaque sur deux Qualcomm Snapdragon S4, sur Android 7 et 8. Nous avons remonté cette faille à Qualcomm en 2019, et un patch a été distribué dans la mise à jour de sécurité Android de mars 2020. Suite à notre rapport, Qualcomm nous a versé une récompense de 8000$ via leur programme HackerOne [14].

6.6 Analyse

Au cours de cette étude, nous avons étudié les *firmwares* de contrôleurs Bluetooth de quatre constructeurs différents. Le choix des cibles s'est fait sans critère particulier, nous avons simplement analysé des appareils très cuorants qui étaient à notre disposition.

14. `https://hackerone.com/qualcomm`

À l'exception d'un appareil qui n'était plus vendu ni mis à jour, nous avons signalé toutes les vulnérabilités aux constructeurs dans la semaine suivant leur découverte.

Cette section condense nos conclusions sur la sécurité des contrôleurs Bluetooth. Nous y présentons les principales sources de vulnérabilités identifiées et des risques qu'elles font peser sur la sécurité de nos appareils. Enfin, nous proposerons des pistes pour mitiger ces risques et améliorer la sécurité des contrôleurs Bluetooth.

6.6.1 Sources de vulnérabilités

Au final, notre analyse a principalement porté sur le décodage des paquets LMP. Les défauts que nous avons identifiés de cette façon ne nécessitaient pas une complète compréhension de la pile LMP. Notre analyse a été superficielle, nous avons principalement observé la façon dont étaient manipulées les entrées utilisateur. Néanmoins, cette approche a permis de découvrir rapidement des défauts simples mais souvent critiques dans les quatre contrôleurs Bluetooth de notre échantillon.

Notez que même si ces défauts étaient identifiables par une revue manuelle du code [15], certains d'entre eux auraient été beaucoup plus difficiles à découvrir par des tests automatisés, comme le montre l'exemple de la vulnérabilité des ROM Broadcom, qui semble quasiment impossible à découvrir autrement que par une analyse statique de la couche LMP.

Malgré tout, certaines vulnérabilités pourtant rapidement identifiables ont survécu pendant trop longtemps dans beaucoup d'équipements Bluetooth. La suite de cette sous-section en décrit les plus représentatives.

Échange de nom. Nous avons décrit la procédure d'échange de nom dans la section 6.5.3. Cette procédure permet d'obtenir le nom de l'appareil distant, qui est envoyé par fragments de 14 octets. Les requêtes de nom (`LMP_NAME_REQ`) contiennent la position demandée dans le nom, et les réponses (`LMP_NAME_RES`) contiennent la position, la taille totale du nom, et le fragment de nom. C'est surtout le traitement des réponses qui nous a intéressé.

Un défaut dans le traitement de ces paquets est particulièrement susceptible de provoquer des corruptions mémoire. Dans chaque *firmware*, nous avons donc commencé par passer en revue la fonction qui traite les paquets `LMP_NAME_RES`.

Tous les gestionnaires de `LMP_NAME_RES` que nous avons observés comprenaient des défauts.

Seul l'un des trois *firmwares* vérifiait que la taille du nom était inférieur ou égale à 248 octets, mais un dépassement d'entier sur le calcul de la taille d'une copie mémoire pouvait causer un crash du contrôleur.

Dans deux *firmwares*, l'absence de vérification causait un dépassement de tampon permettant d'écraser des variables adjacentes. Sur le premier des *firmwares* cela pouvait provoquer l'exécution de code arbitraire ; sur le second nous n'avons pas pû identifier de primitive

15. Toutes les vulnérabilités que nous avons pu exploiter impliquaient la fonction *memcpy*.

d'exploitation utile et nous pensons que le dépassement ne permettait pas de RCE (sans pouvoir en être sûr).

Dans le dernier *firmware*, l'absence de vérification ne provoquait par chance aucun dépassement. C'est un contexte logiciel particulier qui empêchait le bogue de se manifester. Si ce défaut n'est pas corrigé, une future évolution de logiciel pourrait le rendre exploitable.

Échange de fonctionnalités. Une des RCE rencontrées aurait facilement pu être découverte par *fuzzing*. C'est cependant en analysant statiquement une couche LMP que nous l'avons identifié. Cette faille était présente dans le *firmware* d'un ancien contrôleur Bluetooth, qui n'est aujourd'hui plus maintenu.

Deux appareils Bluetooth peuvent s'échanger leurs *fonctionnalités*, pour pouvoir négocier des paramètres qu'ils supportent chacun. En effet, le standard Bluetooth a beaucoup évolué au fil des années, et les nouveaux appareils doivent rester compatibles avec les anciens.

Les fonctionnalités sont représentées sous la forme de masques de 64-bits. Les premiers contrôleurs Bluetooth ne supportaient qu'une *page* de 64-bits, que les appareils échangent via des paquets `LMP_FEATURES_REQ` et `LMP_FEATURES_RES`. À l'exception de l'opcode, ces deux types de paquets partagent le même contenu : la page de 64 bits de leur émetteur. Cela permet d'échanger rapidement les fonctionnalités en deux paquets : la requête et la réponse.

Les contrôleurs Bluetooth supportent désormais des pages supplémentaires de fonctionnalités *étendues* qui sont échangées via les paquets `LMP_FEATURES_REQ_EXT` et `LMP_FEATURES_RES_EXT`. Comme précédemment, la requête et la réponse contiennent chacunes une page de fonctionnalités. Mais en plus, ces messages indiquent l'indice de la page demandée, et l'indice maximal supportée par l'émetteur. Ces indices sont stockés sur 8 bits et peuvent valoir de 0 à 255.

Dans l'un des *firmwares* analysés, la fonction prenant en charge les paquets `LMP_FEATURES_RES_EXT` ne vérifiait pas que l'indice de la page était valide. Étonnamment, seul le gestionnaire de `LMP_FEATURES_RES_EXT` était vulnérable. La fonction qui traitait les `LMP_FEATURES_REQ_EXT` vérifiait correctement l'indice. En fait, les deux fonctions étaient très similaires, à l'exception de la vérification manquante. Le *firmware* pouvait gérer trois pages de fonctionnalités. On pouvait donc provoquer un dépassement de tampon en envoyant un paquet spécifiant un indice de page supérieur à deux.

Il était ainsi possible d'écrire jusqu'à 2048 octets après le début du tampon [16]. Comme dans l'exemple de la section 6.5.2, ce dépassement visait une adresse statique, et permettait d'écraser des variables situées au-delà du tampon sans corrompre les variables situées entre le tampon et les variables écrasées. Il nous a donc fourni un moyen fiable d'exécuter une calculette LMP.

Cette dernière vulnérabilité, présente dans des équipements répandus depuis une vingtaine d'années, et qui aurait immanquablement pu être identifiée par analyse statique ou

16. Le tampon pouvait contenir 24 octets, mais on pouvait en écrire 255.

par des tests unitaires, montre encore une fois à quel point la surface d'attaque contenue dans les puces Bluetooth reste négligée.

6.6.2 Implications sur pour la sécurité

Nous distinguons deux types d'attaques qui pourraient être menées par un *firmware* Bluetooth malicieux : les attaques génériques, et les attaques spécifiques à l'hôte.

Attaques génériques. Les attaques génériques sont possibles indépendamment de l'implémentation de l'hôte, car elles découlent de la spécification Bluetooth. Les principaux risques seraient l'usurpation de l'identité d'un appareil légitime, ou l'interception et la manipulation des échanges. Ces attaques seraient plus simples à mettre en oeuvre car l'attaque doit seulement être calibré selon la version du contrôleur ciblé. Elles sont cependant moins utiles, car on peut obtenir quasiment les mêmes privilèges en exploitant des vulnérabilités intrinsèques à la spécification Bluetooth, comme par exemple BIAS [ATR20].

Attaques spécifiques à l'hôte. Cette classe d'attaque vise l'hôte du modem Bluetooth. Elles pourraient permettre l'exécution de code arbitraire dans l'implémentation des couches supérieures du Bluetooth par l'hôte, par exemple BlueZ [blu] dans le noyau Linux, ou le service Bluedroid [and] d'Android. La portée d'une telle attaque serait maximale puisqu'elle permettrait souvent une totale compromission de l'hôte.

Ces attaques reposent sur la surface supplémentaire accessible au contrôleur malicieux, en particulier la couche HCI, et la couche de transport qui véhicule les commandes HCI. Nous croyons que beaucoup d'implémentations Bluetooth n'ont pas été écrites pour résister à un contrôleur Bluetooth malicieux, et pourraient receler des vulnérabilités exploitables dans ce cas de figure.

Enfin, on pourrait également envisager des attaques de type BadUSB [NKL14] lorsque le contrôleur Bluetooth est branché en USB. C'est plus souvent le cas que l'on ne pourrait le croire, car les puces Bluetooth intégrées aux cartes mères d'ordinateurs sont souvent connectées en HSIC [17].

Les attaques spécifiques à l'hôte sont plus difficiles à mettre en oeuvre, car elles requièrent non seulement de calibrer l'attaque selon la version du contrôleur ciblé, mais aussi selon la version de l'hôte.

Elles sont donc plus praticables dans des environnement logiciels très uniformes où de nombreuses cibles possèdent exactement le même matériel et le même logiciel, comme certaines marques de téléphones ou d'objets connectés.

Attaques inter-puce. Une dernière catégorie d'attaque a été récemment présentée par Classen et al. [CG20]. Ces attaques intitulées Spectra visent le système de coexistence du

17. HSIC est une variante d'USB optimisée pour les circuits imprimés.

Wi-Fi et du Bluetooth. Comme ces deux protocoles partagent la bande RF de 2,4 GHz, des mécanismes permettent d'arbitrer leur coexistence [18].

Un contrôleur Bluetooth malicieux pourrait par exemple abuser de ces mécanismes pour désactiver la puce Wi-Fi, voire en prendre le contrôle. Classen a notamment démontré une élévation de privilèges permettant de prendre le contrôle de la partie Wi-Fi d'un SoC Broadcom mixte qui supporte Wi-Fi et Bluetooth. Un *firmware* Bluetooth malicieux pouvait simplement modifier des de pointeurs de fonction présents dans un segment mémoire partagé avec le CPU Wi-Fi pour détourner son flot d'exécution.

6.6.3 Contremesures

Plusieurs pistes peuvent permettre de réduire les risques posés par les contrôleurs Bluetooth. On peut réduire les risques et chercher à les éliminer.

Détection d'intrusion. Les primitives d'exploitation que nous avons développées requéraient un contrôle précis de l'état de l'appareil ciblé et de sa mémoire, par l'envoi de nombreux paquets LMP différents et nécessitaient parfois d'établir plusieurs connexions successives vers la cible. Cela causait l'envoi d'un grand nombre d'évènements HCI du contrôleur vers l'hôte. Un système de détection d'intrusion comme BIDS [SSH18] qui inspecte le trafic HCI échangé entre l'hôte et le contrôleur pourrait donc les détecter. Même si un attaquant pourrait chercher à réduire sa signature, un HIDS sur la couche HCI lui compliquerait drastiquement la tâche. Cette mesure pourrait être assez rapidement mise en oeuvre dans des ordinateurs ou des ordiphones. Elle serait en revanche plus difficile à mettre en oeuvre dans des systèmes embarqués contraints en mémoire et en calcul.

Mitigation

Les autres mesures que nous suggérons sont nettement plus simples à mettre en oeuvre. D'abord, éteindre le Bluetooth lorsqu'on ne l'utilise pas. Lorsqu'on utilise le Bluetooth, mais que l'on ne s'attend pas à recevoir une connexion distante, on peut quitter l'état *connectable* et empêcher les connexions distantes en désactivant le *page scan*. Cela n'empêche pas l'utilisation normale du Bluetooth : c'est notre ordiphone qui établit la connexion vers un périphérique ; l'inverse se produit plus rarement. Après s'être connecté à un périphérique, on peut par exemple entièrement désactiver le scan d'*inquiry* et de *page* à l'aide de la commande :

```
hciconfig hci0 noscan
```

Cette dernière mesure serait peut-être plus difficile à adopter dans nos téléphones, car la configuration par défaut de leur pile Bluetooth est un sujet particulièrement sensible en 2020, en raison de son utilisation dans les applications de recherche de contacts déployées pour lutter contre la pandémie de COVID-19.

18. cf. IEEE 802.15.2

Éliminer les risques. À notre sens, la mesure de sécurité la plus pertinente reste d'éliminer les risques à la racine, en utilisant des composants de communication moins faillibles.

D'abord, il convient de s'interroger sur le choix du protocole que l'on utilise. Le Bluetooth a l'avantage d'être largement déployé sur nos équipements, et d'intégrer quelques fonctions de sécurité. Il est donc utile pour des objets simples qui n'ont de toute façon pas la capacité d'ajouter des couches de sécurité sur leurs communications, et qui peuvent donc se reposer sur la faible sécurité intégrée au protocole Bluetooth.

Mais de notre expérience, le Bluetooth ne devrait pas être utilisé dans des applications sensibles. Pour des applications nécessitant un haut niveau de sécurité, nous préconisons l'utilisation de protocoles plus simples, et d'assurer les propriétés d'authenticité, de confidentialité et d'intégrité au niveau de la couche applicative.

La dernière piste d'amélioration est celle que nous avons décrite dans ce chapitre. Au cours de notre étude, nous avons souvent eu l'impression de faire face à des *firmwares* qui n'avaient jamais fait l'objet d'une revue de code. En plus d'alerter les constructeurs des vulnérabilité présentes dans leurs composants, il est donc important d'encourager les recherches indépendantes sur la sécurité des composants de communication sans-fil.

6.7 Conclusion

Ce chapitre présentait notre démarche pour évaluer la sécurité des contrôleurs Bluetooth Classic.

Nous sommes parti d'un modèle d'attaque crédible, permettant un totale compromission du composant, et avons recherché des vulnérabilités dans la couche LMP qui répondaient à ce modèle.

Nous avons analysé quatre implémentations différentes de la couche LMP, dans des chipsets Bluetooth très répandus. Au moins trois d'entre eux contenaient des vulnérabilités qui répondaient à ce modèle d'attaque [19]. Nous l'avons démontré en y exécutant des calculettes pilotables à distance.

Nous avons été surpris de constater qu'en 2019, ces vulnérabilités critiques, anciennes, et souvent facilement identifiables n'avaient jamais été signalées.

Seule la vulnérabilité Qualcomm a à ce jour été publiquement corrigée (CVE-2019-14095). Le NIST a classé cette vulnérabilité critique, et lui a attribué un score de sévérité CVSS3 de 9.8, qui est à notre connaissance le plus haut score CVSS attribué à une faille Bluetooth *baseband*.

En réalité, trois des quatre *firmwares* que nous avons audité présentaient des vulnérabilités comparables à la CVE-2019-14095, qui pouvaient causer les mêmes conséquences en partant du même modèle d'attaque.

Selon nous, il convient donc de considérer a priori les contrôleurs Bluetooth comme vulnérables à l'exécution de code arbitraire, et d'exclure systématiquement les périphériques de communication sans-fil de la base de confiance du système. Par exemple, les pilotes

19. Par chance, le seul contrôleur que nous n'avons pas su compromettre est celui qui est intégré à l'ordinateur portable sur lequel fut rédigé ce manuscrit.

logiciels qui gère des modems sans-fil devraient être écrits en tenant compte du risque posé par un composant malicieux.

Pour stimuler la recherche sur la sécurité des *firmwares* Bluetooth, nous avons entrepris l'implémentation d'une pile Bluetooth *baseband* Open source, décrite dans le chapitre suivant.

Chapitre 7

Une couche *baseband* Bluetooth Classic Open source

Après notre analyse de la sécurité des contrôleurs Bluetooth Classic, nous avons cherché à contribuer à la recherche sur leur sécurité.

Les implémentations ouvertes de protocoles de communication sont des outils essentiels à l'étude de leur sécurité. Aujourd'hui, près de 20 ans après l'apparition du Bluetooth, il n'existe pas encore d'implémentation Open source d'un *baseband* Bluetooth Classic.

Il semble que la fermeture de l'écosystème Bluetooth a nuit à sa sécurité, et peut-être conduit les constructeurs à négliger la sécurité de leurs contrôleurs Bluetooth.

Nous avons entrepris le développement d'un *baseband* Bluetooth Classic Open source, pour faciliter l'étude de la sécurité des puces Bluetooth. Nous l'avons basé sur la plateforme Ubertooth, qui est déjà largement utilisée pour l'étude de la sécurité Bluetooth.

Comme il ne supporte que le Bluetooth *Basic Rate* (BR), nous avons nommé ce projet Ubertooth-btbr (UBTBR).

La section 7.1 de ce chapitre présente les principaux concepts de la spécification Bluetooth qui sont implémentés dans UBTBR. Cela permettra de mieux comprendre la section 7.2, qui décrit notre implémentation. Enfin, la section 7.3 illustre les perspectives offertes par UBTBR, en présentant une nouvelle méthode pour *sniffer* des connexions Bluetooth dès leur initiation.

7.1 *Baseband* Bluetooth Classic

Cette section condense les principaux concepts de la spécification Bluetooth *Basic Rate* (BR) implémentés dans UBTBR. Elle est essentiellement extraite de la spécification Bluetooth [1]. Nous la destinons avant tout aux futurs utilisateurs avertis d'UBTBR. Il n'est pas nécessaire de la lire en intégralité pour comprendre le reste de ce chapitre. Nous conseillons cependant de la parcourir au moins une fois avant d'aborder la suite.

1. Core v5.2, Volume 2 : BR/EDR Controller, Part B : Baseband specification

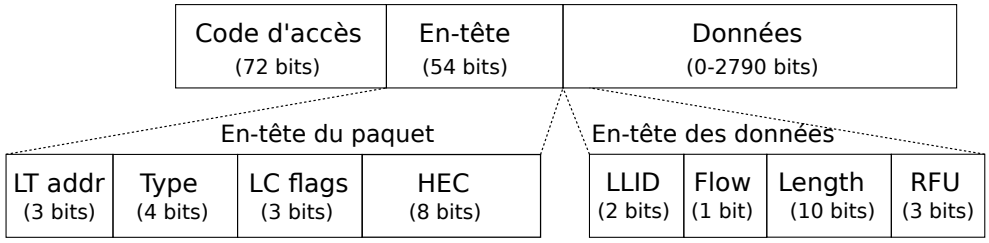


FIGURE 7.1: Formats des paquets Bluetooth Classic

Bases du protocole. Le protocole Bluetooth BR/EDR opère dans la bande ISM de 2,4 GHz. Il y a 79 canaux RF espacés d'1 MHz, de 2402 MHz à 2480 MHz. Le Bluetooth BR utilise une modulation GFSK (Gaussian Frequency-Shift Keying) avec un débit de 1 Msym/s.

Un *piconet* Bluetooth implique au moins deux appareils, un *maître* et un *esclave*. Les communications d'un *piconet* sont synchronisées sur l'horloge du maître (`CLKN`). `CLKN` est un compteur de 28 bit, incrémenté 3200 fois par seconde (chaque $312,5$ μs), qui revient à zero au bout d'environ 23 heures. Chaque communication doit commencer au début d'un créneau horaire (*timeslot*). Il y a 1600 *timeslots* par seconde (de 625 µs), qui commencent lorsque le bit de poid faible de `CLKN` (noté `CLKN_0`) vaut 0. Sur un *piconet*, toutes les communications du maître doivent commencer lorsque `CLKN_1` vaut 0 et les communications des esclaves commencent lorsque `CLKN_1` vaut 1. Les paquets peuvent occuper un ou plusieurs *timeslots*, auquel cas on parle de paquets *multislots*.

Une *Adresse Bluetooth* (`BD_ADDR`) de 48 bits est assignée à chaque appareil Bluetooth. La `BD_ADDR` est découpée en plusieurs parties :

— Lower Address Part (LAP) : bits 0 à 23 ;
— Upper Address Part (UAP) : bits 24 à 31 ;
— Non-Significant Address Part (NAP) : bits 32 à 47.

Le LAP et l'UAP sont utilisée pour paramétrer le fonctionnement de la couche physique et il faut les connaitre pour pouvoir se connecter à un appareil.

7.1.1 Paquets

La figure 7.1 illustre la structure générale d'un paquet Bluetooth Classic. Chaque paquet commence par un *code d'accès*[2] (**AC**), qui peut être suivi d'un préambule et de données. L'AC est composé d'un préambule de 4 bits, du *mot de synchronisation*[3] et d'un *trailer*[4] de 4 bits (seulement présent quand un en-tête suit l'AC). L'en-tête et le trailer sont simplement des séquences alternant des 0 et des 1. Le mot de synchronization est dérivé des 24 bits du LAP (du maître ou de l'esclave selon le canal physique). L'AC complet permet la synchronization, la correction du décalage DC et l'identification du canal.

2. NdT : Access Code
3. NdT : Syncword
4. Remorque

L'en-tête du paquet contient sur l'adresse `LT_ADDR` sur trois bits, le type du paquet sur quatre bits, trois bits indicateurs pour la couche *Link Control* (**LC**) et un *Header Error Check* (**HEC**) sur huit bits. Le HEC est initialisé avec l'UAP du canal et calculé sur les 10 bits de l'en-tête.

Types de paquets

Le paquet le plus court est l'**ID**, qui ne contient que l'AC. Tous les autres types de paquets comportent un en-tête.

Deux types de paquets *communs* ne contiennent qu'un AC suivi d'un en-tête :

— NULL : Paquet envoyé lorsqu'il n'y a pas de données à envoyer, qui ne transporte que la couche LC ;

— POLL : Paquet similaire à NULL, envoyé par le maître pour forcer un esclave à transmettre.

Le dernier type de paquet commun est le *Frequency Hopping Synchronization* (**FHS**). Il transmet les informations nécessaires pour se synchronizer à son émetteur : sa `BD_ADDR` et sa valeur de `CLKN` au début de la transmission. Les données du FHS sont encodées avec FEC2/3.

Les autres types de paquets sont utilisés pour transmettre des données. Les données sont précédées par un en-tête d'un ou deux octets. L'image 7.1 montre le format de l'en-tête de deux octets. Cet en-tête indique la taille des données (sans compter l'en-tête et le CRC), le *Logical Link Identifier* (**LLID**) et un indicateur FLOW.

Correction d'erreur

La *Forward Error Correction* (**FEC**) permet d'améliorer la robustesse du lien et la correction d'erreurs. Deux types de FEC sont utilisées :

— FEC1/3 : chaque bit est répété trois fois ;

— FEC2/3 : 5 bits de parité sont ajoutés à 10 bits d'information.

Le FEC1/3 est utilisé sur tous les en-têtes de paquets et le FEC2/3 est éventuellement appliqué sur les données selon le type de paquet.

UBTBR ne supporte que les paquets *Asynchronous Connection-Oriented* (**ACL**), dont le rôle est de transmettre des données ordinaires.Ils permettent de transmettre des données de différentes tailles, à un débit moyen ou élevé. Le nommage des types indique le débit du paquet et le nombre de *timeslots* qu'il occupe. Les paquets DM sont encodés par FEC2/3 (permettant des communications plus robustes), tandis que les paquets DH ne sont pas encodés (moins robutes, mais débit plus élevé). Le dernier chiffre du type indique le nombre de *timeslots* utilisés par le paquet, par exemple un paquet DH5 occupe cinq *timeslots*. Tous les paquets ACL sont suivis d'un CRC de 16 bits, initialisé avec l'UAP du canal et calculé sur l'ensemble des *données*.

Les paquets ACL ont un rôle prédominant dans le Bluetooth Classic. Ils sont indispensables pour pouvoir établir une connexion et échanger des données.

Les autres types de paquets servent à transporter des données audio ou à améliorer le débit d'une connexion. Ils ne sont pas implémentés dans UBTBR, mais comme ils sont nettement moins importants que les ACL, ce n'est pas très grave à ce stade.

Whitening

Avant la transmission, l'en-tête du paquet et les données subissent un *whitening*. Les bits sont xorés avec une séquence pseudo-aléatoire, pour éliminer les longues séquences de 0 et de 1 et minimiser le décalage DC. La graine utilisée pour générer la séquence de *whitening* dépend du type de canal physique. Sur les canaux *basic piconet* et *adapted piconet*, elle utilise des bits de `CLKN`, tandis que les canaux de *page* et d'*inquiry* utilisent le compteur (x) utilisé pour générer la séquence de sauts de fréquence. Les paquets du canal *Synchronization train* ne subissent aucun *whitening*.

7.1.2 Canaux physiques

Le Bluetooth Classic utilise un étalement de spectre par saut de fréquence (**FHSS**). La fréquence RF d'un canal physique change dans chaque *timeslot*, 1600 fois par seconde. Un canal physique est défini par sa *séquence de sauts* et le *code d'accès* qui préfixe tous ses paquets.

Cinq types de canaux physiques sont définis : inquiry, page, basic piconet, adapted piconet et synchronization scan.

Par convention, dans les canaux d'*inquiry* et de *paging*, l'appareil initiant la procédure est appelé maître et l'appareil répondant est appelé esclave.

Inquiry

Le canal physique d'*inquiry* permet de scanner les appareils voisins *découvrables*. Il utilise un AC réservé : le *General Inquiry Access Code* (**GIAC**). Sa séquence de saut ne dépend que du GIAC et d'un compteur de sauts (x). Cette séquence peut donc être suivie sans avoir à connaitre la `CLKN` du maître. Dans l'état *inquiry*, le maître envoie un paquet ID (avec le GIAC) au début de chaque *timeslot*. L'esclave doit répondre 625 µs plus tard avec un FHS, puis évenutuellement envoyer une *Extended Inquiry Response* (**EIR**) 1250 µs après le FHS, sur la même fréquence. En réalité, grâce à la petitesse d'un ID, le maître peut envoyer deux ID dans chaque *timeslot* (sur deux *tics* consécutifs de `CLKN`) et écouter les deux fréquences de réception correspondantes dans le *timeslot* suivant. À l'heure actuelle, UBTBR n'envoie qu'un seul ID par *timeslot* dans l'état *inquiry*.

Page

Le canal physique *page* est utilisé pour établir une connexion à un appareil voisin *connectable*. L'AC est dérivé du LAP de l'esclave appelé. La séquence de saut de fréquence est similaire à celle de l'*inquiry*, mais utilise la `BD_ADDR` de l'esclave au lieu du GIAC. La procédure de *paging* se compose de quatre étapes. D'abord, le maître envoie des paquets

ID. Lorsque l'esclave détecte un paquet ID, il répond avec un ID (*page response 1*) dans le *timeslot* suivant. Si le maître reçoit la *page response 1*, il transmet enfin un FHS dans le *timeslot* suivant et attend un ID (*page response 2*) en acquittement. Une fois la *page response 2* reçue, le maître peut passer sur le canal *basic piconet* (défini par l'AC et `CLKN` du maître) et commencer à envoyer des messages POLL à l'esclave. L'adresse `LT_ADDR` primaire assignée à l'esclave par le maître est indiquée dans le contenu du FHS.

Basic/Adapted piconet

Les canaux physiques *basic piconet* et *adapted piconet* sont utilisés pour les connexions établies. L'AC est dérivé du LAP du maître du piconet. La séquence de saut dépend des valeurs de `CLKN` et `BD_ADDR` du maître. Sur un *piconet*, la moitié des *timeslots* sont réservés au maître et la seconde moitié peut être utilisé par les esclaves. Les transmissions du maître commencent lorsque `CLKN_1`=1 et les esclaves transmettent lorsque `CLKN_1`=0. Les esclaves doivent vérifier la valeur de `LT_ADDR` et ne traiter que les paquets qui sont adressés à leur propre `LT_ADDR`, ou à la `LT_ADDR` zéro (broadcast). Le premier paquet envoyé par le maître doit être un POLL. La connexion est considérée établie lorsque le maître reçoit une première réponse de l'esclave. Si aucun paquet n'est reçu avant la fin d'un *timeout* [5], les appareils doivent reprendre la procédure de *paging*. Une fois la connexion établie, les participants du *piconet* peuvent échanger des données.

Le canal *adapted piconet* est similaire au *basic piconet*, mais utilise un saut de fréquence adaptatif (AFH) [6]. Certaines fréquences RF peuvent être exclues de la séquence pour éviter les fréquences avec trop d'interférences. Par ailleurs, sur le canal *adapted piconet* l'esclave transmet sur la même fréquence que le maître dans le *timeslot* précédent (*same channel mechanism*) [7].

Synchronisation scan

La canal physique *synchronization scan* permet de transmettre des paquets *synchronization train*, utilisés par le transport logique *Connectionless Slave Broadcast* (CSB). Il n'est pas implémenté dans UBTBR.

7.1.3 Transports et canaux logiques

Le Bluetooth Classic offre différents types de *transports logiques* (**LT**), qui peuvent eux-même véhiculer un ou plusieurs *liens logiques* (**LL**). Il y a quatre types de *transports logiques* : *Asynchronous Connection-Oriented* (**ACL**), *Active Slave Broadcast* (ASB), *(extended) Synchonous Connection Oriented* ((e)SCO) et *Connectionless Slave Broadcast* (CSB).

5. New connection timeout
6. AFH : Adaptive Frequency Hopping
7. Sur le canal *adapted*, les sauts de fréquence sont donc deux fois moins fréquents.

Identification des liens logiques

Les *transports logiques* véhiculent différents *liens logiques* : ACL-C, ACL-U, ASB-C, ASB-U, SCO-S, eSCO-S, PDB et LC.

Le type du *transport logique* est indiqué par la `LT_ADDR` de 3 bits de l'en-tête du paquet. Dans un *piconet*, le maître assigne une `LT_ADDR` *primaire* (non nulle) à chaque esclave. Le trafic ACL utilise toujours la `LT_ADDR` *primaire*. Le trafic multicast ASB utilise pour sa part la `LT_ADDR` réservée zéro. Les (e)SCO utilisent des `LT_ADDR` *secondaires* qui sont allouées en-même temps que le (e)SCO. Le trafic CSB utilise une `LT_ADDR` allouée par le maître.

Le type de *lien logique* est indiqué dans le *Logical Link Identifier* (**LLID**) de deux bits de l'en-tête des données. L'interprétation du LLID dépend du type du transport logique. Sur les transports ACL, il distingue si le paquet est le début d'un message L2CAP (LLID=2), un fragment de continuation L2CAP (LLID=1), ou un message de contrôle LMP (LLID=3).

Asynchronous Connection-oriented

L'ACL est un transport bidirectionnel point-à-point fiable pour échanger des données asynchrones. Il tranporte deux *liens logiques* : l'ACL-C (Contrôle) pour le trafic LMP et l'ACL-U (Utilisateur) pour le trafic L2CAP. Le transport ACL est le seul à être entièrement supporté par UBTBR. Les transports ASB et CSB utilisent également les paquets ACL mais leur support n'est pas complet.

Active Slave Broadcast

L'ASB est un transport unidirectionnel non fiable (les paquets peuvent être transmis plusieurs fois pour améliorer la fiabilité), dirigé du maître vers tous les esclaves. Il transporte les liens logiques ASB-C (Contrôle) et ASB-U (Utilisateur).

(extended) Synchronous Connection-oriented

Les transports SCO et eSCO permettent la transmission de flux de données à un débit constant, en utilisant des *timeslots* réservés qui sont alloués par le maître. Les transports SCO sont typiquement utilisés dans la transmission de voix et ne permettent pas la retransmission des paquets. Les eSCO permettent un nombre limité de retransmissions et sont préférés pour la transmission de données. Les SCO et eSCO transportent les *liens logiques* SCO-S (Stream) et eSCO-S. Ils n'utilisent pas des paquets ACL et leur support n'est pas implémenté dans UBTBR.

Connectionless Slave Broadcast

Les tranports CSB véhiculent le LL *Profile Broadcast Data* (**PBD**), pour diffuser des données (sans garantie de fiabilité) du maître vers un ou plusieurs esclaves. Il est partiellement supporté dans UBTBR.

Link Control.

Le lien *Link Controller* (LC) gère le contrôle du débit et les retransmissions. Il est présent dans chaque en-tête de paquet et comporte trois bits indicateurs :

— FLOW : (contrôle du débit) indique si on est prêt à recevoir des paquets (GO=1, STOP=0).
— ARQN : (acquittement) indique la réception du paquet précédent.
— SEQN : (numéro de séquence), incrémenté après chaque transmission réussie.

Le Link Control n'est requis que dans les paquets point-à-point contenant un CRC.

7.2 Implémentation d'UBTBR

Cette section détaille l'implémentation d'UBTBR. La première partie présente la plateforme Ubertooth qui sert de base à ce projet. La suite décrit l'architecture générale d'UBTBR et nos principaux choix techniques. L'avant dernière partie relate les différentes itérations du développement et du débogage d'UBTBR. Enfin, la dernière partie présente les premiers résultats obtenus.

7.2.1 Matériel

UBTBR est conçu pour fonctionner sur l'Ubertooth-One [OSR+]. C'est une carte de développement USB basée sur une MCU LPC1756 et un émetteur-récepteur RF CC2400. Le LPC1756 possède 256 kB de mémoire flash, 32 kB de SRAM (dont 16 kB sont réservés pour le *bootloader*) et son CPU ARM Cortex-M3 est cadencé à une fréquence de 100 MHz. Le CC2400 peut opérer dans toute la bande ISM de 2,4 GHz et supporte une gamme de débits et de types de modulation. Bien qu'il n'ait pas été spécifiquement conçu pour communiquer en Bluetooth, il est compatible avec la modulation utilisée par les paquets Bluetooth Basic Rate.

L'Ubertooth est donc un outil de choix pour la sécurité Bluetooth et a servi de base à plusieurs travaux sur la sécurité du Bluetooth Low-Energy [Rya13] et Bluetooth Classic [AHX16].

La principale difficulté était d'exécuter les opérations en temps critique suffisamment rapidement sur son CPU Cortex-M3.

7.2.2 Architecture logicielle

Contrairement à un contrôleur Bluetooth ordinaire, UBTBR n'exécute pas toute la pile *baseband* du Bluetooth sur la MCU. Dans UBTBR, seules les opérations à durée critique sont exécutées sur la MCU (partie *firmware*), tandis que les couches supérieures comme LMP et L2CAP peuvent être implémentées sur l'hôte. Nous avons concentré nos efforts sur la partie *firmware* exécutée sur la MCU. La partie hôte est avant tout fournie à titre d'exemple et peut être étendue pour offrir davantage de fonctionnalités.

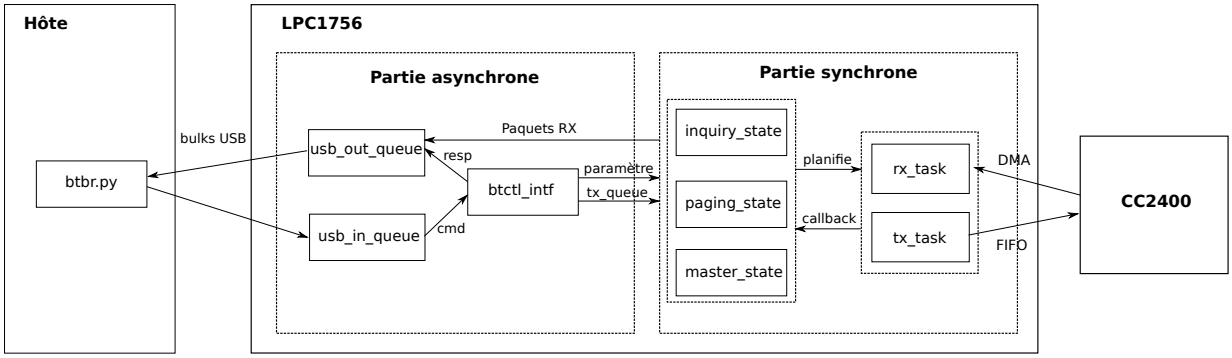


FIGURE 7.2: Architecture générale d'UBTBR

La figure 7.2 présente l'architecture générale d'UBTBR. Le *firmware* d'UBTBR est segmenté en deux parties : une partie *asynchrone* qui gère principalement l'interface de contrôle *BTCTL*, et une partie *synchrone* exécutée par le gestionnaire d'interruption `CLKN` qui est chargée des opérations en temps critique.

La partie synchrone se compose d'**états** et de **tâches**. Les **états** représentent les différents états du *Link Controller* de la spécification Bluetooth BR/EDR, tandis que les **tâches** peuvent être planifiées par différents états pour effectuer des opérations communes, comme la réception ou la transmission d'un paquet.

Par soucis de clarté, la figure 7.2 ne montre que les états et les tâches utilisées par le maître. Il y a d'autres états correspondant au rôle d'esclave : `inquiry_scan_state`, `page_scan_state` et `slave_state`, et une tâche `scan_task` qui est utilisée par les états `inquiry_scan_state` et `page_scan_state` pour attendre un paquet ID et aligner l'horloge sur celle du maître lorsqu'un paquet est détecté.

Modèle d'exécution

L'essentiel de la partie synchrone se déroule dans le gestionnaire d'interruption de l'horloge `CLKN` (appelé toutes les 312.5 µs), implémentée à l'aide d'un *timer* matériel de la MCU. Le principal rôle du gestionnaire d'interruption `CLKN` est d'appeler un ordonnanceur minimaliste. Cet ordonnanceur permet de planifier des appels de fonctions (**événements**) dans un certain nombre de *tics* de `CLKN` et peut gérer des priorités d'appel. Avant de développer notre propre ordonnanceur, nous avons étudié l'implémentation du scheduler TDMA [8] de la couche L1 d'OsmocomBB [9]. En fait, cet ordonnanceur correspondait à notre besoin et nous avons pu l'utiliser tel quel.

UBTBR utilise une politique d'ordonnancement simple : tous les événements planifiés sont exécutés dans le gestionnaire d'interruption `CLKN` par ordre de priorité.

À cause de cette faible granularité temporelle (un appel à l'ordonnanceur toutes les 312,5 µs), il existe un délai entre le déclenchement de l'interruption `CLKN` et le début effectif

8. TDMA : accès multiple à répartition dans le temps
9. OsmocomBB est un *baseband* GSM open-source

d'une transmission ou d'une réception. En fait, ce délai correspond au temps nécessaire pour démarrer une transmission du CC2400 (29 µs). Cela ne pose pas de réel problème dans UBTBR : nous recevons simplement les paquets avec le même délai de 29 µs. Dans la spécification Bluetooth, la transmission d'un paquet doit finir au plus 426 µs après le début du dernier *timeslot* dans lequel il est transmis [10]. Cela laisse une fenêtre d'environ 200 µs (avant le début du prochain *timeslot* de 625 µs) pour traiter le paquet. C'est plus qu'assez pour compenser notre délai de 29 µs.

Tâches de réception et de transmission

Voici la manière dont sont planifiées les tâches de réception et la transmission de paquets.

La tâche de réception (`rx_task`) se compose de trois événements :

— `rx_prepare` : saute sur le prochain canal ; configure le CC2400 en réception ; prépare le codec de réception ; alloue un tampon pour la réception ; démarre le DMA ;

— `rx_execute` : attend la détection d'un paquet par le CC2400 ; décode l'en-tête du paquet ; configure le codec de réception ; décode le premier fragment de données ; décode l'en-tête des données ;

— `rx_decode` : decode le(s) fragments de données ; s'il s'agit du dernier fragment : vérifie le CRC et appele le callback utilisateur, sinon planifie un `rx_decode` supplémentaire.

L'événement `rx_prepare` est exécuté dans le tic `CLKN` qui précède le moment réel de la réception. L'événement `rx_execute` est exécuté dans le tic `CLKN` qui correpond au début réel de la réception. Si le paquet n'est pas entièrement reçu par `rx_execute`, celui-ci planifie un événement `rx_decode` dans le tic suivant. L'événement `rx_decode` peut être appelé zéro fois ou plus, pour décoder davantage de fragments de données. Lorsque le décodage du paquet prend fin, le paquet est passé au *callback* de l'utilisateur.

La tâche de transmission se déroule d'une façon similaire, avec un événement `tx_prepare` appelé dans le tic `CLKN` précédent et un événement `tx_execute` appelé au début de la transmission. La principale différence avec la tâche de réception tient au fait que nous ne somme pas parvenus à faire fonctionner le DMA pour la transmission. À la place, nous avons utilisé le mode *buffered* du CC2400, qui fournit une file (FIFO) de 32 octets. Nous configurons une interruption sur l'indicateur FIFO du CC2400, qui se déclenche lorsque la FIFO est quasiment vide. Lorsque cette interruption survient, nous re-remplissons la FIFO et encodons un fragment de données supplémentaire si besoin.

Traitement des paquets

Les fonctionnalités de traitements de paquets d'UBTBR incluent l'encodage et le décodage de FEC1/3 et FEC2/3 ; le whitening ; la génération du HEC de l'en-tête de paquet ; et le calcul du CRC sur les données.

10. Bluetooth Spec Core v5.2, Vol B, Part 2, figure 2.3

Dans la spécification Bluetooth Classic, il existe plusieurs situations dans lesquelles la réponse doit être transmise dans le *timeslot* qui suit la réception d'un paquet. Dans ce cas, le paquet reçu doit être entièrement décodé avant le début du *timeslot* suivant et le paquet de réponse doit être entièrement encodé avant le début du *timeslot* suivant pour pouvoir être émis à temps.

Prenons l'exemple de la procédure de *paging* : le maître doit transmettre un paquet FHS dans le *timeslot* qui suit la réception d'une première réponse de *paging*. Pour pouvoir transmettre ce FHS, le maître doit préparer l'en-tête du paquet et calculer son HEC, indiquer la valeur de `CLKN` au début de la transmission dans le contenu du paquet, calculer le CRC du contenu, appliquer le *whitening* sur l'en-tête et le contenu du paquet, encoder l'en-tête avec FEC1/3, et encoder le contenu avec FEC2/3. L'esclave doit pour sa part avoir entièrement décodé le FHS et vérifié son CRC avant la fin du *timeslot*, avant de transmettre l'acquittement dans le *timeslot* suivant.

Il y d'autres parties de la spécification dans lesquelles un appareil est obligé de répondre dans le *timeslot* suivant. Par exemple sur un *piconet*, l'esclave doit envoyer son acquittement dans le *timeslot* suivant la réception d'un paquet.

Pour ces raisons, toutes les fonctionnalités de traitement de paquets doivent être réalisées en temps réel. Dans la tâche de réception, le paquet doit être entièrement décodé avant la fin du dernier *timeslot* dans lequel il est transmis. Dans la tâche de transmission, le paquet nous avons au plus un tic de `CLKN` (312,5 µs) pour préparer l'envoi d'un paquet encodé (l'événement `tx_prepare`).

Réaliser le traitement des paquets en temps-réel fut l'un des principaux défis à relever dans le développement d'UBTBR. Pour y parvenir, nous nous sommes concentrés sur deux axes : implémenter des fonctions rapides de traitement du *bitstream* et fragmenter le traitement des paquets.

Implémentation efficace des traitements. La MCU de l'Ubertooth-One est relativement contrainte. Heureusement, la taille de la mémoire flash est immense comparée à la taille de notre *firmware*. Nous avons donc utilisé un classique compromis temps-mémoire en précalculant autant de constantes que possible :

- Décodage FEC1/3 : six bits d'entrée mappés à 2 bits de sortie & l'erreur (64×16 bits) ;
- Encodage FEC1/3 : huit bits d'entrée mappés à 24 bits de sortie (256×32 bits) ;
- FEC2/3 : dix bits d'entrée mappés à 5 bits de codeword (1024×8 bits) ;
- Whitening : sept bits d'état mappés à un mot de 8 bits (128×8 bits), 8×7 bits d'entrée (nombre de bits & etat précédent), mappés à 7 bits d'état suivant (1024×8 bits) ;
- HEC : cinq bits d'entrée mappés à 8 bit de HEC (32×8 bits) ;
- CRC : huit bits d'entrée mappés à 16 bits de CRC (256×16 bits).

Nous avons également utilisé une énorme *lookup table* de 8 kB pour accélérer l'opération *perm5* utilisée dans le calcul de la séquence de saut. Avec toutes ces tables, la taille totale de notre *firmware* est d'approximativement 44 kB, ce qui laisse la plupart de la mémoire

flash disponible pour des développements ultérieurs.

Pour développer les fonctions de traitement du *bitstream*, nous nous sommes beaucoup reposé sur la librairie *libbtbb* [OSRoc], développée dans le cadre du projet Ubertooth. Cette librairie fournit notamment toutes les fonctions d'encodage et de décodage utilisées dans le Bluetooth Classic. Ces implémentations sont peu optimisées. Elles sont donc assez claires et simples à comprendre, mais trop lentes à l'exécution pour qu'on puisse les utiliser dans un *firmware* d'Ubertooth-One. Nous avons utilisé ces fonctions pour générer nos tables de correspondances et tester la validité de nos implémentations.

Fragmentation des traitements. Dans notre première implémentation naïve, les paquets reçus étaient entièrement sauvegardés dans un tampon mémoire avant d'être décodés, et les paquets émis étaient entièrement encodés dans l'événement `tx_prepare` avant le début de la transmission. Bien que cette approche puisse fonctionner avec de petits paquets [11], elle n'était pas suffisante pour gérer des paquets plus gros. Nous ne ne pouvions pas terminer l'encodage et le décodage des paquets en un seul *tic* de `CLKN`.

Pour que les performances soient constantes, nous avons ensuite fragmenté le traitement des paquets. Les paquets reçus sont traités à la volée pendant leur réception. D'abord, l'entête du paquet est entièrement décodé, puis le contenu est décodé par fragments de 10 octets (après décodage). Nous avons choisi une taille de 10 octets car c'était la plus petite taille qui nous permettait de conserver un alignement constant de l'entrée quel que soit le type d'encodage (FEC2/3 ou aucun). Avec un alignement constant de l'entrée, nous pouvons dérouler les boucles de traitement et obtenir un gain supplémentaire de performances.

Nous avons optimisé la tâche de transmission de la même façon. Pendant l'événement `tx_prepare`, seul le premier fragment des données est encodé avant de remplir la FIFO du CC2400. Les fragments suivants peuvent ensuite être encodés dans le gestionnaire d'interruption FIFO. Nous avons utilisé une taille de 80 octets pour avoir suffisamment de données pour remplir deux fois la FIFO (une fois dans `tx_prepare` et une seconde fois dans la première interruption FIFO). Nous pouvons nous permettre une plus grande taille de fragments dans la tâche de transmission, car les fonctions d'encodage nécessitent moins de calculs, n'ayant aucune correction d'erreur à réaliser.

7.2.3 Interface de contrôle

L'Ubertooth-One comporte un *bootloader* qui permet la mise à jour du *firmware* par USB. La moitié de la SRAM de la MCU est réservée au *bootloader*. Il est possible d'enlever le *bootloader* pour disposer de la totalité de la SRAM, mais nous avons préféré le conserver afin de maintenir la fonctionnalité de mise à jour du *firmware* par USB sans matériel supplémentaire. Pour ces raisons, nous maintenons deux interfaces USB distinctes dans UBTBR : l'interface *héritée* (basée sur des transfers de contrôle USB) qui permet à l'outil `ubertooth-dfu` de fonctionner et l'interface `BTCTL` utilisée pour piloter UBTBR. Les messages `BTCTL` échangés entre l'hôte et l'Ubertooth sont transmis via un petit protocole

11. Elle ne pouvait pas non plus gérer correctement la réception de paquets multislots.

ad-hoc dans des *bulks* USB. La conception de l'interface `BTCTL` a été inspirée par la gestion des messages de la `libosmocore` [12] et les tampons réseau de Linux (*skb*). Elle permet une gestion flexible de messages de tailles arbitraires avec un nombre d'opérations réduit.

7.2.4 Code hôte

Avec UBTBR, nous avons concentré nos efforts sur le code *firmware*, et le code hôte est encore à l'état d'ébauche. Le code de l'hôte ne réalise pas d'opérations en temps réel, nous avons donc choisi de l'écrire en Python pour simplifier son développement. Le code de l'hôte comporte la librairie Python `ubtbr`, et l'outil `ubertooth-btbr`. La librairie `ubtbr` prend en charge l'interface de contrôle `BTCTL`, et fournit une implémentation minimaliste du LMP suffisante pour établir une connexion dans le rôle de maître ou d'esclave. Nous avons validé notre code LMP avec des exemples simples, comme une implémentation sommaire de l'outil `l2ping`.

L'outil `ubertooth-btbr` offre une interface en ligne de commandes pour configurer et contrôler le *firmware*. Il est principalement fourni à titre de référence et peut facilement être étendu pour conduire différentes expériences.

7.2.5 Approche du développement

Pour concevoir UBTBR, il nous fallait aussi pouvoir le diagnostiquer précisément. Dans un premier temps, nous avons simplement modifié le code source du *firmware* original de l'Ubertooth One (`bluetooth_rxtx.c`) pour envoyer des paquets ID(GIAC) à intervalles réguliers. L'idée était de mimer le côté maître d'une procédure d'*inquiry*. À ce stade, nous n'utilisions l'Ubertooth que pour transmettre. Nous avons utilisé *Phyltre* pour capturer les paquets et les réponses. Grâce à l'expérience acquise au cours de nos travaux sur le *fingerprinting*, nous avons rapidement pû constater que les paquets que nous transmettions étaient malformées et y remédier. Dès lors, nous avons commencé à recevoir des réponses à notre *inquiry* avec *Phyltre*.

Une fois cette étape terminée, nous avons parallèlement entrepris l'écriture de fonctions optimisées pour le traitement du *bitstream* (que nous avons d'abord évalué sur PC), et d'un nouveau *firmware* basé sur l'ordonnanceur TDMA d'OsmocomBB. Dans un premier temps, UBTBR ne comportait que la procédure d'*inquiry*, que nous avons ensuite formalisé selon l'architecture de la figure 7.2. Cette procédure nous a fourni un exemple simple pour perfectionner les fonctions de traitement de paquets, tout en mettant en place l'architecture générale.

Nous avons ensuite pû poursuivre le développement avec la procédure de *paging*, dans le rôle du maître qui est un peu plus simple à implémenter que celui de l'esclave (le maître n'a pas besoin de ré-ajuster régulièrement son horloge). Cette procédure étant assez semblable à celle d'*inquiry*, nous avons pû l'écrire assez rapidement. Malheureusement, elle ne fonctionnait pas. Pour la déboguer, nous avons utilisé *Phyltre* pour capturer des

12. Utilisée entre autres par OsmocomBB.

établissements de connexion valides et les comparer avec notre implémentation. Cela nous a permis d'identifier une erreur simple dans le calcul du HEC : les bits de la graine du HEC était inversés, mais cela n'apparaissait pas dans la procédure d'*inquiry* car la graine du HEC y est fixée à zéro.

Une fois le rôle du maître implémenté, il était assez simple de réaliser celui de l'esclave, car la plupart des fonctions étaient déjà écrites. En plus des états additionnels `page_scan_state` et `slave_state`, il ne manquait que la tâche `scan_task` et quelques fonctions supplémentaires pour re-synchroniser notre horloge avec celle du maître après chaque réception de paquet.

7.2.6 Résultats

UBTBR supporte les principaux canaux physiques Bluetooth : *Inquiry, Page, Basic/Adapted piconet*. Il peut effectuer un scan Bluetooth ou y répondre, initier ou recevoir des connexions, puis maintenir une connexion active. Il est également possible d'établir une connexion (non sécurisée) vers l'hôte, et d'échanger des paquets L2CAP.

Le *firmware* d'UBTBR comprend moins de 6000 lignes de code C. L'image du *firmware* utilise un peu moins de 44 kB de mémoire flash, et 10 kB de mémoire SRAM. La majeure partie de l'utilisation de la SRAM provient de notre allocateur de *pool*, qui fournit 16 blocs d'environ 380 octets (suffisant pour un paquet ACL complet). Cette consommation modeste de ressources conserve une bonne partie de la mémoire disponible pour des développements ultérieurs.

Performances

Nous avons mesurée les performances d'UBTBR en mesurant le nombre de paquets échangés sur une connexion établie.

Pour tester la qualité du lien dans des conditions idéales, nous mettons en place un *piconet* basic en tant que maître, en envoyons des paquets POLL dans chaque *timeslot* du maître (800 paquets POLL par seconde). L'esclave est censé répondre à chaque POLL avec un paquet NULL. Avec l'Ubertooth-One placé à deux mètres de l'esclave, nous avons reçu environ 533 réponses par secondes (soit environ 33% de paquets perdus dans un sens ou l'autre). La couche de contrôle du lien (LC) prend en charge la retransmission des paquets. La perte de paquets ne constitue donc pas un réel obstacle aux communications.

Il serait cependant préférable de réaliser d'avantage d'expériences pour optimiser les performances, par exemple en faisant varier la configuration du CC2400 et de l'antenne.

Nous n'avons pas observé de perte de connexion causée par des erreurs de synchronisation d'horloge. Nous vérifions que le gestionnaire d'interruption `CLKN` termine son exécution avant le prochain *tic* de `CLKN`, et lançons une exception lorsque ce n'est pas le cas. Cela nous a permis de détecter les potentiels dépassements de *timeslot* et d'adapter le code en conséquence.

Limites et perspectives

UBTBR n'est pas un *contrôleur* Bluetooth complet. Il vise d'abord à créer une couche *baseband* Bluetooth Classic, c'est à dire une implémentation fonctionnelle des canaux physiques et logiques.

Presque tous les canaux physiques sont supportés, à l'exception du *Synchronization chan*. Le support des transports logiques est loin d'être complet, puisque actuellement seuls les paquets ACL sont gérés. La gestion des transports *(e)SCO* nécessitera la gestion de nouveaux types de paquets.

UBTBR ne peut pas encore compléter un appairage, car nous n'avons pas pu implémenter le chiffrement des paquets. Le LPC1756 de l'Ubertooth n'aura certainement pas assez de puissance de calcul pour chiffrer et déchiffrer les paquets. Pour supporter le chiffrement, il faudra donc déporter ces opérations sur l'hôte (ce qui dégraderait les performances), ou envisager de remplacer le LPC1756 par une MCU plus puissante.

Enfin, plusieurs importantes procédures *baseband* restent à implémenter, en particulier la procédure de *Role Switch* qui permet au maître et à l'esclave d'échanger leurs rôles.

UBTBR permet déjà de reproduire la plupart des attaques RCE identifiées dans des contrôleurs Bluetooth Classic. Nous avons recherché des solutions permettant aussi de les détecter. La section suivante présente une méthode de *sniffage* Bluetooth Classic simple, implémentée dans UBTBR, qui permet d'observer le trafic LMP échangé par deux appareils Bluetooth au début d'une connexion.

7.3 Cas d'usage : Surveillance d'une connexion

Dans cette section, nous montrons comment UBTBR peut être utilisé pour surveiller une connexion Bluetooth Classic dès son établissement. Nous proposons une méthode *active* pour acquérir les paramètres du *piconet* au cours de la procédure de *paging*, avant l'esclave légitime. Avec ces paramètres, nous pouvons rejoindre le *piconet* avant ses vrais participants, et capturer les paquets dès le début de la connexion. Cette méthode ne fonctionne pas toujours, car elle repose sur une condition de course qu'il nous faut gagner pour pouvoir se synchroniser au *piconet*. Elle facilite cependant déjà grandement l'étude de l'appairage entre des appareils Bluetooth, et illustre les possiblités offertes par un *baseband* Bluetooth Classic Open Source.

7.3.1 Contexte

Receveur Bluetooth mono-canal

À l'heure actuelle, il est toujours difficile de *sniffer* des connexions Bluetooth Classic sans recourir à de coûteux équipements de test. L'Ubertooth offre une alternative à bas coût, mais il n'est capable d'écouter qu'une seule fréquence à la fois. Le principal défi pour un receveur mono-canal est de parvenir à suivre la séquence de sauts de fréquence.

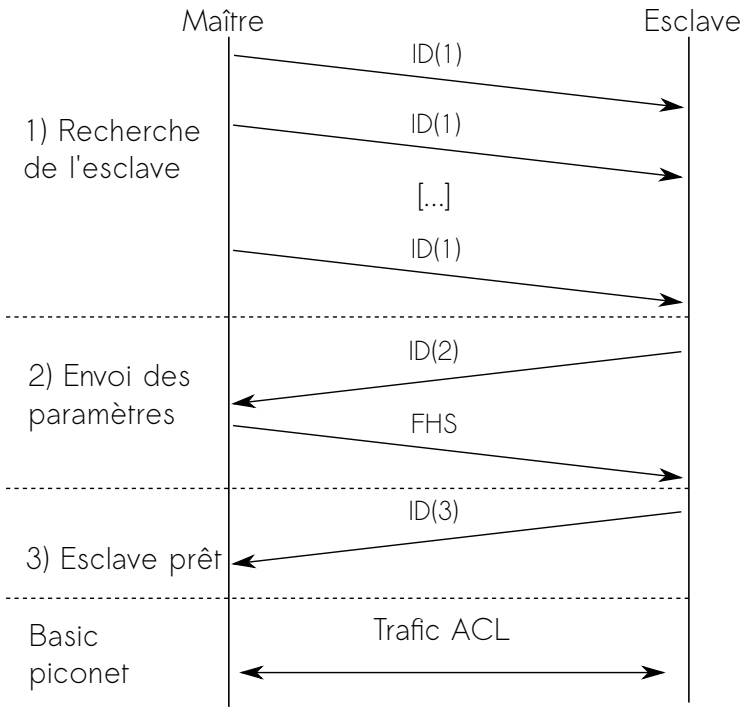


FIGURE 7.3: Procédure de *paging*

Spill et al. [SB07] ont proposé une méthode pour deviner les paramètres de saut en recevant passivement des paquets. Cette méthode a été implémentée dans la base de code de l'Ubertooth, mais reste il reste difficile de suivre les séquences de saut de fréquence adaptatif. En 2016, Albazrqaoe a proposé BlueEar [AHX16] pour suivre des canaux AFH en utilisant deux Ubertooth, un *snooper* qui suit le *piconet* et enregistre le trafic, et un *scout* qui arpente les canaux Bluetooth pour estimer les paramètres de l'AFH, et brouille sélectivement certains canaux pour manipuler la séquence de sauts de fréquence.

Ces méthodes ciblent des connexions déjà établies, qui transportent essentiellement du trafic chiffré. Notre méthode cible les premiers instants d'une connexion, et permet donc de capturer les paquets transmis en clair au cours de l'établissement de la connexion par LMP.

Procédure de *paging*

La procédure de *paging* est utilisée pour établir une nouvelle connexion Bluetooth. C'est le *maître* qui initie la connexion avec la `BD_ADDR` de l'esclave. Cette procédure utilise le canal physique *page*. Lorsque la procédure se termine avec succès, l'esclave rejoint le canal physique *basic piconet* du maître.

La figure 7.3 illustre la procédure de *paging*. D'abord, le maître envoie de nombreux paquets ID contenant l'*Access Code* de l'esclave, tout en observant la séquence de sauts de fréquence du canal physique *page*. Lorsque l'esclave reçoit un ID(1), il répond avec un

ID(2) identique. Le maître transmet alors un paquet FHS qui contient sa propre `BD_ADDR` et sa valeur de `CLKN`. Finalement, l'esclave acquitte le FHS en transmettant un dernier ID(3).

Lorsque ID(3) est reçu par le maître, la connexion est considérée établie, et les deux appareils commmencent à échanger des données sur le chanal physique *basic piconet* du maître. Notez que si le maître ne reçoit pas l'ID(3), il recommencera la procédure à l'étape 1.

7.3.2 Procédure moniteur

Nous avons ajouté une procédure *moniteur* à UBTBR, très similaire au rôle de l'esclave dans la procédure de *paging*.

Pour rejoindre le *piconet* dès le début de la connexion, nous avons besoin du paquet FHS qui contient l'horloge du maître. Mais capturer passivement le paquet FHS envoyé à l'esclave serait difficile : en principe, le FHS n'est transmis qu'une seule fois, sur l'une des 32 fréquences utilisées par le maître sur le canal *page*.

Nous proposons une méthode **active** pour obtenir le FHS :

— 1) Attendre l'ID(1) sur le canal physique *page* ;
— 2) répondre avec ID(2) ;
— 3) recevoir le FHS ;
— 4) rejoindre le *basic piconet*.

La seule différence avec une procédure de *paging* classique est que nous n'envoyons pas l'ID(3), le maître reprendra donc la procédure jusqu'à ce que l'esclave réponde. Dans cette procédure, nous sommes en situation de course avec l'esclave. S'il atteint l'étape 2 avant nous, nous ne recevrons pas le FHS.

Lorsque le moniteur reçoit le FHS, il se synchronise immédiatement au *piconet*, et commence à écouter sur tous les *timeslots* du maître et de l'esclave. Après avoir complété la procédure de *paging*, les appareils rejoignent le *piconet* et commencent à échanger des paquets, qui peuvent être reçus par le moniteur.

Au début d'une connexion, les appareils utilisent le protocole LMP pour échanger leur capacités et procéder à l'appairage avant d'activer le chiffrement. Les premiers paquets échangés ne sont donc ni chiffrés ni authentifiés, et UBTBR peut recevoir leur contenu. Lorsqu'ils ont terminé l'appairage, les appareils activent le chiffrement, et nous ne sommes plus en mesure de comprendre le contenu des paquets reçus.

Réception de paquets chiffrés

Pour activer le chiffrement, les appareils envoyent des `LMP_START_ENCRYPTION_REQ`. Ces paquets et leurs acquittements sont les derniers à être transmis en clair.

Le contenu des paquets suivants est chiffré, par contre leur en-tête ne l'est pas. Pour pouvoir recevoir les paquets chiffrés, nous avons ajouté une option dans la tache de réception d'UBTBR qui permet de recevoir des paquets *bruts*. En temps normal, UBTBR vérifie le CRC des paquets qu'il reçoit, et tient compte de la taille indiquée dans l'en-tête des données. Dans le mode *brut*, les seules opérations réalisées sont le *dewhitening* et la correction

d'erreur. Le CRC des paquets est ignoré, et la taille indiquée dans l'en-tête des données n'est pas prise en compte.

Pour évaluer la taille réelle des paquets, le mode *brut* se repose sur le type de paquet indiqué dans son en-tête. Dans la spécification Bluetooth, un certain nombre de *timeslots* sont réservés pour l'émission d'un paquet en fonction du type de paquet (voir section 7.1), par exemple un paquet DM5 occupera toujours 5 *timeslots*. Pour recevoir des paquets complets, le mode *brut* écoute donc sur la durée maximale possible pour un paquet. On peut ainsi capturer les paquets chiffrés dans leur intégralité, tout en continuant de suivre correctement le *piconet*.

Saut de fréquence adaptatif

Le saut de fréquence adaptatif (AFH) représente un défi pour les solutions d'écoute Bluetooth Classic. Il permet d'adapter la séquence de saut de fréquence d'un canal, pour éviter les fréquences qui comportent trop d'interférences [13]. La configuration de l'AFH est envoyée par le maître dans des paquets `LMP_SET_AFH`, qui contiennent la liste des canaux utilisables.

Pour suivre les canaux physiques AFH, il faut connaitre en permanence la configuration de l'AFH, ce qui est en principe difficile car elle peut changer fréquemment, et les paquets `LMP_SET_AFH` envoyés après l'appairage sont chiffrés.

Dans nos expériences, nous avons observé que le premier paquet `LMP_SET_AFH` est généralement transmis très tôt dans la connexion, avant l'activation du chiffrement. Nous pouvons en tenir compte pour commencer à suivre la séquence de saut adaptatif.

Mais la carte des canaux utilisables est régulièrement modifiée. Après l'activation du chiffrement, le moniteur ne tient plus compte des changements des configuration de l'AFH et le nombre de paquets reçus diminue.

7.3.3 Évaluation

Pour bien comprendre les performances du mode moniteur, nous devons répondre à deux questions. Quelles sont les chances de gagner la course et de recevoir le FHS avant l'esclave ? Une fois synchronisés, quelle proportion des paquets pouvons nous recevoir ?

Il faudrait beaucoup d'expériences pour répondre clairement à ces questions, mais nous n'avons pas pû toutes les réaliser. Dans cette section, nous présentons des expériences préliminaires qui donnent un aperçu des performances du mode moniteur.

Établissement de connexion

Dans la première, nous avons observé les établissements de connexion entre un smartphone et une enceinte. Nous avons placé les deux appareils et l'Ubertooth à 5 mètres les uns des autres, et avons établis 50 connexions successives. À notre surprise, nous avons obtenu le FHS dans 30 connexions sur 50. Il s'avère que le smartphone établissait en fait deux

13. Typiquement, les fréquences utilisées par des points d'accès Wi-Fi proches

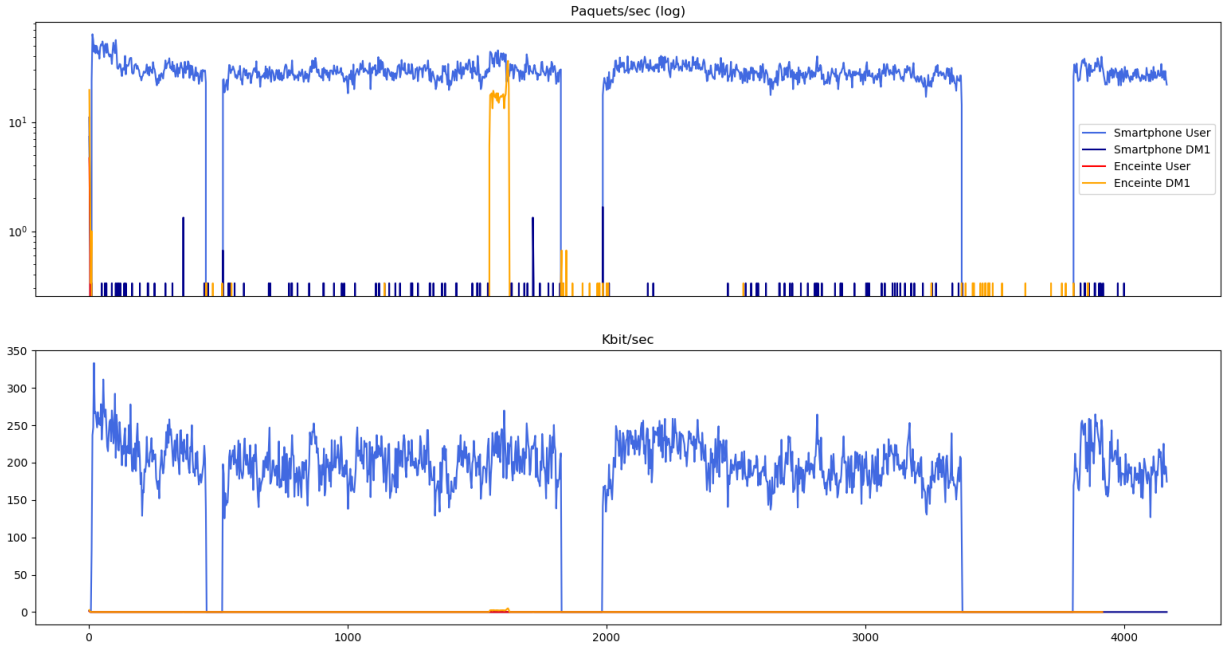


FIGURE 7.4: Estimation du débit entre un smartphone et une enceinte Bluetooth

connexions successives à chaque fois qu'on le connectait à l'enceinte. Dans cette expérience, nous avons en fait gagné la course dans 30% des cas.

Pour estimer le nombre de paquets LMP perdus, nous avons utilisé les paquets de types `LMP_HOST_CONNECTION_REQ`, `LMP_SET_AFH`, et `LMP_SUPERVISION_TIMEOUT`. Dans notre environnement, ces paquets n'étaient envoyés qu'une seule fois par le maître au début de l'appairage. En principe, sur 30 connexions, nous aurions pu recevoir 90 de ces paquets. Nous en avons recu 66, ce qui représente Environ 70% de paquets correctement reçus.

Suivi d'une connexion chiffrée

Nous avons observé des connexions entre un smartphone et une enceinte. Les appareils et l'Ubertooth étaient placés à 5 mètres de distance les uns des autres. Les deux appareils étaient déjà appairés. Pour forcer une reconnexion et permettre au moniteur de démarrer, le Bluetooth du smartphone était relancé à chaque essai.

Pour chaque paquet reçu, nous conservons la valeur de `CLKN` ainsi que la `lt_addr` et le type indiqués dans le header. Le type du paquet permet de connaitre sa taille maximale.

Dans cette expérience, nous ne pouvons pas recevoir le contenu des paquets du smartphone. D'abord parce qu'ils sont chiffrés, mais aussi car nous savons qu'il s'agit de paquets 3-DH5 qui utilisent une modulation non supportée par l'Ubertooth. Le comportement des appareils peut toutefois être observé à travers les flux de données qu'ils s'échangent.

La figure 7.4 présente une mesure du débit échangé entre le smartphone et l'enceinte sur un peu plus d'une heure. Le graphique du haut montre le nombre de paquets échangés

par secondes sur une échelle logarithmique, et le graphique du bas montre une estimation du débit en kilobit par seconde.

Selon la spécification Bluetooth, un paquet LMP doit toujours être de type DM1 [14]. Pour distinguer le trafic LMP des données utilisateur, nous mesurons les paquets DM1 séparément.

L'essentiel du débit provient du smartphone qui transmet la musique à l'enceinte et quelques paquets DM1. À trois reprises, la musique est mise en pause. Le flux musical s'interrompt alors brutalement et l'enceinte commence à transmettre des paquets DM1 par intermittence. À un moment, l'enceinte transmet un grand nombre de paquets DM1 dont nous ignorons la signification.

Observons le débit estimé par le moniteur dans cette capture. Au début de la connexion il est d'environ 250 kbps par seconde, puis baisse et varie autour de 200 kbps. Nous pensons que cette variation du débit est causée par des modification des paramètres de l'AFH.

Estimons maintenant le taux de paquets correctement détectés. Nous ignorons le débit réellement envoyé par le smartphone. D'après la spécification, il devrait être compris entre 229 et 345 kbps dans notre configuration. Cela nous permet de situer le taux de paquets détectés dans cette capture entre 57 et 87%.

Limites et perspectives

Le mode moniteur présente plusieurs limites. D'abord, il ne fonctionne que lorsque UBTBR reçoit le FHS avant l'esclave. Le taux de paquets perdus reste importants et on ne collecte pas la totalité des paquets échangés. L'AFH est encore un obstacle, mais les techniques proposées dans BlueEar [AHX16] pourraient y apporter une solution. Enfin, le *Role Switch* n'est pas encore implémenté dans UBTBR, et le moniteur ne peut donc pas suivre un échange de rôle.

Cependant, cette méthode a aussi des avantages, et offre d'intéressantes perspectives. Son principal atout est qu'elle n'est pas invasive : on peut l'utiliser pour observer des connexions Bluetooth Classic entre tous types d'appareils non modifiés. Par ailleurs, nous croyons que ses performances pourraient être grandement améliorées, en modifiant la configuration du CC2400, ou en utilisant plusieurs Ubertooth pour multiplier les chances de récupérer le FHS. On peut envisager de l'appliquer à la détection d'intrusion, en cherchant à surveiller les connexions établies vers l'appareil à protéger, par exemple pour détecter les attaques LMP décrites dans le chapitre 6.

Les obstacles posées par l'AFH et le suivi direct de connexions actives pourrait aussi être surmontés par le développement d'autres méthodes actives pour acquérir directement l'horloge et la configuration de l'AFH d'un *piconet*. Certaines procédures LMP pourraient être utilisées pour interroger des appareils sur leur configuration, ou l'influencer. Par exemple la procédure de *Role Switch* permet de forcer un appareil à dévoiler la valeur précise de son horloge.

14. Mais UBTBR permet envoyer des messages LMP dans d'autres types de paquets ACL, et beaucoup d'appareils Bluetooth du commerce les acceptent.

Enfin, le mode moniteur ouvre la voie à de nouvelles attaques *actives*, comme l'injection des paquets LMP dans un *piconet*[15], par exemple pour manipuler le processus d'appairage et mener des attaques à la KNOB [ATR19]. Dans un scénario de détection d'intrusion, lorsqu'une alerte est détectée la connexion pourrait également être brouillée afin d'interrompre l'attaque.

7.4 Conclusion

Avec UBTBR, nous proposons une couche *baseband* Bluetooth Classic Open Source, pour faciliter les expériences sur ce protocole. UBTBR est déjà capable de gérer les principaux canaux physiques, et peut communiquer avec des contrôleurs conventionnels. Cela lui permet déjà de reproduire toutes les attaques RCE identifiées dans des contrôleurs Bluetooth Classic.

De plus, son implémentation peut facilement être modifiée pour développer de nouvelles expériences sur les couches les plus basses du Bluetooth Classic. Nous l'avons illustré en développpant une méthode de sniffage simple, ne nécessitant que des modifications simples dans le comportement d'UBTBR.

UBTBR constitue déjà un outil très utile pour étudier la sécurité Bluetooth Classic, et nous espérons qu'il sera encore enrichi dans le futur pour pouvoir explorer d'avantage la spécification Bluetooth. Le code source d'UBTBR est disponible sur le *git* de l'équipe 2XS[16].

15. Nous avons par exemple réussi à injecter des paquets LMP de broadcast sur un piconet.
16. `https://github.com/2xs/ubertooth/tree/monitor`

Chapitre 8

Conclusion

8.1 Synthèse

Cette thèse explore en détail le sujet de la sécurité des communications sans fil utilisées par les objets connectés. En particulier elle s'intéresse au délicat problème de la sécurisation d'objets connectés sans fil qui exécutent du code propriétaire et ne sont pas conçus pour être modifiables par l'utilisateur. Cela inclut non seulement les objets connectés qui relèvent de l'Internet des Objets, mais également les composants intégrés de communication sans fil.

Phyltre est le premier outil que j'ai développé durant cette thèse. Il a vocation à servir de base à un système de détection d'anomalies radio multi-protocoles, en produisant efficacement des indicateurs précis et intelligibles. *Phyltre* permet d'enregistrer et de démoduler rapidement des signaux radio de plusieurs protocoles sans fil couramment utilisés par les objets connectés. Cet outil supporte une large gamme d'interfaces radio, fonctionne sur des CPU conventionnels et ne nécessite qu'un ensemble réduit de dépendances. Il est donc facile à installer et à faire fonctionner.

Ce outil fournit effectivement des informations utiles pour la détection d'anomalies radio. Cependant, dans ce domaine l'un des principaux avantages des radio-logicielles est de permettre de récupérer des informations supplémentaires sur la couche physique. Il était donc important d'explorer ces possibilités.

Mes travaux sur le *fingerprinting* ont montré qu'il était effectivement possible de distinguer le modèle d'un transmetteur Bluetooth par l'observation d'informations uniquement observables au niveau de la couche physique, en se basant sur les données collectées en temps réel à l'aide de *Phyltre*. D'autres chercheurs ont exploré des méthodes de *fingerprinting* Bluetooth, parfois même capables de différencier des appareils du même modèle [HBK06, AUK19]. Cependant ces méthodes nécessitent des fréquences d'échantillonnage très élevées, qui semblent difficiles à mettre en œuvre dans un IDS radio. De plus les empreintes qu'elles produisent prennent la forme de vecteurs de nombres figurant diverses propriétés statistiques du signal. Les alertes produites par un IDS exploitant de telles empreintes seraient donc difficilement interprétables par un administrateur.

Mon approche se veut plus pratique et a pour principal objectif d'enrichir et de contextualiser les informations collectées à l'aide de *Phyltre*, pour pouvoir envisager une détection d'anomalie radio exploitant non seulement des paquets démodulés, mais également des métadonnées provenant de la couche physique. Il est important que ces métadonnées soient de taille réduite pour pouvoir être intégrées à un modèle tenant compte des propriétés physiques des transmetteurs. De plus, j'ai tenu à ce qu'elles soient exprimées dans des grandeurs physiques précises, pour pouvoir être comprises par un être humain. Un exemple qui confirme la viabilité de cette approche est la durée du préamble d'un paquet Bluetooth. Cet indicateur suffit à distinguer la marque d'un adaptateur Bluetooth, peut facilement être mesuré avec une SDR à bas coût et est parfaitement intelligible. Bien que mes techniques de *fingerprinting* ne permettent que de distinguer le modèle d'un transmetteur et pas des appareils du même modèle, elles constituent une piste crédible pour améliorer la détection d'anomalies radio.

Mon approche de détection d'intrusion vise à détecter des attaques évoluées, pouvant intervenir sur des couches qu'il est difficile de surveiller par un HIDS. Les expériences sur le *fingerprinting* ont porté sur le Bluetooth Classic. Ce protocole est particulièrement intéressant, notamment car c'est le premier protocole destiné aux objets connectés à avoir été très largement adopté en étant intégré à tous les téléphones mobiles depuis 2006. En 2019, très peu de vulnérabilités avaient été signalées dans des contrôleurs Bluetooth Classic et aucune d'entre elles ne permettaient l'exécution de code arbitraire.

Pour montrer que mon approche de détection d'intrusion est pertinente, j'ai évalué la sécurité des contrôleurs Bluetooth. J'ai défini un modèle d'attaque reposant sur des prérequis minimes [1] (identiques à ceux de Blueborne [SV17]) et conduisant à une complète compromission du contrôleur. L'implémentation de la couche LMP concentre la majeure partie de la surface d'attaque d'un contrôleur Bluetooth Classic. J'ai donc procédé à la rétro-ingénierie de cette couche dans les *firmwares* de contrôleurs de quatre constructeurs différents et recherché des vulnérabilités permettant d'en prendre le contrôle à distance. Dans trois des contrôleurs testés des vulnérabilités permettant une prise de contrôle à distance ont été identifiées. L'implémentation d'attaques réalistes a permis de le démontrer. Une de ces vulnérabilités concernait tous les processeurs mobiles Qualcomm, soit environ 30 % des téléphones mobiles vendus dans le monde [2]. Elle a été corrigée suite nos travaux.

Enfin, un outil de sécurité essentiel m'a fait cruellement défaut dans mes expériences sur le Bluetooth Classic. Il s'agit implémentation ouverte de la couche *baseband*, capable de transmettre et de recevoir des paquets en temps réel et de communiquer avec des contrôleurs Bluetooth conventionnels. Le seul outil s'en approchant était InternalBlue [MCSH19], qui permet de modifier le fonctionnement de contrôleurs Bluetooth Broadcom/Cypress. Cependant il n'offre pas un total contrôle sur les couches *baseband* et il semble impossible de l'utiliser pour manipuler le fonctionnement de la couche de contrôle du lien. De plus, ce projet est complexe à utiliser et à faire évoluer car il nécessite une compréhension intime du fonctionnement des contrôleurs Bluetooth Broadcom/Cypress qui ne peut

1. Le contrôleur cible est allumé et connectable. L'attaquant connaît l'adresse de la cible.
2. https://www.counterpointresearch.com/qualcomm-leads-market-despite-losing-share-to-hisilicon-q2-2020/

être acquise que par une rétro-ingénierie en profondeur de leur *firmware* J'ai donc conçu UBTBR, un *baseband* Bluetooth Classic Open Source basé sur l'Ubertooth-One, une plateforme déjà largement utilisée pour l'expérimentation sur le protocole Bluetooth. UBTBR offre plusieurs avantages : il permet d'implémenter des attaques sur la couche LMP avec un matériel et un logiciel entièrement maîtrisés ; il offre une implémentation de référence du Bluetooth Classic, qui permet d'acquérir rapidement une compréhension pratique des couches basses de ce protocole. Enfin, il crée des perspectives intéressantes pour développer de nouvelles techniques d'attaques ou de défense. À titre d'illustration, j'ai présenté une nouvelle technique active pour surveiller une connexion Bluetooth dès son établissement.

8.2 Limites et travaux futurs

8.2.1 Outillage

Les outils que j'ai développés méritent d'être complétés et évalués plus en profondeur. Des fonctions importantes font encore défaut pour qu'UBTBR puisse d'implémenter un contrôleur Bluetooth complet. D'abord, il faudrait implémenter le chiffrement des paquets, ce qui constitue un défi au vu de la faible puissance de calcul de l'Ubertooth. Cela peut être résolu soit en utilisant un processeur plus puissant, soit en déportant les opérations de génération de clés ou de chiffrement sur l'hôte. Enfin, il faudrait une implémentation complète de la couche LMP, ce qui ne devrait pas poser de problème particulier.

L'outil *Phyltre* pourrait également être mieux évalué, pour bien mesurer la puissance de calcul nécessaire selon la configuration : largeur de la bande de fréquence capturée, nombre de canaux et protocoles. Pour pouvoir utiliser *Phyltre* comme sonde de détection d'intrusion, il faudrait également considérer l'utilisation de processeurs embarqués. Par exemple, des ordinateurs mono-cartes basse consommation, ou encore des GPU. Par exemple, les systèmes NVIDIA Jetson [3] disposent de puissants GPU pour une faible consommation d'énergie et offrent donc une piste intéressante pour créer une sonde SDR embarquée.

8.2.2 *Fingerprinting*

De nouvelles expériences peuvent être menées pour mieux comprendre les possibilités et les limites du *fingerprinting* sur la couche physique. Nous avons montré que la température influe fortement sur le décalage de la fréquence porteuse d'un transmetteur Bluetooth. On pourrait mesurer l'effet de la température sur d'autres caractéristiques. Enfin, d'autres facteurs peuvent influer sur ces empreintes. Par exemple, on peut s'interroger sur l'effet de la tension de la batterie ou le vieillissement de l'appareil.

3. `https://www.nvidia.com/fr-fr/autonomous-machines/embedded-systems/`

8.2.3 Détection d'intrusion

Dans mes travaux sur la détection d'intrusion radio, je me suis penché en détail sur deux briques élémentaires. D'une part la capture de signaux multi-canaux et multi-protocoles à l'aide de SDR, et d'autre part le *fingerprinting* sur la couche physique. Ces travaux fournissent donc la base d'une sonde SDR pour la détection d'intrusion sans fil. Cependant, je n'ai pas conçu de réelle solution de détection d'intrusion basée sur cette sonde.

Pour réellement valider la viabilité de cette sonde, il faudrait donc développer des moteurs d'analyse pour la détection d'attaques. Plusieurs méthodes peuvent être étudiées. D'abord, on pourrait tenter détecter des anomalies en exploitant directement les informations brutes remontées par notre sonde, à savoir le contenu des paquets démodulés lorsqu'il est disponible, les métadonnées correspondantes (identifiants, taille des paquets, etc.), ainsi que les métadonnées de la couche physique (numéro du canal, RSSI, décalage de fréquence, taille du préambule, etc.). Par exemple, Elasticsearch et Kibana [4] permettent respectivement d'archiver tous types de données, de les visualiser, de les analyser et d'y détecter des anomalies.

À plus long terme, on peut aussi travailler sur l'élaboration d'indicateurs de plus haut niveau, pour faciliter la détection d'intrusion. Cela nécessiterait de concevoir un moteur d'analyse intermédiaire, qui prendrait en entrée des suites de paquets bruts remontées par notre sonde et fournirait en sortie des informations plus concises. Par exemple une liste des connexions actives, le volume et la périodicité des données qui sont échangées, ou encore des évènements comme l'établissement ou la fin d'une connexion ainsi que les empreintes radiométriques des appareils impliqués. Cela réduirait le volume des informations remontées par les sondes, tout en les rendant plus intelligibles. Enfin, cela permet d'envisager la conception d'un IDS basé sur des règles. On pourrait par exemple vérifier que seuls les appareils autorisés établissent des connexions et s'assurer que le volume et la direction du trafic respectent un modèle prédéfini.

4. `https://www.elastic.co/fr/kibana`

Résumé

Cette thèse porte sur la sécurité des communications sans fil, appliquée aux équipements tels que les téléphones mobiles, les ordinateurs portables, ou les objets communicants relevant de l'Internet des Objets.

Aujourd'hui, les communications sans fil sont réalisées à l'aide de composants intégrés (modem), qui peuvent eux-même être la cible d'attaques. Effectivement, ces modem peuvent contenir des logiciels, au code fermé, qui sont peu audités et peuvent recéler des vulnérabilités.

Au cours de cette thèse, nous avons poursuivi deux approches complémentaires qui visent à adresser le problème de la sécurité des modem sans fil. La première consiste à détecter les attaques pour mitiger les risques posés par les vulnérabilité ; la seconde à savoir identifier et à corriger ces vulnérabilités afin d'éliminer les risques.

Les modem sans fil posent des contraintes particulières pour les systèmes de détection d'intrusion (IDS). De fait, si le modem risque d'être compromis, le système d'exploitation (OS) ne peut pas faire confiance aux informations qu'il remonte : le modem n'est pas fiable. Il est ainsi délicat de détecter des attaques sans fil depuis l'OS, car il ne dispose d'aucune source d'information fiable sur laquelle baser cette détection.

Dans ce contexte, il est préférable de réaliser la détection d'intrusion au niveau du réseau, en capturant directement les signaux échangés sans fil. Cependant, il n'est pas toujours simple de récupérer les signaux qui nous intéressent. Aujourd'hui, les équipements supportent une multitude de normes de communication différentes. Cette hétérogénéité représente un défi pour les solutions de capture. De plus, certains protocoles se prêtent mal à une capture passive de leurs échanges, et sont parfois même spécifiquement conçus pour l'empêcher. Enfin, les données sont généralement chiffrées, ce qui constitue un obstacle supplémentaire pour les IDS.

Les radio logicielles peuvent répondre en partie aux défis posés par cette diversité. Elles se composent d'une partie matérielle, mais surtout de logiciel, qui peut être adapté pour recevoir des signaux de n'importe quel standard - dans les limites du matériel.

Dans cette thèse, nous présentons une radio-logicielle spécialement concue pour permettre la capture et l'analyse d'une bande de fréquence donnée, afin d'identifier et d'étiqueter les signaux présents. Il s'agit d'une brique élémentaire pour construire des systèmes de détection d'intrusion sans-fil.

Par ailleurs, les radio-logicielles traitent les signaux au niveau de leur représentation physique. Cela leur permet de collecter des informations supplémentaires, qui n'auraient

pas été accessibles si on avait utilisé un modem conventionnel pour capturer les signaux. Dans cette thèse, nous décrivons des méthodes permettant d'identifier le modèle d'un appareil Bluetooth en analysant la représentation physique des paquets qu'il transmet.

Dans la seconde partie de cette thèse, nous avons analysé les micrologiciels de plusieurs modem Bluetooth, afin d'identifier des vulnérabilités permettant d'en prendre le contrôle à distance. Cela nous a permis de découvrir plusieurs vulnérabilités exploitables dans des modem très largement utilisés. Dans un second temps, nous avons développé un modem Bluetooth libre et Open Source qui permet d'interagir avec de véritables modem pour faciliter la recherche et développement sur leur sécurité.

Summary

This thesis focuses on the security of wireless communications, as used on devices such as mobile phones, laptops, or connected devices that make up the Internet of Things.

Nowadays, wireless communications are carried out using integrated components (modem), which can themselves be the target of attacks. Indeed, these modems contain Closed Source software, that are poorly audited, and may have flaws.

During this thesis, we pursued two complementary approaches that aim to address the problem of wireless modems security.

The first is to detect attacks in order to mitigate the risks posed by vulnerabilities ; the second is to identify and correct these vulnerabilities in order to eliminate the risks.

Wireless modems pose particular constraints for Intrusion Detection Systems (IDS). In fact, if the modem is at risk of being compromised, the operating system (OS) cannot trust the information it is sending back : the modem is unreliable. This makes it difficult to detect wireless attacks from the OS, as it has no reliable source of information on which to base detection.

In this context, it is preferable to perform intrusion detection at the network level, by directly capturing the signals exchanged wirelessly. However, it is not always easy to recover the signals of interest. Today's equipment supports a multitude of different communication standards. This heterogeneity represents a challenge for capture solutions. In addition, some protocols do not lend themselves well to passive capture of their exchanges, and are sometimes even specifically designed to prevent it. Finally, data is usually encrypted, which is an additional obstacle for intrusion detection systems.

Software Defined Radio (SDR) can partly meet the challenges posed by this diversity. They consist of a hardware part, but above all of software, which can be adapted to receive signals of any standard - within the limits of the material.

In this thesis, we present a SDR specifically designed to allow the capture and analysis of a given frequency band, in order to identify and label the signals present. It is an elementary building block for building wireless intrusion detection systems.

In addition, software radio processes signals in terms of their physical representation. This allows them to collect additional information, which would not have been accessible if a conventional modem had been used to capture the signals. In this thesis, we describe methods to identify the model of a Bluetooth device by analysing the physical representation of the packets it transmits.

In the second part of this thesis, we analysed the firmware of several Bluetooth modems,

in order to identify vulnerabilities that would allow remote control. This allowed us to discover several exploitable vulnerabilities in widely used modems. Finally, we developed a free and open-source Bluetooth modem that allows interaction with real-world modems to facilitate research and development on their security.

Bibliographie

[afl] American fuzzy loop. `https://lcamtuf.coredump.cx/afl/`. Accessed : 2020-11-18.

[AHX16] Wahhab Albazrqaoe, Jun Huang, and Guoliang Xing. Practical Bluetooth Traffic Sniffing. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services - MobiSys '16*, pages 333–345, 2016.

[Air] Louis Granboulan Airbus. cpu_rec. `https://github.com/airbus-seclab/cpu_rec`. Accessed : 2020-07-01.

[and] Android bluetooth architecture. `https://source.android.com/devices/bluetooth`.

[Ang19] Hugues Anguelkov. Reverse-engineering Broadcom wireless chipsets, 2019.

[apa] Apache NimBLE. `https://mynewt.apache.org/`.

[Art17] Nitay Artenstein. Broadpwn : Remotely Compromising Android and iOS via a Bug in Broadcom's Wi-Fi Chipsets, 2017.

[ATR19] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Low Entropy Key Negotiation Attacks on Bluetooth and Bluetooth Low Energy. *Cryptology ePrint Archive*, (Report 2019/933), 2019.

[ATR20] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. BIAS : Bluetooth Impersonation AttackS. *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 549–562, 2020.

[AUK19] Aysha M. Ali, Emre Uzundurukan, and Ali Kara. Assessment of Features and Classifiers for Bluetooth RF Fingerprinting. *IEEE Access*, 7 :50524–50535, 2019.

[Bar48] M. S. Bartlett. Smoothing periodograms from time-series with continuous spectra. *Nature*, 161(4096) :686–687, may 1948.

[BBGO08] Vladimir Brik, Suman Banerjee, Marco Gruteser, and Sangho Oh. Wireless device identification with radiometric signatures. *Proceedings of the Annual International Conference on Mobile Computing and Networking, MOBICOM*, pages 116–127, 2008.

[Ben17] Gal Beniamini. Project Zero : Over The Air : Exploiting Broadcom's Wi-Fi Stack, 2017.

[BF] David Berard and Synacktiv Fargues, Vincent. How to design a Baseband debugger. In *SSTIC 2020*, pages 1–14.

[BF20] David Berard and Vincent Fargues. How to design a Baseband debugger. In *SSTIC*, pages 1–14, 2020.

[Blo04] Eric Blossom. Gnu radio : Tools for exploring the radio frequency spectrum. *Linux J.*, 2004(122) :4, June 2004.

[blu] Bluez : Official linux bluetooth protocol stack. `http://www.bluez.org/`.

[Blu19] Bluetooth SIG. Expedited Errata Correction 11838 : Encryption Key Size Updates. Technical report, 2019.

[BMP+] Ravishankar Borgaonkar, Andrew Martin, Shinjo Park, Altaf Shaik, and Jean-Pierre Seifert. White-Stingray : Evaluating IMSI Catchers Detection Applications.

[BSSD13] Bastian Bloessl, Michele Segata, Christoph Sommer, and Falko Dressler. An IEEE 802.11a/g/p OFDM Receiver for GNU Radio. In *ACM SIGCOMM 2013, 2nd ACM Software Radio Implementation Forum (SRIF 2013)*, pages 9–16, Hong Kong, China, 8 2013. ACM.

[CG20] Jiska Classen and Francesco Gringoli. Spectra : New Wireless Escalation Targets. In *DEF CON 28*, 2020.

[Cog] Cognosec. Secbee. `https://github.com/Cognosec/SecBee`. Accessed : 2020-08-05.

[CVPH17] Ronny Chevalier, Maugan Villatel, David Plaquin, and Guillaume Hiet. Co-processor-based behavior monitoring. In *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACM, December 2017.

[Del11] Guillaume Delugré. Reverse engineering a Qualcomm baseband. In *28th Chaos Communication Congress*, 2011.

[DGHH+15] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with PANDA. *ACM International Conference Proceeding Series*, 08-December-2015, 2015.

[DPM11] Loïc Duflot, Yves Alexis Perez, and Benjamin Morin. What if you can't trust your network card ? *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6961 LNCS :378–397, 2011.

[DPVL10] Loïc Duflot, Yves-alexis Perez, Guillaume Valadon, and Olivier Levillain. Can you still trust your network card ? In *CanSecWest*, 2010.

[ea11] F. Pedregosa et al. Scikit-learn : Machine learning in Python. *Journal of Machine Learning Research*, 12 :2825–2830, 2011.

[FJ] Matteo Frigo and Steven G. Johnson. fftw. `http://www.fftw.org`. Accessed : 2020-08-10.

[Fou] The Linux Foundation. toe. `https://wiki.linuxfoundation.org/networking/toe`. Accessed : 2020-07-10.

[Gae] Joseph D. Gaeddert. liquid-dsp. `https://liquidsdr.org/`. Accessed : 2020-08-10.

[GMGSS+16] Ismael Gomez-Miguelez, Andres Garcia-Saavedra, Paul D. Sutton, Pablo Serrano, Cristina Cano, and Doug J. Leith. srsLTE. In *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization*. ACM, October 2016.

[GRS13] Nico Golde, Kevin Redon, and Jean-Pierre Seifert. Let me answer that for you : Exploiting broadcast information in cellular networks. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 33–48, Washington, D.C., August 2013. USENIX Association.

[GSM06] GSMA. Withdrawal of A5/2 from Handsets Deadline (S3-060751), 2006.

[HAX14] J. Huang, W. Albazrqaoe, and G. Xing. BlueID : A practical system for bluetooth device identification. In *INFOCOM*. IEEE, April 2014.

[HBK06] Jeyanthi Hall, Michel Barbeau, and Evangelos Kranakis. Detecting Rogue Devices in Bluetooth Networks using Radio Frequency Fingerprinting. *International Conference on Communications and Computer Networks (IASTED)*, 2006.

[HLBGH20] Etienne Helluy-Lafont, Alexandre Boe, Gilles Grimaud, and Michael Hauspie. Bluetooth devices fingerprinting using low cost SDR. In *2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC)*. IEEE, April 2020.

[HMT+] Grant Hernandez, Marius Muench, Tyler Tucker, Hunter Searle, Weidong Zhu, Patrick Traynor, Kevin Butler, Grant Hernandez, and Marius Muench. Emulating Samsung' s Baseband for Security Testing.

[JOA+19] Hossein Jafari, Oluwaseyi Omotere, Damilola Adesina, Hsiang Huang Wu, and Lijun Qian. IoT Devices Fingerprinting Using Deep Learning. *Proceedings - IEEE Military Communications Conference MILCOM*, 2019-October :901–906, 2019.

[Jom18] Narjes Jomaa. *Le co-design d'un noyau de système d'exploitation et de sa preuve formelle d'isolation*. PhD thesis, 2018. Thèse de doctorat dirigée par Grimaud, Gilles et Nowak, David Informatique et applications Lille 1 2018.

[Kov20] Veronica Kovah. Finding New Bluetooth Low Energy Exploits via Reverse Engineering Multiple Vendors ' Firmwares Hello World ! In *Black Hat USA*, 2020.

[LSSS09] Kaushik Lakshminarayanan, Samir Sapra, Srinivasan Seshan, and Peter Steenkiste. RFDump. *Proceedings of the 5th international conference on Emerging networking experiments and technologies - CoNEXT '09*, page 253, 2009.

[Mar99] Martin Roesch. Snort – Lightweight Intrusion Detection for Networks. In *13th USENIX Conference on System Administration*, pages 229–238, Seattle, Washington, 1999. USENIX Association.

[Mav05] Nikos Mavrogiannopoulos. On Bluetooth (TM) security. 2005.

[MCSH19] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. Internalblue – Bluetooth binary patching and experimentation framework. In *MobiSys 2019 - Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pages 79–90, 2019.

[MMH+17] Markus Miettinen, Samuel Marchal, Ibbad Hafeez, N. Asokan, Ahmad Reza Sadeghi, and Sasu Tarkoma. IoT SENTINEL : Automated Device-Type Identification for Security Enforcement in IoT. *Proceedings - International Conference on Distributed Computing Systems*, pages 2177–2184, 2017.

[MNFB18] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar 2 : A Multi-Target Orchestration Platform . (February) :1–11, 2018.

[MSP20] Dominik Maier, Lukas Seidel, and Shinjo Park. BaseSAFE : Baseband sanitized fuzzing through emulation. *WiSec 2020 - Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 122–132, 2020.

[Mun13] Sylvain Munaut. Further hacks on the Calypso platform. *29th Chaos communication congress*, 2013.

[NKK+14] Navid Nikaein, Raymond Knopp, Florian Kaltenberger, Lionel Gauthier, Christian Bonnet, Dominique Nussbaum, and Riadh Ghaddab. Demo : OpenAirInterface. In *Proceedings of the 20th annual international conference on Mobile computing and networking*. ACM, September 2014.

[NKL14] Karsten Nohl, Sascha Krissler, and Jakob Lell. BadUSB - On accessories that turn evil. In *Black Hat USA*, 2014.

[Noh09] Karsten Nohl. Subverting the security base of GSM. In *HAR 2009*, 2009.

[OCo08] Terrence OConnor. *Bluetooth Intrusion Detection*. PhD thesis, 2008.

[OSR+] Michael Ossmann, Dominic Spill, Mike Ryan, Will Code, Jared Boone, and other contributors. Ubertooth project. `https://github.com/greatscottgadgets/ubertooth/`. Accessed : 2020-04-22.

[OSRoc] Michael Ossmann, Dominic Spill, Mike Ryan, and other contributors. Libbtbb. `https://github.com/greatscottgadgets/libbtbb/`. Accessed : 2020-09-08.

[Pag10] Chris Paget. Practical Cellphone Spying Before we start : Privacy. In *DEFCON 18*, 2010.

[PLD14] Jean-Michel Picod, Arnaud Lebrun, and Jonathan-Christofer Demay. Bringing Software Defined Radio to the Penetration Testing Community. *Black Hat USA*, 2014 :1–7, 2014.

[PN18] Johannes Pohl and Andreas Noack. Universal Radio Hacker : A Suite for Analyzing and Attacking Stateful Wireless Protocols. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[qem] Qemu. `https://www.qemu.org/`. Accessed : 2020-11-17.

[Qua19] Quarkslab. Breaking Samsung's ARM TrustZone. In *BlackHat*, 2019.

[RAA+18] Jonathan Roux, Eric Alata, Guillaume Auriol, Mohamed Kaaniche, Vincent Nicomette, and Romain Cayre. RadIoT : Radio communications intrusion detection for IoT - A protocol independent approach. *NCA 2018 - 2018 IEEE 17th International Symposium on Network Computing and Applications*, pages 1–8, 2018.

[RCGH20] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein : Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. In *USENIX Security Symposium*, 2020.

[Riq15] Damien Riquet. *DISCUS : une architecture de détection d'intrusions réseau distribuée basée sur un langage dédié*. PhD thesis, 2015. Thèse de doctorat dirigée par Grimaud, Gilles et Hauspie, Michaël Informatique Lille 1 2015.

[RML+17] Pieter Robyns, Eduard Marin, Wim Lamotte, Peter Quax, Dave Singelee, and Bart Preneel. Physical-layer fingerprinting of LoRa devices using supervised and zero-shot learning. *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec 2017*, pages 58–63, 2017.

[RMLP19] Sreeraj Rajendran, Wannes Meert, Vincent Lenders, and Sofie Pollin. Unsupervised Wireless Spectrum Anomaly Detection with Interpretable Features. *IEEE Transactions on Cognitive Communications and Networking*, PP(c) :1, 2019.

[RTM10] Donald R. Reising, Michael A. Temple, and Michael J. Mendenhall. Improved wireless security for gmsk-based devices using rf fingerprinting. *International Journal of Electronic Security and Digital Forensics*, 3(1) :41–59, 2010.

[Rya13] Mike Ryan. Bluetooth : With low energy comes low security. *7th USENIX Workshop on Offensive Technologies, WOOT 2013*, 2013.

[SB07] Dominic Spill and Andrea Bittau. BlueSniff : Eve meets Alice and Bluetooth. *WOOT '07 Proceedings of the first USENIX workshop on Offensive Technologies*, page 10, 2007.

[SBA+15] Altaf Shaik, Ravishankar Borgaonkar, N Asokan, Valtteri Niemi, and Jean-Pierre Seifert. Practical Attacks Against Privacy and Availability in 4G/LTE Mobile Communication Systems. 2015.

[Sch06] Thomas Schmid. Gnu radio 802.15.4 en- and decoding. 01 2006.

[SF02] Robin Sommer and Anja Feldmann. NetFlow. In *Proceedings of the second ACM SIGCOMM Workshop on Internet measurment - IMW '02*. ACM Press, 2002.

[SLH11] Zhiyuan Shi, Man Liu, and Lianfen Huang. Transient-based identification of 802.11b wireless device. *2011 International Conference on Wireless Communications and Signal Processing, WCSP 2011*, 2011.

[SMT17] Sandra Siby, Rajib Ranjan Maiti, and Nils Ole Tippenhauer. IoTScanner : Detecting privacy threats in IoT neighborhoods. *IoTPTS 2017 - Proceedings of the 3rd ACM International Workshop on IoT Privacy, Trust, and Security, co-located with ASIA CCS 2017*, pages 23–30, 2017.

[Spa09] Dieter Spaar. A practical DoS attack to the GSM network. In *DeepSec*, 2009.

[srl13] srlabs.de. Catchercatcher. `https://opensource.srlabs.de/projects/mobile-network-assessment-tools/wiki/CatcherCatcher`, 2013. Accessed : 2020-10-19.

[SSH18] Pratik Satam, Shalaka Satam, and Salim Hariri. Bluetooth intrusion detection system (BIDS). In *2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA)*. IEEE, October 2018.

[sur] Suricata ids. `https://suricata-ids.org/`. Accessed : 2020-11-06.

[SV17] Seri, Ben (Armis, Inc.) and Vishnepolsky, Gregory (Armis, Inc.). BlueBorne Technical White Paper. Technical report, 2017.

[SW05] Yaniv Shaked and Avishai Wool. Cracking the Bluetooth PIN. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services - MobiSys '05*, page 39, 2005.

[SWH15] Matthias Schulz, Daniel Wegemer, and Matthias Hollick. NexMon : A Cookbook for Firmware Modifications on Smartphones to Enable Monitor Mode. 2015.

[SZV19] Benn Seri, Dor Zusman, and Gregory Vishnepolsky. BLEEDINGBIT : The hidden attack surface within BLE chips. Technical report, Armis, Inc., 2019.

[TTS17] Duygu Sinanc Terzi, Ramazan Terzi, and Seref Sagiroglu. Big data analytics for network anomaly detection from netflow data. In *2017 International Conference on Computer Science and Engineering (UBMK)*. IEEE, October 2017.

[uni] Unicorn engine. `https://www.unicorn-engine.org/`. Accessed : 2020-11-17.

[USC12] Saeed Ur Rehman, Kevin Sowerby, and Colin Coghill. RF fingerprint extraction from the energy envelope of an instantaneous transient signal. *2012 Australian Communications Theory Workshop, AusCTW'12*, pages 90–95, 2012.

[VHVHN16] Tien Dang Vo-Huu, Triet Dang Vo-Huu, and Guevara Noubir. Fingerprinting Wi-Fi Devices Using Software Defined Radios. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks - WiSec '16*, pages 3–14, 2016.

[Vol] volk. https://www.libvolk.org. Accessed : 2020-08-17.

[Wei10] Ralf-Philipp Weinmann. All Your Baseband Are Belong To Us : Over-the-air exploitation of memory corruptions in GSM software stacks. In *DeepSec*, 2010.

[Wel10] Harald Welte. In six weeks from bare hardware to receiving bcchs. http://laforge.gnumonks.org/blog/20100213-six_weeks_to_bcch/, 2010. Accessed : 2020-10-19.

[Wil] Brandon Wilson. Why are we deprecating network performance features. https://techcommunity.microsoft.com/t5/core-infrastructure-and-security/why-are-we-deprecating-network-performance-features-kb4014193/ba-p/259053. Accessed : 2020-07-10.

[WM10] Harald Welte and Steve Markgraf. OsmocomBB Running your own GSM stack on a phone GSM / 3G Network Security Introduction. *27th Chaos communication congress*, (December 2010), 2010.

[zep] Zephyr projet. https://zephyrproject.org/.

[ZJZ+18] Zhou Zhuang, Xiaoyu Ji, Taimin Zhang, Juchuan Zhang, Wenyuan Xu, Zhenhua Li, and Yunhao Liu. FBSleuth : Fake Base Station Forensics via Radio Frequency Fingerprinting. pages 261–272, 2018.

[ZNKS08] Hui Zhou, Charles Nicholls, Thomas Kunz, and Howard Schwartz. Frequency accuracy & stability dependencies of crystal oscillators. *Technical Report SCE-08-12*, (November) :1–15, 2008.